

Command Line, DVCS, Text Editing, Configuration

Previously we did an interactive quickstart and then went through the basics of Linux. Today we'll introduce you to a large number of new Unix commands that you'll use throughout the rest of the quarter, including `git` and `emacs`.

The Command Line

In general, we believe the best way to understand the command line is to learn by doing. We'll go through a number of useful commands with examples, and you will get quite good by the end of this class. It is nevertheless worth buying some books to consult as references. Here are four particularly good ones:

- [Sobell's Linux Book](#): If you only get one book, get Sobell's. Probably the single best overview, and applicable to both Mac and Unix systems.
- [Unix Power Tools](#): While this came out in 2002, much of it is still relevant to the present day. New options have been added to commands but old ones are very rarely deprecated.
- [Unix/Linux Sysadmin Handbook](#): Somewhat different focus than Sobell. Organized by purpose (e.g. shutdown), and a good starter guide for anyone who needs to administer machines. Will be handy if you want to do anything nontrivial with EC2.
- [The Command Line Crash Course](#) is Zed Shaw's free online tutorial introduction to the command line. Much shorter than the others and recommended.

With those as references, let's dive right in. As always, the commands below should be executed after sshing into your EC2 machine¹. If your machine is messed up for some reason, feel free to terminate the instance in the [AWS EC2 Dashboard](#) and boot up a new one. Right now we are doing all these commands in a “vanilla” environment without much in the way of user configuration; as we progress we will set up quite a lot.

The three streams: STDIN, STDOUT, STDERR

Most `bash` commands [accept input](#) as a single stream of bytes called `STDIN` or standard input and yields two output streams of bytes: `STDOUT`, which is the standard output, and `STDERR` which is the stream for errors.

- `STDIN`: this can be text or binary data streaming into the program, or keyboard input.
- `STDOUT`: this is the stream where the program writes its data. This is printed to the screen unless otherwise specified.

¹You can probably get away with running them on your local Mac, but beware: the built-in command line tools on OS X (like `mv`) are ridiculously old BSD versions. You want to install the new ones using [homebrew](#). We'll write up an optional guide to setting up the Mac for local development if there is interest, but just be warned for now.

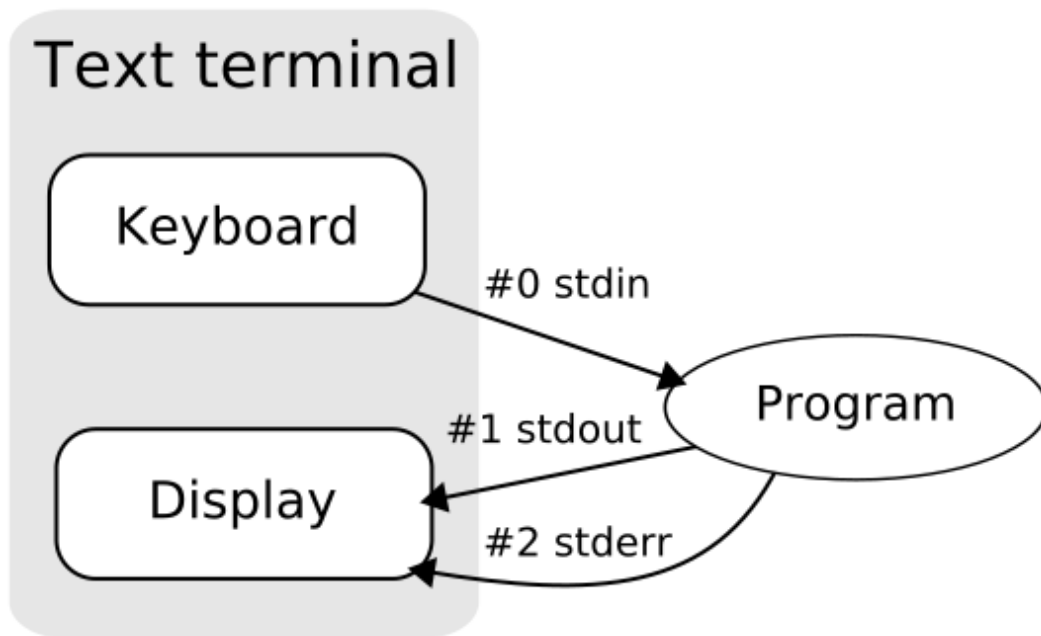


Figure 1: Visualizing the three standard streams. From [Wikipedia](#).

- **STDERR**: this is the stream where error messages are display. This is also printed to the screen unless otherwise specified.

Here are a few simple examples; execute these at your EC2 command line.

```
1 # Redirecting STDOUT
2 echo -e "line1\nline2"
3 echo -e "line1\nline2" > demo.txt
4
5 # Redirecting STDERR
6 curl fakeurl # print error message to screen
7 curl fakeurl 2> errs.txt
8 cat errs.txt
9
10 # Redirecting both to different files
11 curl google.com fakeurl 1> out1.txt 2> out2.txt
12
13 # Redirecting both to the same file
14 curl google.com fakeurl &> out.txt
15
16 # Getting STDIN from a file
17 head errs.txt
18
19 # Getting it from a pipe
20 cat errs.txt | head
21
```

```
22 # Putting it all together  
23 curl -s http://google.com | head -n 2 &> asdf.txt
```

Note what we did in the last two examples, with the `|` symbol. That's called a pipe, and the technique of connecting one command's STDOUT to another command's STDIN is very powerful. It allows us to build up short-but-powerful programs by composing individual pieces. If you can write something this way, especially for text processing or data analysis, you almost always should. . . even if it's a complex ten step pipeline. It will be incredibly fast and robust, and not that hard to understand (it's still a one-liner, albeit a long one). See this piece on [Taco Bell programming](#) and [commentary](#). With these principles in mind, let's go through some interactive examples of Unix commands. All of these should work on your default Ubuntu EC2 box.

Navigation

list files: `ls`

The most basic and heavily used command. You will use this constantly to get your bearings and confirm what things do.

```
1 ls  
2 ls --help  
3 ls --help | less # 'pipe' the output of ls into less  
4 ls -a  
5 ls -alrth  
6 alias ll='ls -alrth' # create an alias in bash  
7 ll
```

modify/create empty files: `touch`

Used to change timestamps and quickly create zero byte files as placeholders. Useful for various kinds of tests.

```
1 touch file1  
2 touch file2  
3 ll  
4 ll --full-time  
5 touch file1  
6 ll --full-time # note that the file1 modification time is updated
```

display text: `echo`

Useful for debugging and creating small files for tests.

```
1 echo "foo"  
2 man echo
```

```
3 echo -e "foo\n"
4 echo -e "line1\nline2" > demo.txt # write directly to tempfile with '>'
```

copy files: cp

Powerful command with many options, works on everything from single files to entire archival backups of huge directories.

```
1 cp --help
2 cp demo.txt demo2.txt
3 cp -v demo.txt demo3.txt # verbose copy
4 cp -a demo.txt demo-archive.txt # archival exact copy
5 ll --full-time demo* # note timestamps of demo/demo-archive
6 echo "a new file" > asdf.txt
7 cat demo.txt
8 cp asdf.txt demo.txt
9 cat demo.txt # cp just clobbered the file
10 alias cp='cp -i' # set cp to be interactive by default
```

move/rename files: mv

Move or rename files. Also extremely powerful.

```
1 mv asdf.txt asdf-new.txt
2 mv -i asdf-new.txt demo.txt # prompt before clobbering
3 alias mv='mv -i'
```

remove files: rm

Powerful command that can be used to delete individual files or recursively delete entire trees. Use with caution.

```
1 rm demo.txt
2 rm -i demo2.txt
3 rm -i demo*txt
4 alias rm='rm -i'
5 alias # print all aliases
```

symbolic link: ln

Very useful command that allows a file to be symbolically linked, and therefore in two places at once. Extremely useful for large files, or for putting in one level of indirection.

```
1 yes | nl | head -1000 > data1.txt
2 head data1.txt
```

```

3 ll data1.txt
4 ln -s data1.txt latest.txt
5 head latest.txt
6 ll latest.txt # note the arrow. A symbolic link is a pointer.
7 yes | nl | head -2000 | tail -50 > data2.txt
8 head latest.txt
9 ln -s data2.txt latest.txt # update the pointer w/o changing the
10 underlying file
11 head latest.txt
12 head data1.txt
13 head data2.txt

```

print working directory: pwd

Simple command to print current working directory. Used for orientation when you get lost.

```
1 pwd
```

create directories: mkdir

Create directories. Has a few useful options.

```

1 mkdir dir1
2 mkdir dir2/subdir    # error
3 mkdir -p dir2/subdir # ok

```

change current directory: cd

Change directories. Along with `ls`, one of the most frequent commands.

```

1 cd ~
2 cd dir1
3 pwd
4 cd ..
5 cd dir2/subdir
6 pwd
7 cd - # jump back
8 cd - # and forth
9 cd # home
10 alias ..='cd ..'
11 alias ...='cd ..; cd ..'
12 cd dir2/subdir
13 ...
14 pwd

```

remove directories: `rmdir`

Not used that frequently; usually recursive `rm` is used instead. Note that `rm -rf` is the most dangerous command in Unix and must be used with extreme caution, as it means “remove recursively without prompting” and can easily nuke huge amounts of data.

```
1 rmdir dir1
2 rmdir dir2 # error
3 rm -rf dir2 # works
```

Network

synchronize local and remote files: `rsync`

Transfer files between two machines. Run this command on your *local* machine, making the appropriate substitutions:

```
1 # on local machine
2 yes | nl | head -500 > something.txt
3 rsync -avz -e 'ssh -i /Users/balajis/.ssh/cs184-john-stanford-edu.pem' \
4 something.txt ubuntu@ec2-23-20-44-183.compute-1.amazonaws.com:~/
```

Then login and run `cat something.txt` on the EC2 machine. We’ll show later on how to make `rsync` much more convenient to use with config files.

retrieve files via `http/https/ftp`: `wget`

Very useful command to rapidly pull down files or webpages.

```
1 wget http://startup-class.s3.amazonaws.com/simple.sh
2 less simple.sh # hit q to quit
3
4 # pull a single HTML page
5 wget https://github.com/joyent/node/wiki/modules
6 less modules
7
8 # recursively download an entire site
9 wget -r -np -k -p http://www.stanford.edu/class/cs106b
```

interact with single URLs: `curl`

A bit different from `wget`, but has some overlaps. `curl` is for interacting with single URLs. It doesn’t have the spidering/recursive properties that `wget` has, but it supports a much wider array of protocols. It is very commonly used as the building block for API calls.

```

1 curl https://install.meteor.com | less # an example install file
2 curl -i https://api.github.com/users/defunkt/orgs # a simple API call
3 GHUSER="defunkt"
4 GHVAR="orgs"
5 curl -i https://api.github.com/users/$GHUSER/$GHVAR # with variables
6 GHVAR="repos"
7 curl -i https://api.github.com/users/$GHUSER/$GHVAR # with diff vars

```

send test packets: ping

See if a remote host is up. Very useful for basic health checks or to see if your computer is online. Surprisingly large number of options.

```

1 ping google.com # see if you have internet connectivity
2 ping stanford.edu # see if stanford is up

```

Text processing

view files: less

Paging utility used to view large files. Can scroll both up and down. Use q to quit.

```

1 rsync --help | less

```

print/concatenate files: cat

Industrial strength file viewer. Use this rather than MS Word for looking at large files, or files with weird bytes.

```

1 # Basics
2 echo -e "line1\nline2" > demo.txt
3 cat demo.txt
4 cat demo.txt demo.txt demo.txt > demo2.txt
5 cat demo2.txt
6
7 # Piping
8 yes | head | cat - demo2.txt
9 yes | head | cat demo2.txt -
10
11 # Download chromosome 22 of the human genome
12 wget ftp://ftp.ncbi.nih.gov/genomes/Homo_sapiens/CHR_22/hs_ref_GRCh37.p10_chr22.gbk.gz
13 gunzip hs_ref_GRCh37.p10_chr22.gbk.gz
14 less hs_ref_GRCh37.p10_chr22.gbk
15 cat hs_ref_GRCh37.p10_chr22.gbk # hit control-c to interrupt

```

first part of files: head

Look at the first few lines of a file (10 by default). Surprisingly useful for debugging and inspecting files.

```
1 head --help
2 head *gbk      # first 10 lines
3 head -50 *gbk  # first 50 lines
4 head -n50 *gbk # equivalent
5 head *txt *gbk # heads of multiple files
6 head -q *txt *gbk # heads of multiple files w/o delimiters
7 head -c50 *gbk # first 50 characters
```

last part of files: tail

Look at the bottom of files; default is last 10 lines. Useful for following logs, debugging, and in combination with head to pull out subsets of files.

```
1 tail --help
2 tail *gbk
3 head *gbk
4 tail -n+3 *gbk | head # start at third line
5 head -n 1000 | tail -n 20 # pull out intermediate section
6 # run process in background and follow end of file
7 yes | nl | head -n 10000000 > foo &
8 tail -F foo
```

extract columns: cut

Pull out columns of a file. In combination with head/tail, can pull out arbitrary rectangular subsets of a file. Extremely useful for working with any kind of tabular data (such as data headed for a database).

```
1 wget ftp://ftp.ncbi.nih.gov/genomes/Bacteria/\
2 Escherichia_coli_K_12_substr__W3110_uid161931/NC_007779.ptt
3 head *ptt
4 cut -f2 *ptt | head
5 cut -f2,5 *ptt | head
6 cut -f2,5 *ptt | head -30
7 cut -f1 *ptt | cut -f1 -d'.' | head
8 cut -c1-20 *ptt | head
```

numbering lines: nl

Number lines. Useful for debugging and creating quick datasets. Has many options for formatting as well.


```
1 nl *gbk | tail -1 # determine number of lines in file
2 nl *ptt | tail -2
```

concatenate columns: paste

Paste together data by columns.

```
1 tail -n+3 *ptt | cut -f1 > locs
2 tail -n+3 *ptt | cut -f5 > genes
3 paste genes locs genes | head
```

sort by lines: sort

Industrial strength sorting command. Very powerful standalone and in combination with others.

```
1 sort genes | less # default sort
2 sort -r genes | less # reverse
3 sort -R genes | less # randomize
4 cut -f2 *ptt | tail -n+4 | head
```

uniquify lines: uniq

Useful command for analyzing any data with repeated elements. Best used in pipelines with sort beforehand.

```
1 cut -f2 *ptt | tail -n+4 | sort | uniq -c | sort -k1 -rn #
2 cut -f3 *ptt | tail -n+4 | uniq -c | sort -k2 -rn # output number
3 cut -f9 *ptt > products
4 sort products | uniq -d
```

line, word, character count: wc

Determine file sizes. Useful for debugging and confirmation, faster than nl if no intermediate information is needed.

```
1 wc *ptt # lines, words, bytes
2 wc -l *ptt # only number of lines
3 wc -L *ptt # longest line length, useful for apps like style checking
```

split large files: split

Split large file into pieces. Useful as initial step before many parallel computing jobs.

```
1 split -d -l 1000 *ptt subset.ptt.  
2 ll subset.ptt*
```

Help

manuals: man

Single page manual files. Fairly useful once you know the basics. But Google or StackOverflow are often more helpful these days if you are really stuck.

```
1 man bash  
2 man ls  
3 man man
```

info: info

A bit more detail than `man`, for some applications.

```
1 info cut  
2 info info
```

System and program information

computer information: uname

Quick diagnostics. Useful for install scripts.

```
1 uname -a
```

current computer name: hostname

Determine name of current machine. Useful for parallel computing, install scripts, config files.

```
1 hostname
```

current user: whoami

Show current user. Useful when using many different machines and for install scripts.

```
1 whoami
```

current processes: ps

List current processes. Usually a prelude to killing a process.

```
1 sleep 60 &
2 ps xw | grep sleep
3 # kill the process number
```

Here is how that looks in a session:

```
1 ubuntu@domU-12-31-39-16-1C-96:~$ sleep 20 &
2 [1] 5483
3 ubuntu@domU-12-31-39-16-1C-96:~$ ps xw | grep sleep
4  5483 pts/0    S      0:00 sleep 20
5  5485 pts/0    S+     0:00 grep --color=auto sleep
6 ubuntu@domU-12-31-39-16-1C-96:~$ kill 5483
7 [1]+  Terminated                  sleep 20
```

So you see that the process number 5483 was identified by printing the list of running processes with `ps xw`, finding the process ID for `sleep 20` via `grep`, and then running `kill 5483`. Note that the `grep` command itself showed up; that is a common occurrence and can be dealt with as follows:

```
1 ubuntu@domU-12-31-39-16-1C-96:~$ sleep 20 &
2 [1] 5486
3 ubuntu@domU-12-31-39-16-1C-96:~$ ps xw | grep sleep | grep -v "grep"
4  5486 pts/0    S      0:00 sleep 20
5 ubuntu@domU-12-31-39-16-1C-96:~$ kill 5486
```

Here, we used the `grep -v` flag to exclude any lines that contained the string `grep`. That will have some false positives (e.g. it will exclude a process named `foogrep`, so use this with some caution.

most important processes: top

Dashboard that shows which processes are doing what. Use `q` to quit.

```
1 top
2 top -o rsize # order by memory consumption
```

stop a process: kill

Send a signal to a process. Usually this is used to terminate misbehaving processes that won't stop of their own accord.

```
1 sleep 60 &
2 kill $(ps xw | grep sleep | cut -f1 -d ' ' | head -1) # a bit dangerous
```

Superuser

become root temporarily: sudo

Act as the root user for just one or a few commands.

```
1 sudo apt-get install git-core
```

become root: su

Actually become the root user. Sometimes you do this when it's too much of a pain to type `sudo` a lot, but you usually don't want to do this.

```
1 touch normal
2 sudo su
3 touch rootfile
4 ls -alrth normal rootfile # notice owner of rootfile
```

Storage and finding

create an archive: tar

Make an archive of files.

```
1 mkdir genome
2 mv *ptt* genome/
3 tar -cvf genome.tar genome
```

compress files: gzip

Compress files. Can also view compressed `gzip` files without fully uncompressing them with `zcat`.

```
1 gzip genome.tar
2 md temp
3 cp genome.tar.gz temp
4 cd temp
5 tar -xzvf genome.tar.gz
6 cd ..
7 gunzip genome.tar.gz
8 rm -rf genome.tar genome temp
9 wget ftp://ftp.ncbi.nih.gov/genomes/Homo_sapiens/CHR_22/hs_ref_GRCh37.p10_chr22.fa.gz
10 zcat hs_ref_GRCh37.p10_chr22.fa.gz | nl | head -1000 | tail -20
```

find a file (non-indexed): find

Non-indexed search. Can be useful for iterating over all files in a subdirectory.

```
1 find /etc | nl
```

find a file (indexed): locate

For locate to work, `updatedb` must be operational. This runs by default on Ubuntu but you need to manually configure it for a mac.

```
1 locate fstab
```

system disk space: df

```
1 df -Th
```

directory utilization: du

```
1 cd ~ubuntu
2 du --max-depth=1 -b | sort -k1 -rn
```

Intermediate text processing

searching within files: grep

Powerful command which is worth learning in great detail. Go through `grep --help` and try out many more of the options.

```
1 grep protein *ptt | wc -l # lines containing protein
2 grep -l Metazoa *ptt *gbk # print filenames which contain 'Metazoa'
3 grep -B 5 -A 5 Metazoa *gbk
4 grep 'JOURNAL.*' *gbk | sort | uniq
```

simple substitution: sed

Quick find/replace within a file. You should review this outstanding set of [useful sed one-liners](#). Here is a simple example of using `sed` to replace all instances of `kinase` with `STANFORD` in the first 10 lines of a `ptt` file, printing the results to STDOUT:

```
1 head *ptt | sed 's/kinase/STANFORD/g'
```

`sed` is also useful for quick cleanups of files. Suppose you have a file which was saved on Windows:

```
1 # Convert windows newlines into unix; use if you have such a file
2 wget http://startup-class.s3.amazonaws.com/windows-newline-file.csv
```

Viewing this file in `less` shows us some `^M` characters, which are Windows-format newlines (`\r`), different from Unix newlines (`\n`).

```
1 $ less windows-newline-file.csv
2 30,johnny@gmail.com,Johnny Walker,,^M1456,jim@gmail.com,Jim Beam,,^M2076,...
3 windows-newline-file.csv (END)
```

The `\r` is interpreted in Unix as a “carriage return”, so you can’t just `cat` these files. If you do, then the cursor moves back to the beginning of the line everytime it hits a `\r`, which means you don’t see much. Try it out:

```
1 cat windows-newline-file.csv
```

You won’t see anything except perhaps a blur. To understand what is going on in more detail, read [this post](#). We can fix this issue with `sed` as follows:

```
1 ubuntu@domU-12-31-39-16-1C-96:~$ sed 's/\r/\n/g' windows-newline-file.csv
2 30,johnny@gmail.com,Johnny Walker,,
3 1456,jim@gmail.com,Jim Beam,,
4 2076,jack@stanford.edu,Jack Daniels,,
```

Here `s/\r/\n/g` means “replace the `\r` with `\n` globally” in the file. Without the trailing `g` the `sed` command would just replace the first match. Note that by default `sed` just writes its output to STDOUT and does not modify the underlying file; if we actually want to modify the file in place, we would do this instead:

```
1 sed -i 's/\r/\n/g' windows-newline-file.csv
```

Note the `-i` flag for in-place replacement. Again, it is worth reviewing this list of [useful sed one-liners](#).

advanced substitution/short scripts: `awk`

Useful scripting language for working with tab-delimited text. Very fast for such purposes, intermediate size tool.

```
1 awk -F"\t" '{print $2,$3, $3+5}' *ptt
```

To get a feel for `awk`, review this list of [useful awk one liners](#). It is often the fastest way to operate on tab-delimited data before importation into a database.

Intermediate bash

Backticks

Sometimes you want to use the results of a bash command as input to another command, but not via STDIN. In this case you can use backticks:

```
1 echo "The date is "`date`"
```

This is a useful technique to compose command line invocations when the results of one is the argument for another (e.g. one command might return a hostname like `ec2-50-17-88-215.compute-1.amazonaws.com` which can then be passed in as the `-h` argument for another command via backticks).

Running processes in the background: `foo &`

Sometimes we have a long running process, and we want to execute other commands while it completes. We can put processes into the background with the ampersand symbol as follows.

```
1 sleep 50 &
2 # do other things
```

Here is how that actually looks at the command line:

```
1 ubuntu@domU-12-31-39-16-1C-96:~$ sleep 60 &
2 [1] 5472
3 ubuntu@domU-12-31-39-16-1C-96:~$ head -2 *ptt
4 Escherichia coli str. K-12 substr. W3110, complete genome - 1..4646332
5 4217 proteins
6 ubuntu@domU-12-31-39-16-1C-96:~$ ps xw | grep 5472
7   5472 pts/0    S        0:00 sleep 60
8   5475 pts/0    S+       0:00 grep --color=auto 5472
9 ubuntu@domU-12-31-39-16-1C-96:~$
10 ubuntu@domU-12-31-39-16-1C-96:~$
11 [1]+  Done                  sleep 60
```

Note that we see `[1] 5472` appear. That means there is one background process currently running (`[1]`) and that the ID of the background process we just spawned is `5472`. Note also at the end that when the process is done, this line appears:

```
1 [1]+  Done                  sleep 60
```

That indicates which process is done. For completeness, here is what it would look like if you had multiple background processes running simultaneously, and then waited for them all to end.

```

1  ubuntu@domU-12-31-39-16-1C-96:~$ sleep 10 &
2  [1] 5479
3  ubuntu@domU-12-31-39-16-1C-96:~$ sleep 20 &
4  [2] 5480
5  ubuntu@domU-12-31-39-16-1C-96:~$ sleep 30 &
6  [3] 5481
7  ubuntu@domU-12-31-39-16-1C-96:~$
8  [1]    Done                sleep 10
9  [2]-  Done                sleep 20
10 [3]+  Done                sleep 30

```

Execute commands from STDIN: xargs

This is a very powerful command but a bit confusing. It allows you to programmatically build up command lines, and can be useful for spawning parallel processes. It is commonly used in combination with the `find` or `ls` commands.

Here is an example which lists all files under `/etc` ending in `.sh`, and then invokes `head -2` on each of them.

```

1  find /etc -name '*.sh' | xargs head -2

```

Here is an example which lists all files under `/etc` ending in `.sh`, and then invokes `head -2` on each of them.

```

1  find /etc -name '*.sh' | xargs head -2

```

Here is another example which invokes the `file` command on everything in `/etc/profile.d`, to print the file type:

```

1  ls /etc/profile.d | xargs file

```

A savvy reader will ask why one couldn't just do this:

```

1  file /etc/profile.d/*

```

Indeed, that will produce the same results in this case, and is feasible because only one directory is being listed. However, if there are a huge number of files in the directory, then `bash` will be unable to expand the `*`. This is common in many large-scale applications in search, social, genomics, etc and will give the dreaded **Argument list too long** error. In this case `xargs` is the perfect tool; see [here](#) for more.

Pipe and redirect: `tee`

Enables you to save intermediate stages in a pipeline. Here's an example:

```
1 ls | tee list.txt
```

The reason it is called `tee` is that it is like a “T-junction”, where it passes the data through while also serializing it to a file. It can be useful for backing up a file while simultaneously modifying it; see some [examples](#).

time any command: `time`

This is a `bash` built in useful for benchmarking commands.

```
1 time sleep 5
```

That should echo the fact that the command took (almost) exactly 5 seconds to complete. It is also useful to know about a more sophisticated GNU `time` command, which can output a ton of useful information, including maximum memory consumption; extremely useful to know before putting a job on a large compute cluster. You can invoke it with `/usr/bin/time`; see documentation for Ubuntu 12.04.1 LTS [here](#).

Distributed Version Control Systems (DVCS): `git`

The purpose of DVCS

To understand the purpose of `git` and Github, think about your very first program. In the process of writing it, you likely saved multiple revisions like `foo.py`, `foo_v2.py`, and `foo_v9_final.py`. Not only does this take up a great deal of redundant space, using separate files of this kind makes it difficult to find where you kept a particular function or key algorithm. The solution is something called *version control*. Essentially, all previous versions of your code are stored, and you can bring up any past version by running commands on `foo.py` rather than cluttering your directory with past versions.

Version control becomes even more important when you work in a multiplayer environment, where several people are editing the same `foo.py` file in mutually incompatible ways at the same time on tight deadlines. In 2005, Linus Torvalds (the creator of Linux) developed a program called `git` that elegantly solved many of these problems. `git` is a so-called *distributed version control system*, a tool for coordinating many simultaneous edits on the same project where byte-level resolution matters (e.g. the difference between `x=0` and `x=1` is often profound in a program). `git` replaces and supersedes previous version control programs like `cvs` and `svn`, which lack distributed features (briefly, tools for highly multiplayer programming), and is considerably faster than comparable DVCS like `hg`.

The SHA-1 hash

To understand `git`, one needs to understand a bit about the [SHA-1](#) hash. Briefly, any file in a computer can be thought of as a series of bytes, each of which is 8 bits. If you put these bytes from left to right, they can be interpreted as a (very) [large number](#). Cryptographers

have come up with a very interesting function called SHA-1 which has the following curious property: any binary number, up to 2^{64} bits, can be rapidly mapped to a 40 character long hexadecimal number, which is 160 bits, which is 20 bytes. Here is an example using python:

```
1 [balajis@jiunit:~]$python
2 Python 2.7.3 (default, May 1 2012, 23:45:52)
3 [GCC 4.2.1 Compatible Apple Clang 3.1 (tags/Applet/clang-318.0.58)] on darwin
4 Type "help", "copyright", "credits" or "license" for more information.
5 >>> import hashlib
6 >>> hashlib.sha1
7 <built-in function openssl_sha1>
8 >>> hashlib.sha1("Hello world.")
9 <sha1 HASH object @ 0x10045ddb0>
10 >>> hashlib.sha1("Hello world.").hexdigest()
11 'e44f3364019d18a151cab7072b5a40bb5b3e274f'
```

Moreover, even binary numbers which are very close map to completely different 20 byte SHA-1 values, which means $\text{SHA-1}(x)$ is very different from most “normal” functions like $\cos(x)$ or e^x for which you can expect [continuity](#). The interesting thing is that despite the fact that there are a finite number of SHA-1 values (specifically, 2^{160} possible values) one can assume that if the hashes of two numbers are unequal, then they themselves are also extremely likely to be unequal.

$$\text{SHA-1}(x_1) \neq \text{SHA-1}(x_2) \rightarrow P(x_1 \neq x_2) \geq .9999\dots$$

Here is how that works out in practice. Note that even small perturbations to the “Hello world.” string, such as adding spaces or deleting periods, result in completely different SHA-1 values.

```
1 >>> hashlib.sha1("Hello world.").hexdigest()
2 'e44f3364019d18a151cab7072b5a40bb5b3e274f'
3 >>> hashlib.sha1("Hello world. ").hexdigest()
4 '43a898d120ad380a058c631e9518c1f57d9c4edb'
5 >>> hashlib.sha1("Hello world").hexdigest()
6 '7b502c3a1f48c8609ae212cdfb639dee39673f5e'
7 >>> hashlib.sha1(" Hello world.").hexdigest()
8 '258266d16638ad59ac0efab8373a03b69b76e821'
```

Because a hash can be assumed to map 1-to-1 to a file, rather than carting around the full file’s contents to distinguish it, you can just use the hash. Indeed, you can just use a hash to uniquely² identify *any* string of bits less than 2^{64} bits in length. Keep that concept in mind; you will need it in a second.

²Is it really unique? No. But finding a collision in SHA-1 is hard, and you can show that it’s much more likely that other parts of your program will fail than that a nontrivial collision in SHA-1 will occur.

A first git session

Now let's go through a relatively simple interactive session to create a small repository ("repo") and thereby understand git. Type in these commands at your Terminal; they should also work on a local Mac, except for the `apt-get` invocations.

```
1 sudo apt-get install -y git-core
2 mkdir myrepo
3 cd myrepo
4 git config --global user.name "John Smith"
5 git config --global user.email "example@stanford.edu"
6 git init
7 # Initialized empty Git repository in /home/ubuntu/myrepo/.git/
8 git status
9 echo -e 'line1\nline2' > file.txt
10 git status
11 git add file.txt
12 git status
13 git commit -m "Added first file"
14 git log
15 echo -e "line3" >> file.txt
16 git status
17 git diff file.txt
18 git add file.txt
19 git commit -m "Added a new line to the file."
20 git log
21 git log -p
22 git log -p --color
```

That is pretty cool. We just created a repository, made a dummy file, and added it to the repository. Then we modified it and looked at the repo again, and we could see exactly which lines changed in the file. Note in particular what we saw when we did `git log`. See those alphanumeric strings after `commit`? Yes, those are SHA-1 hashes of the entire commit, conceptually represented as a single string of bytes, a “diff” that represents the update since the last commit.

```
1 ubuntu@ip-10-196-107-40:~/myrepo$ git log
2 commit 5c8c9efc99ad084af617ca38446a1d69ae38635d
3 Author: Balaji S <balajis@stanford.edu>
4 Date: Thu Jan 17 16:32:37 2013 +0000
5
6     Added a new line to the file.
7
8 commit 1b6475505b1435258f6474245b5161a7684b7e2e
9 Author: Balaji S <balajis@stanford.edu>
10 Date: Thu Jan 17 16:31:23 2013 +0000
```

```
11
12 Added first file.
```

What is very interesting is that `git` uses SHA-1 all the way down. We can copy and paste the SHA-1 identifier of a commit and pass it as an argument to a `git` command (this is a very frequent operation). Notice that `file.txt` itself has its own SHA-1.

```
1 ubuntu@ip-10-196-107-40:~/myrepo$ git ls-tree 5c8c9efc99ad084af617ca38446a1d69ae38635d
2 100644 blob 83db48f84ec878fbfb30b46d16630e944e34f205 file.txt
```

And we can use this to pull out that particular file from within `git`'s bowels:

```
1 ubuntu@ip-10-196-107-40:~/myrepo$ git cat-file -p 83db48f84ec878fbfb30b46d16630e944e34f205
2 line1
3 line2
4 line3
```

The point here is that `git` tracks *everything* with a SHA-1 identifier. Every alteration you make to the source code tree is tracked and can be uniquely addressed and manipulated with a SHA-1 identifier. This allows very interesting kinds of computations using `git`.

Pushing to Github.com

While you can use `git` purely in this kind of local mode, to collaborate with others and protect against data loss you'll want to set up a centralized repository. You can certainly host your own repository, but it will be bare bones and yet another service to maintain. Or you can use one of many `git` *hosting services*, such as Bitbucket, Google Code, Sourceforge, or Github. The last will be our choice for this class. You can think of `github.com` as a web frontend to `git`, but it is now much more than that. It's one of the largest open source communities on the web, and the github.com website is a big deal³ in its own right. With that background, let's create a web version of our repo and push this content to github. Go to github.com/new and initialize a new repo as shown:

³At some point you should go through github's [online tutorial](https://github.com) content to get even more comfortable with `git`; the learn.github.com/p/intro.html page is a good place to begin, following along with the commands offline. This will complement the instructions in this section. Once you've done that, go to help.github.com and do the four sections at the top to get familiar with how the `github.com` site adds to the `git` experience.

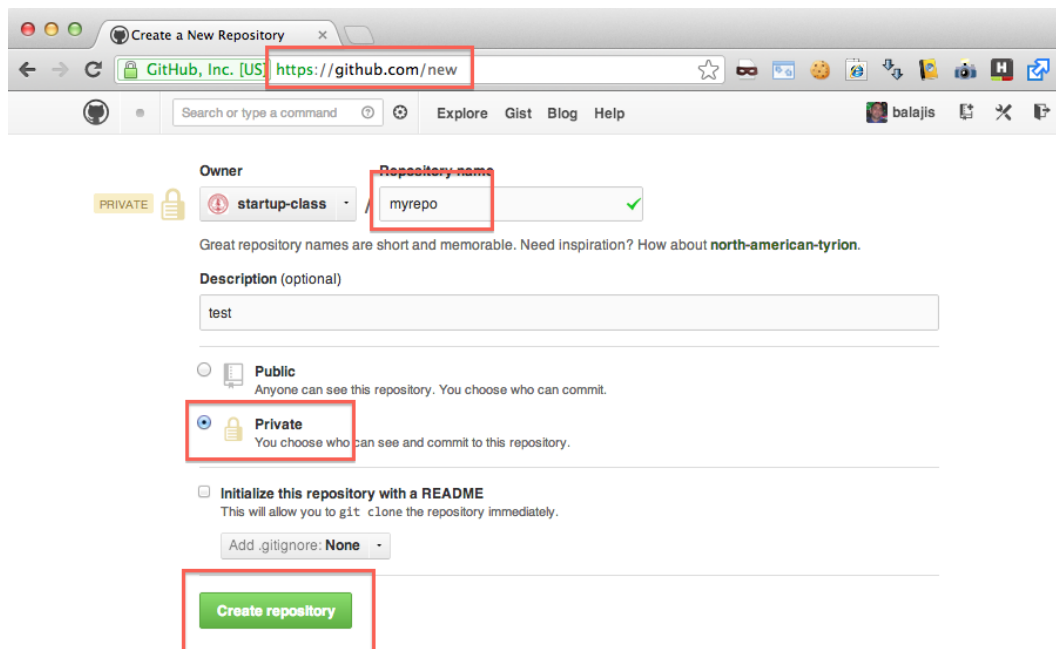


Figure 2: Creating a new repo at github.com/new.

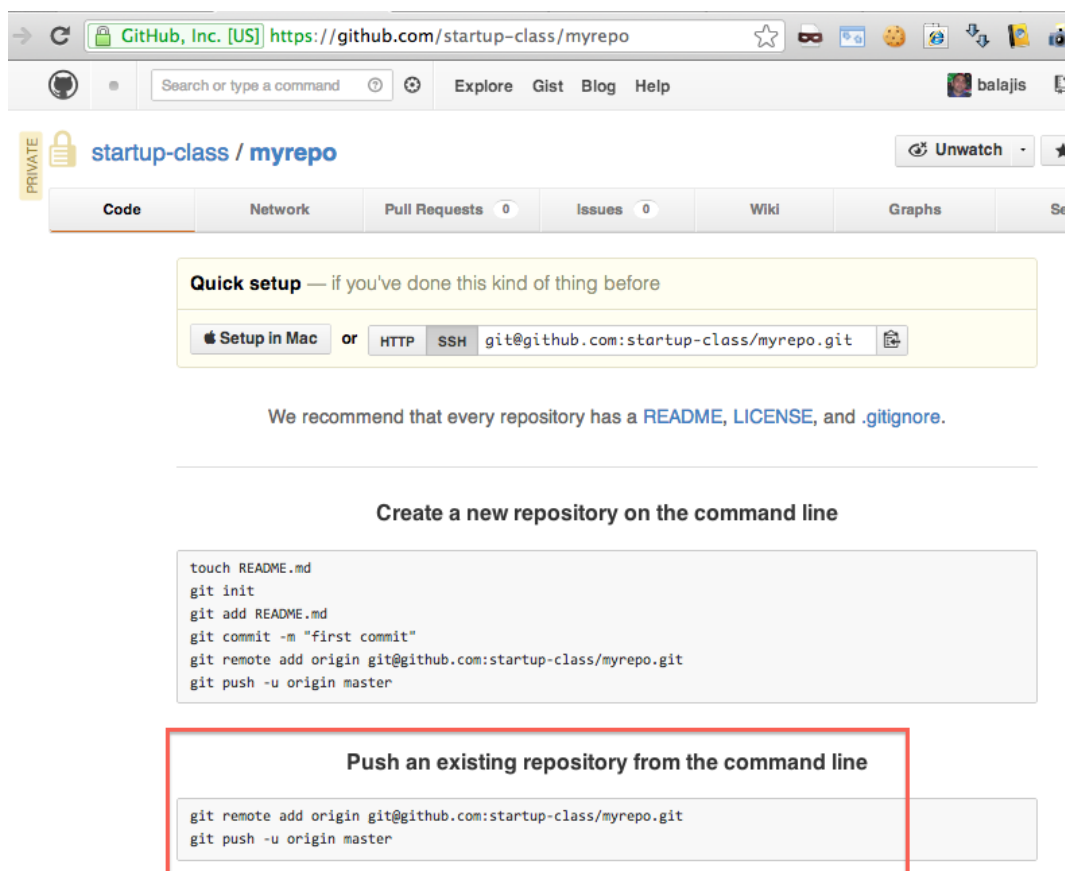


Figure 3: Instructions after a new repo is created. Copy the first line in the highlighted box.

Now come back to EC2 and run these commands:

```
1 git remote add origin git@github.com:startup-class/myrepo.git
2 git push -u origin master # won't work
```

The second command won't work. We need to generate some `ssh` keys so that github knows that we have access rights as [documented here](#). First, run the following command:


```
1 cd $HOME
2 ssh-keygen -t rsa -C "foo@stanford.edu"
```

Here's what that looks like:

[illegible]

Figure 4: *Generating an ssh-key pair.*

```
ubuntu@ip-10-196-107-40:~$ cd .ssh/  
ubuntu@ip-10-196-107-40:~/..ssh$ cat id_rsa.pub  
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQDNNgDElEraSAZAVrkcyj6gYYAnZkEefuMn/V0py6yflL4sdU8+  
F  
t  
V  
-----BEGIN PUBLIC KEY-----  
EvI30eSazr4yR3dAsKxZ7fzmsanymt balajis@stanford.edu  
ubuntu@ip-10-196-107-40:~/..ssh$
```



Confirm password to continue

Password (Forgot password)

Confirm password

Figure 7: *You will see a prompt for your github password.*

Now come back to the command line and type in:

```
1 ssh -T git@github.com
```

```
ubuntu@ip-10-196-107-40:~/.ssh$ ssh -T git@github.com
Hi balajis! You've successfully authenticated, but GitHub does not provide shell access.
```

Figure 8: *You should see this if you've added the SSH key successfully.*

Now, finally you can do this:

```
1 git push -u origin master # will work after Add Key
```

```
ubuntu@ip-10-196-107-40:~/.ssh$ cd ..
ubuntu@ip-10-196-107-40:~$ cd myrepo/
ubuntu@ip-10-196-107-40:~/myrepo$ git push -u origin master
Counting objects: 6, done.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (6/6), 458 bytes, done.
Total 6 (delta 0), reused 0 (delta 0)
To git@github.com:startup-class/myrepo.git
 * [new branch]      master -> master
Branch master set up to track remote branch master from origin.
```

Figure 9: *Now you can run the command to push to github.*

And if you go to the URL you just set up, at `https://github.com/$USERNAME/myrepo`, you will see a git repository. If you go to `https://github.com/$USERNAME/myrepo/commits/master` in particular, you will see your Gravatar next to your commits. Awesome. So we've just linked together EC2, Linux, git, Github, and Gravatar.

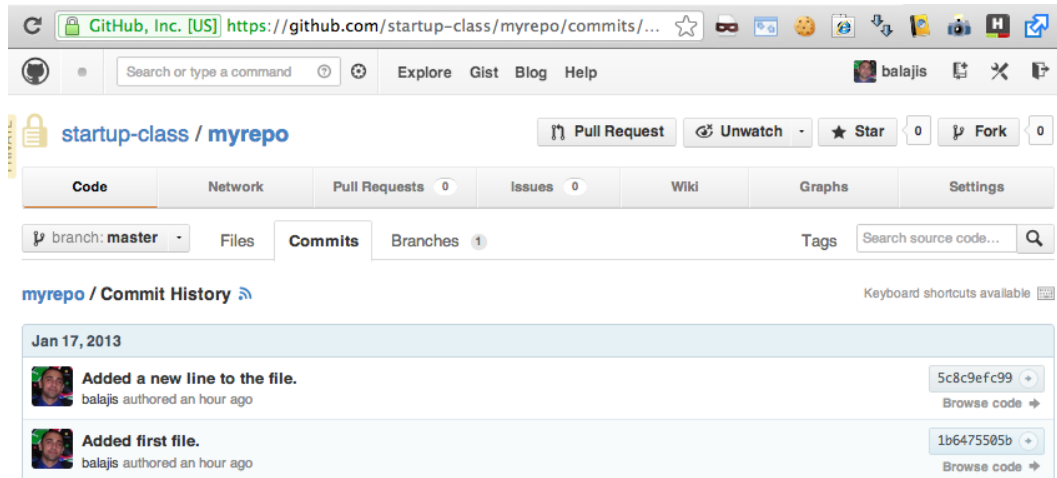


Figure 10: Now you can see your command line commits on the web, and share them with classmates.

Congratulations. That's the basics of `git` right there: creating a repository, adding files, making commits, and pushing to a remote repository. We will be using `git` quite a bit over the course, and will get into intermediate usage soon, but in the interim it will be worth your while to go a bit deeper on your own time. There are a number of excellent references, including the official [book](#) and [videos](#) and this [reference](#) for advanced users.

Text Editing: emacs

Intro

A text editor is different from a word processor like Microsoft Word in that it allows you to directly manipulate the raw, unadorned string of bytes that make up a file. That's why they are used by programmers, while word processors are employed by end users.

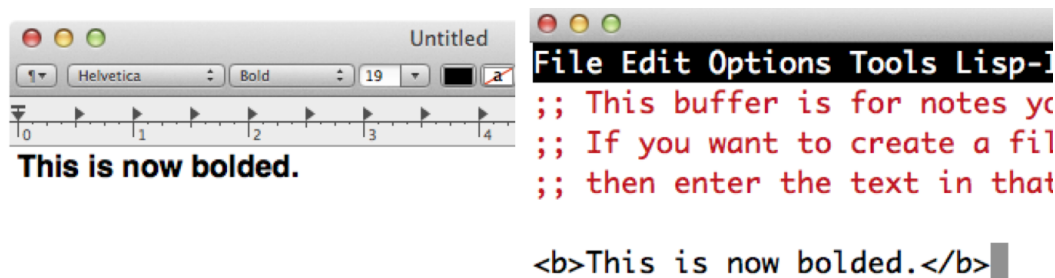


Figure 11: A word processor (left) vs. a text-editor (right). The character escapes to bold text are explicit in a text editor and hidden in a word processor.

Many people use GUI-based text editors like [Textmate](#), [Sublime Text](#), [Visual Studio](#), [XCode](#), or [Eclipse](#). These editors are in many ways quite good, but don't support all of the following properties simultaneously:

- free
- open-source
- highly configurable
- large dev community
- editing modes for virtually every language
- external tool integration
- ultra cross platform
- embeddable in other programs
- fast boot time
- lightweight memory footprint

The two editors that do have all these features are `vim` (<http://vim.org>) and `emacs` (<http://www.gnu.org/software/emacs>). If you are a *real* engineer, you probably want to learn one of these two⁴. In general, `vim` is more suitable for sysadmins or people who must manually edit many little files. It has excellent defaults and is very fast for pure text-editing. By contrast, `emacs` is greatly preferred for the data scientist or academic due to features like [orgmode](#) and strong read-evaluate-print-loop ([REPL](#)) integration for interactive debugging sessions. For intermediate tasks like web development, one can use either editor, but the REPL integration features of `emacs` carry the day for beginners in our view.

Keybindings for both editors are built into Unix in many ways. The `emacs` keybindings are used by default in any application that uses the [readline](#) library (like `bash`), while `vim` keybindings are used in commands like `less`, `top`, and `visudo`. The major advantage of investing in one of these editors is that you can get exactly the same editing environment on your local machine or any server that you connect to regularly.

AWS Preliminaries: A fresh EC2 web server

Let's first get a clean instance which can serve up some webpages. Go to your [AWS dashboard](#) and terminate your old instance(s) by going to "Instances", right-clicking on it, and choosing "Terminate". Then follow [these instructions](#) to allow your EC2 instances to respond to HTTP requests. Specifically, set the security group to allow connections on port 80 as shown:

⁴The one editor out there that might be a serious contender to the `vim`/`emacs` duopoly is the new [Light Table](#) editor by Chris Granger. A key question will be whether it can amass the kind of open-source community that `emacs` and `vim` have built, or alternatively whether its business model will provide enough funds to do major development in-house.

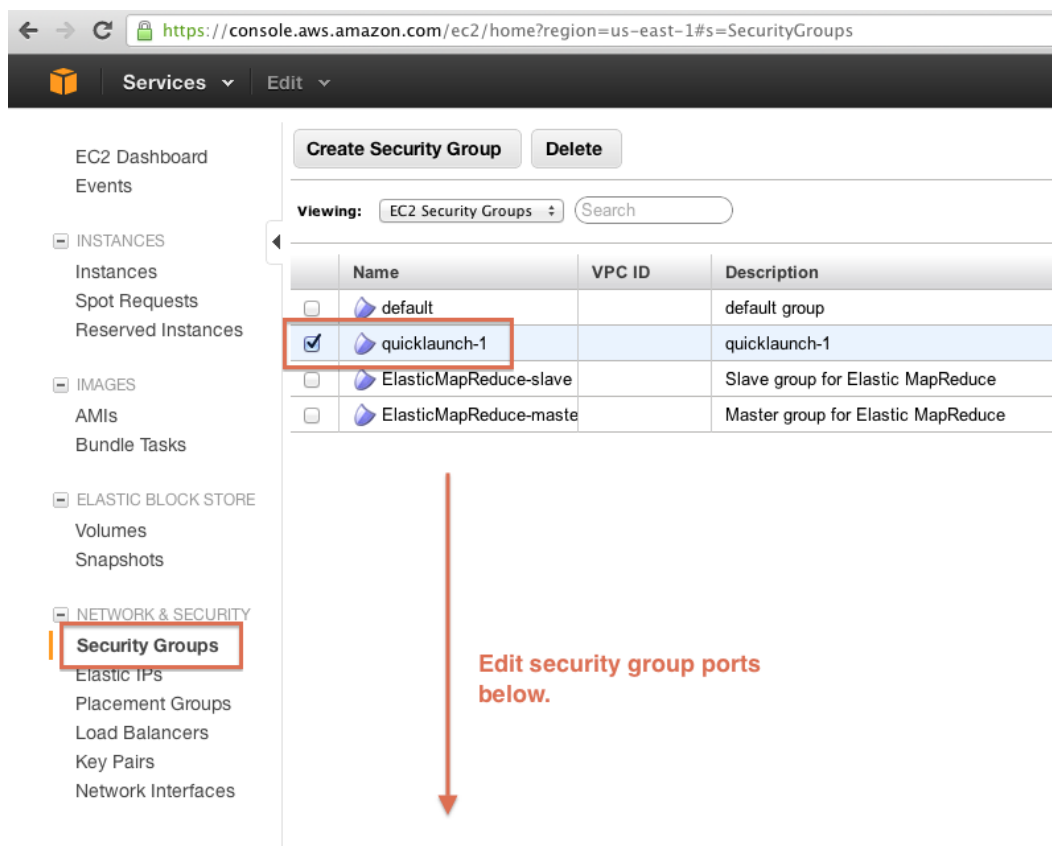


Figure 12: Click on Instances, and then the quicklaunch-1 group.

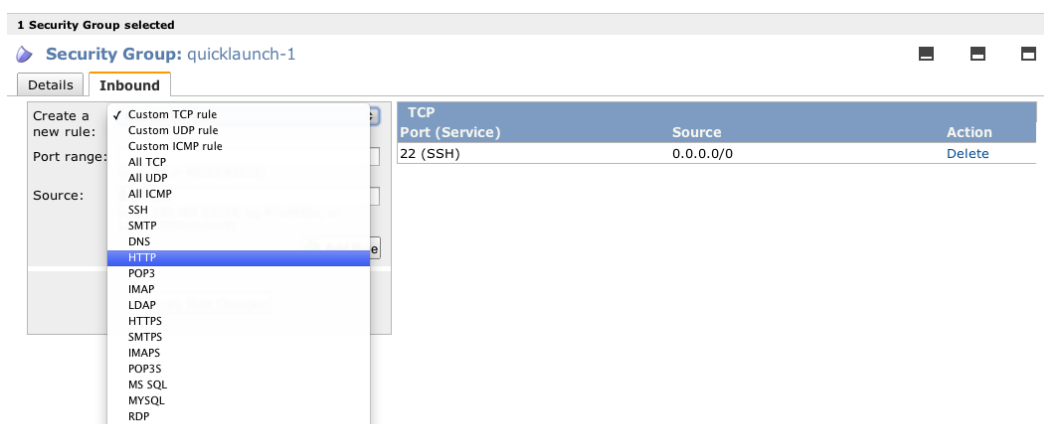


Figure 13: Select the HTTP protocol in the dropdown.

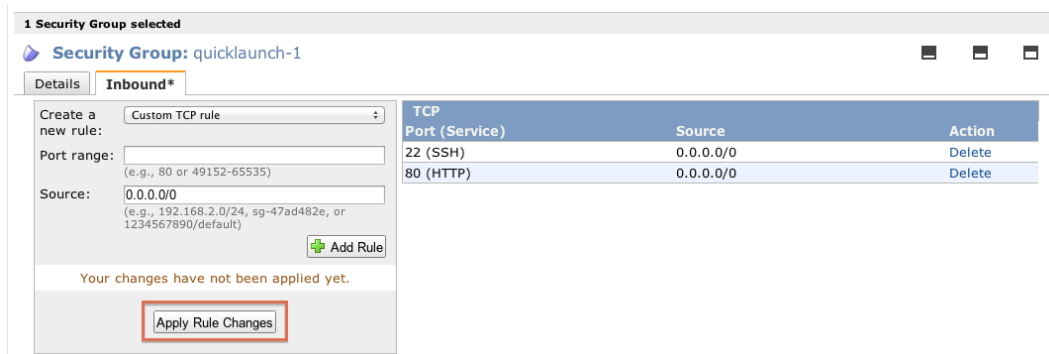


Figure 14: *Click Apply Rule Changes.*

Now launch a new instance via the “Launch Instance” button, with the same options as in the [interactive start](#) lecture (e.g. using Quick Launch with your existing keypair, booting up a t1.micro running Ubuntu 12.04.1 LTS). *Make sure to use the same security group you just edited* (e.g. quicklaunch-1), as shown:

Create a New Instance Cancel

Ubuntu Server 12.04.1 LTS (ami-3d4ff254)
Platform: Ubuntu
Architecture: x86_64
Ubuntu Server 12.04.1 LTS with support available from Canonical
(<http://www.ubuntu.com/cloud/services>).

Please review your settings and click **Launch** to finish or **Edit details** to make changes.

Instance Details

Name:	Type: t1.micro
Detailed Monitoring: No	Availability Zone: No preference
Shutdown Behaviour: Stop	Termination Protection: No
Launch into a VPC: No	

Security Details

Key Pair: cs184-john-stanford-edu	Security Group: quicklaunch-0
-----------------------------------	-------------------------------

Advanced Details

Kernel ID: Default	Ramdisk ID: Default
User Data:	IAM Role:

[Go Back](#) Edit details Launch

Figure 15: *Click Edit Details after the first Quick Launch screen*

Create a New Instance

Cancel

Ubuntu Server 12.04.1 LTS (ami-3d4ff254)

Platform: Ubuntu
Architecture: x86_64

Ubuntu Server 12.04.1 LTS with support available from Canonical
(<http://www.ubuntu.com/cloud/services>).

Click **Save details** in order to save your changes and return to the review screen.

☐ Instance Details

☐ Modify Tags

☒ Security Settings

☐ Advanced Details

☐ Storage Device Configuration

Security groups determine whether a network port is open or blocked on your instances. You may use an existing security group, or we can help you create a new security group to allow access to your instances.

☐ Create new Security Group

☒ Select Existing Security Groups

default
quicklaunch-1
ElasticMapReduce-slave
quicklaunch-0

(Selected groups: quicklaunch-1)

Save details

Launch

Figure 16: *Select quicklaunch-1 (the group for which you opened port 80) and Save Details*

Finally, after you boot the instance, execute this command to install the Apache web server.

```
1 sudo apt-get install -y apache2
```

Your new instances will now be able to serve up webpages placed in `/var/www` (more [details](#) here). This will come in handy in a second.

Keyboard Preliminaries: The Control and Meta Keys

To use `emacs` properly you need to set up your control and meta keys. On a Mac, you can configure Terminal Preferences to have “option as meta” and Keyboard Preferences to swap⁵ the [option](#) and [command](#) keys, and set Caps Lock to Control, as shown:

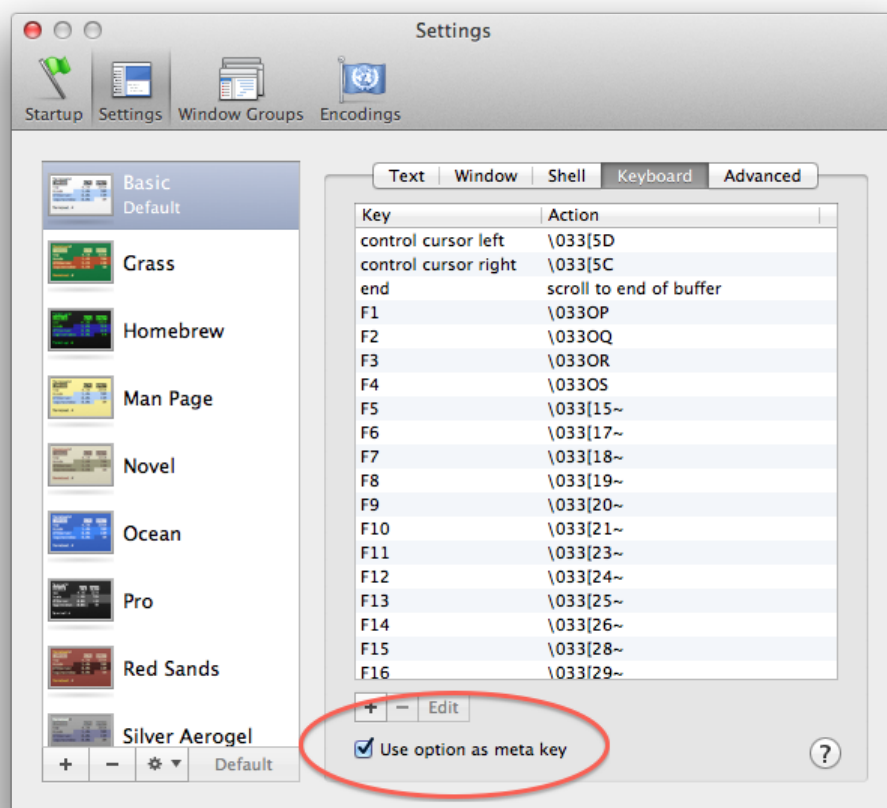


Figure 17: Click *Settings* (in the top row of buttons) and then “*Keyboard*”. Click “*Use option as meta key*”. You’ll need this for *emacs*.

⁵In theory there should be a way to set the command key as `emacs` Meta, and leave both Terminal Preferences and Keyboard Preferences alone, as shown in the [snippet](#) here. This reportedly [works](#) for some Macs but not others; if you get it to work robustly do get in touch.

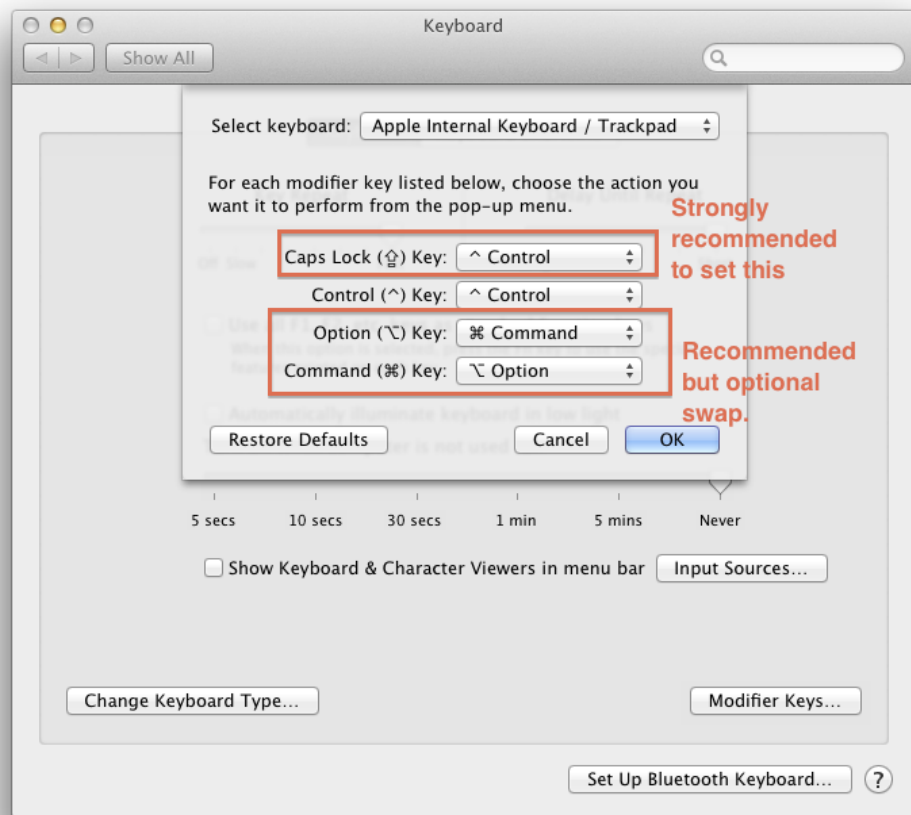


Figure 18: Switch Caps-lock and Control, and (at your discretion) switch Option and Command. The latter will make emacs much easier to use as you'll be hitting Option/Meta a lot.

On Windows you will want to use a [keyboard remapper](#) and possibly use [Putty](#) if you have troubles with Cygwin. Once you have things set up, we will use the following notation going forward for key combinations using Control or Meta:

- C-a: press Control and A simultaneously
- M-d: press Meta and D simultaneously
- C-u 7: press Control and U together, then press 7
- C-c C-e: press Control and C simultaneously, let go, and then press Control and E simultaneously

You get the idea.

Installing Emacs

Now let's get started. You will want the latest emacs, version 24. On a Mac you can install this from [emacsformacosx.com](#), while here are the instructions for [Ubuntu](#):

```
1 sudo apt-add-repository ppa:cassou/emacs
2 sudo apt-get update
3 sudo apt-get install -y emacs24 emacs24-el emacs24-common-non-dfsg
```

Watch this screencast of [emacs installation](#) if you have problems.

Emacs basics

To get started with emacs, watch the [second screencast](#). This shows you how to launch and quit emacs, and how to start the online tutorial. Notice in particular the command combinations streaming by in the lower right corner. The first thing you want to do is go through M-x `help-with-tutorial` and start learning the keyboard shortcuts. With emacs, you will never again need to remove your hands from the touch typing position. You will also want to keep this [reference card](#) handy as you are learning the commands.

Emacs major modes

Now let's edit some files. Watch the [third screencast](#), where we are executing these commands and then editing the buffers.

```
1 # First confirm you did "sudo apt-get install -y apache2" and opened up HTTP
2 # in the quicklaunch-1 Security Group
3
4 # Then make /var/www/ writeable and go to that directory
5 sudo chown ubuntu:ubuntu -R /var/www
6 cd /var/www
7
8 # Open up emacs in 'no window' (-nw) mode with three files.
9 emacs -nw index.html demo.css demo.js
10 # Now we edit
```

You will be typing in the following three files, using emacs keybindings.

/var/www/index.html

```
1 <html>
2   <head>
3     <title>From Scratch</title>
4     <script type="text/javascript" src="demo.js"></script>
5     <link type="text/css" rel="stylesheet" href="demo.css" />
6   </head>
7   <body>
8     <p>Hello world.</p>
9   </body>
10 </html>
```

/var/www/demo.css

```
1 p { font-weight: bold;}
```

/var/www/demo.js

```
1 alert("Your first alert.");
```

After editing the files, browse to your EC2 instance via HTTP (e.g. <http://ec2-50-17-88-215.compute-1.amazonaws.com> in your browser). If you configured your Security Groups correctly as in the “Preliminaries” section, and if you edited the files in the same way

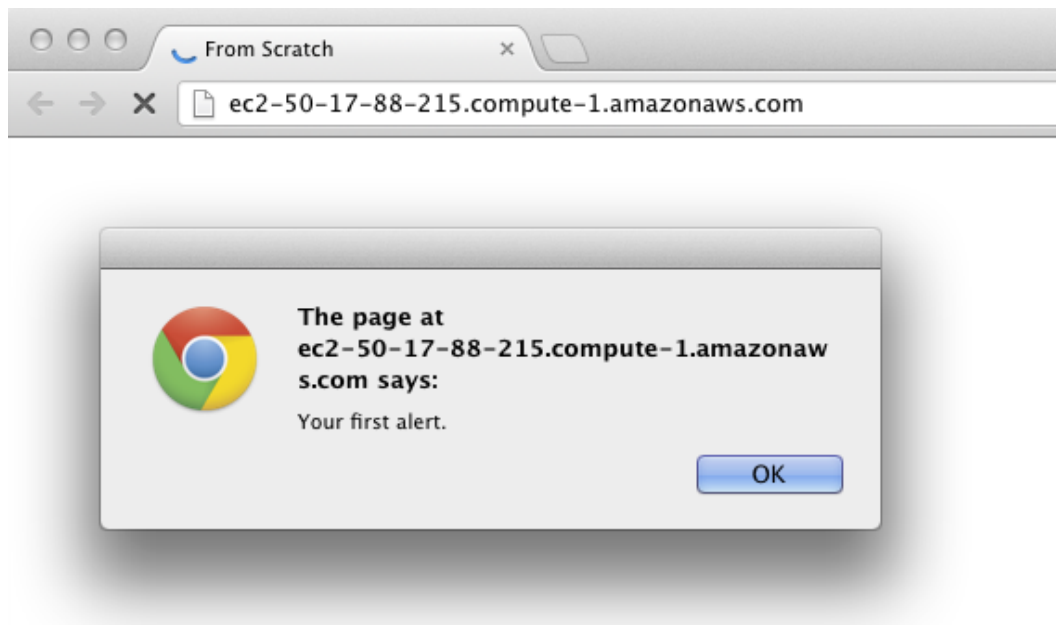


Figure 19: *If you set up your security groups right, you should see this alert first upon browsing to the DNS name of your EC2 instance.*

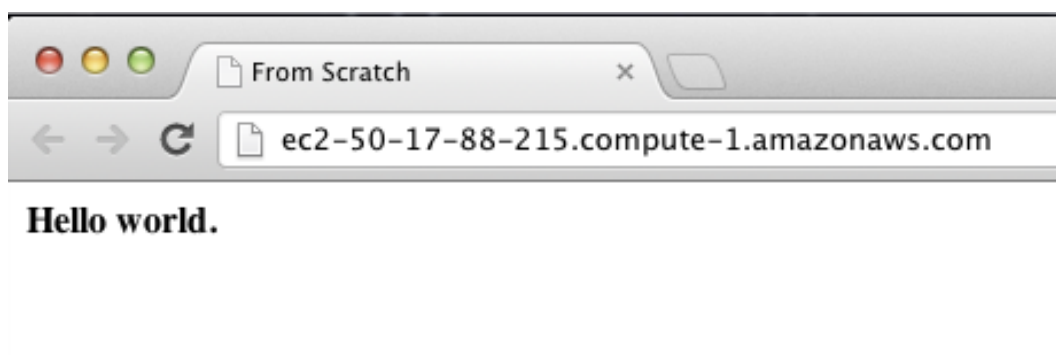


Figure 20: *Then you should see this HTML content.*

This illustrates how to launch emacs at the command line, the use of major modes for HTML/CSS/JS, opening and switching windows, finding online help (with `M-x describe-mode`, `M-x describe-command`, `M-x apropos`), and opening/creating/saving files. You should practice editing files and use the [reference card](#) to make sure you master the basics. Later on we'll go through advanced emacs and shell configuration.

Summary

Up to this point you've learned the basics of connecting to and administering an Ubuntu EC2 instance (`ssh`, `bash`, `screen`, `apt-get`). Next, you learned a large number of Unix commands. You familiarized yourself with `git` and created and pushed your first repos to github. Finally, you learned the basics of `emacs`. Good progress.