CS506/CS606 (Research Programming)
The UNIX programming environment

Kyle Gorman
Center For Spoken Language Understanding
Oregon Health & Science University

Contents

1 Background

UNIX-like operating systems (including Linux and Mac OS X) are now the dominant tools for software development and quantitative research.

1.1 The UNIX philosophy

The first idea of UNIX is a preference for simplicity, as articulated in R.P. Gabriel's 1991 essay, *Lisp: Good News, Bad News, How to Win Big*.

> I and just about every designer … has had extreme exposure to the MIT/Stanford style of design. The essence of this style can be captured by the phrase *the right thing*. To such a designer it is important to get all of the following characteristics right:

- Simplicity: the design must be simple, both in implementation and interface. It is more important for the interface to be simple than the implementation.
- Correctness: the design must be correct in all observable aspects. Incorrectness is simply not allowed.
- Consistency: the design must not be inconsistent. A design is allowed to be slightly less simple and less complete to avoid inconsistency. Consistency is as important as correctness.
- Completeness: the design must cover as many important situations as is practical. All reasonably expected cases must be covered. Simplicity is not allowed to overly reduce completeness.

[Correctness, Consistency ≫ Completeness ≫ Simplicity–KG]

> […] The worse-is-better philosophy is only slightly different:
>
> - Simplicity: the design must be simple, both in implementation and interface. It is more important for the implementation to be simple than the interface. Simplicity is the most important consideration in a design.
> - Correctness: the design must be correct in all observable aspects. It is slightly better to be simple than correct.
> - Consistency: the design must not be overly inconsistent. Consistency can be sacrificed for simplicity in some cases, but it is better to drop those parts of the design that deal with less common circumstances than to introduce either implementational complexity or inconsistency.
> - Completeness: the design must cover as many important situations as is practical. All reasonably expected cases should be covered. Completeness can be sacrificed in favor of any other quality. In fact, completeness must be sacrificed whenever implementation simplicity is jeopardized. Consistency can be sacrificed to achieve completeness if simplicity is retained; especially worthless is consistency of interface.
>
> Early Unix and C are examples of the use of this school of design, and I will call the use of this design strategy the New Jersey approach.

[Simplicity ≫ Correctness, Consistency ≫ Completeness–KG]

But why is the New Jersey style better? Gabriel's idea is simple: "quality does not necessarily increase with functionality." New Jersey style produces simpler software. Simpler software is done sooner, and more quickly adopted by the community, than a complex (and incomplete) system.

The way this is implemented in UNIX is nicely described by M.D. McIlroy:

> Write programs that do one thing and do it well. Write programs to work together. Write programs that handle text streams, because that is a universal interface.

In other words, everything is a file.

1.2 A very brief history of UNIX

- 1969: a team at Bell Labs begin developing a simple operating system they called UNICS (UNiplexed Information and Computing Service); renamed UNIX, the first version is completed in 1970. Due to some creative antitrust litigation from the Eisenhower administration, Bell is forced to license UNIX to anyone who asks nicely.
- 1977: the University of California, Berkeley begins to produce and distribute a free variant of UNIX they called the Berkeley Software Distribution (BSD); direct descendants of this system are now available in FreeBSD, OpenBSD, NetBSD, etc.
- 1982: Sun Microsystems begins selling workstations running SunOS, a variant of BSD.
- 1983: Richard Stallman, a programmer at the MIT Artificial Intelligence Laboratory, founded the GNU project and announced plans to make a "copyleft" UNIX-compatible operating system.
- 1985: Steve Jobs is forced out of Apple Computer, and founds NeXT. NeXT forks BSD to create NeXTSTEP.
- 1991: Linus Torvalds, a Finnish student, began developing the a UNIX-like kernel called Linux, and adopted components (and the license) from GNU.
- 1996: Apple Computer purchases NeXT—and rehires Jobs—and they change the name from NeXTSTEP to Macintosh OS X.
- 2005: Android, Inc. is purchased by Google, and begins developing a mobile device operating system based on Linux.
- 2007: Apple unveils the iPhone, running the iOS operating system, based on the Darwin kernel (which ultimately derives from BSD).

If you use Mac OS X, Linux, or a smartphone, you probably use a UNIX derivative.

2 The UNIX environment

Sidebar: How to learn about UNIX

- Open a terminal window and type in all these commands (really!); it builds muscle memory and self-confidence.
- Embrace the "heuristic" nature of your own knowledge: You don't need to understand how every piece of the puzzle works, just where to look for solutions.
- Stay patient and assist others if you already know some of the material (particularly in the first two weeks).
- Read the `man` ("manual") pages (e.g., `man man`) to find out more.

Terminal.app improvements (Mac OS X users only)

We will be working today in the terminal. You may want to make it look a bit nicer.

1. Launch Terminal.app in one of the following ways:
   - Use Finder.app to navigate to `/Applications/Utilities/`
   - Use Launchpad
   - Use Spotlight
2. Paint it black:
   (a) Press Command + Comma to launch the Preferences pane.
   (b) Click on "Settings" in the top menu.
   (c) Click on "Pro" in the left menu ("Profiles").
   (d) To make this the default for any newly opened terminal window, click on the button marked "Default" at the bottom of the left menu.
3. Set up log-in and log-out features:
   (a) In the Preferences pane, click on "Shell" in the top menu.
   (b) In the dropdown box labeled "When the shell exists:", select "Close if shell exited cleanly".
   (c) Click on "Startup" in the top menu.
   (d) For "Shells open with:', select "Default login shell".
   (e) Close the Preferences pain.
   (f) Type `exit` in the terminal window, and hit return.
4. Pretty colors: execute

```
echo "export CLICOLOR=1" >> ~/.profile
```

in the terminal window, and hit return.

In what follows, I will use '$' to indicate the start of a command-line directive: your prompt probably looks different. Also, I will use `...` to indicate that I have abridged some output.

I also assume that you're using the `bash` shell. Since this is the default on nearly all UNIX-like systems, if you don't know which shell you're using, chances are it's `bash`.

2.1 Terminal shortcuts

- To execute a command, simply type it at the prompt and hit Return.
- To scroll backwards through recent commands you've executed (the "history"), push the Up arrow-key; hitting Return will execute the command again.
- To perform an interactive string search through your history, hit Ctrl+r; once again, hitting Return will execute the command again.
- When typing in the name of an command or a file, hit the Tab key. This will perform "autocompletion" of the command or file. If there is no unique autocompletion (i.e., if there is more than one command or file with the prefix you've entered so far), hit Tab a second time to see a list of all possible completions.
- To jump to the beginning of a command (i.e., to edit it), hit Ctrl+a; to jump to the end of a command, hit Ctrl+e.
- The left and right arrow keys can used to move the cursor around on the command line. To cut all text up to the cursor, hit Ctrl+u; to cut all text from the cursor to the end of the line, hit Ctrl+k. Hit Ctrl+y to paste.
- To halt a running command, hit Ctrl+c. This sends the `SIGINT` signal to the current process, which will usually kill the process.

There are many more such shortcuts; this are just some of the most common.

2.2 The who and where

- *What machine am I working on?*

  ```
  $ hostname
  CHAMP.local
  ```

  Remember, you could be typing on your laptop but connected to another machine (e.g., via SSH), so always check this before doing something drastic.

- *What directory am I in?* When you open a terminal window, you will usually be in your home directory. This will usually have a name like one of the following.

  - `/Users/gormanky`
  - `/home/gormanky`
  - `/u/gormanky`

  To get your current directory, use the `pwd` ("print working directory") command:

```
$ pwd
/Users/gormanky
```

- *Who am I?*

```
$ whoami
gormanky
```

- *Who else is working on this system?*

```
$ users
gormanky rstites
```

2.3 Files & directories

- To create a directory, use `mkdir` ("make directory").

```
$ mkdir foo
$ mkdir foo
mkdir: baz: File exists
$ mkdir -p foo
```

- To list files, use `ls` ("list").

```
$ ls
foo
$ ls foo
baz
```

  Other `ls` flags worth learning about.

  - `-t` (sort by time modified)
  - `-a` (show hidden files/directories).

- To move yourself around, use `cd` ("change directory").

```
$ cd foo
$ pwd
/Users/gormanky/Desktop/scratch/foo
$ cd ..
$ pwd
```

```
/Users/gormanky/Desktop/scratch
$ cd .
$ pwd
/Users/gormanky/Desktop/scratch
```

- To make an empty file, use `touch`.

```
$ touch baz
$ ls
baz foo
```

  If the file already exists, `touch` updates the file modifications and access times to the current time.

- To move files or directories, use `mv` (“move”).

```
$ mv foo bar
$ mv baz qux
$ ls
bar qux
```

- To remove a file, use `rm` (“remove”).

```
$ rm qux
$ ls
bar
```

- To remove a directory, use `rm -r` (“ remove recursively’).

```
$ rm bar
rm: bar: is a directory
$ rm -r bar
```

- To copy a file, use `cp`.

```
$ mkdir foo
$ touch foo/bar
$ cp foo/bar foo/baz
$ ls foo
bar baz
```

- To copy a directory use `cp -R` (“copy Recursively”).

```
$ cp foo qux
cp: foo is a directory (not copied).
$ cp -R foo qux
$ ls qux
bar baz
```

Note that it's `mv -r`, but `cp -R`. The reason for this is purely historical.

- Character ranges and wildcards:

```
$ touch 01 02 03 05 07 11 13 17 19
$ ls 0[1-3]
01 02 03
$ ls 1*
11 13 17 19
```

- To find the size of a file, use `ls -l -h` ("list, long output, human-readable").

```
$ ls -l -h gorman-dissertation.pdf
-rw-r--r--  1 gormanky  staff   648K Jun  1  2013 gorman-
   dissertation.pdf
```

- The size of a directory is *not* the sum of the sizes of the contents: for that, use `du -s -h` ("disk usage, shallow depth, human-readable").

```
$ du -d 0 -h ~/Music
 60G      /Users/gormanky/Music/
```

- To learn more about a file's format, use `file`:

```
$ file 01 gorman-dissertation.pdf
01: empty
gorman-dissertation: PDF document, version 1.5
```

2.4 Streams & pipes

Before we begin, let's grab a text file (a transcript of *The Importance of Being Earnest*), using the `curl -O` ('save output') command (NB: the flag is "capital O", not zero.)

```
curl -O http://www.openfst.org/twiki/pub/GRM/NGramQuickTour/
   earnest.txt
...
```

- To see the first few lines of a file, use `head`.

```
$ head earnest.txt
MORNING ROOM IN ALGERNON S FLAT IN HALF MOON STREET
THE ROOM IS LUXURIOUSLY AND ARTISTICALLY FURNISHED
THE SOUND OF A PIANO IS HEARD IN THE ADJOINING ROOM
DID YOU HEAR WHAT I WAS PLAYING LANE
I DIDN T THINK IT POLITE TO LISTEN SIR
I M SORRY FOR THAT FOR YOUR SAKE
I DON T PLAY ACCURATELY ANY ONE CAN PLAY ACCURATELY BUT I
   PLAY WITH WONDERFUL EXPRESSION
```

```
AS FAR AS THE PIANO IS CONCERNED SENTIMENT IS MY FORTE
I KEEP SCIENCE FOR LIFE
AND SPEAKING OF THE SCIENCE OF LIFE HAVE YOU GOT THE CUCUMBER
    SANDWICHES CUT FOR LADY BRACKNELL
$ head -2 earnest.txt
MORNING ROOM IN ALGERNON S FLAT IN HALF MOON STREET
THE ROOM IS LUXURIOUSLY AND ARTISTICALLY FURNISHED
```

- Similarly, there's `tail`.

```
$ tail -2 earnest.txt
MY NEPHEW YOU SEEM TO BE DISPLAYING SIGNS OF TRIVIALITY
ON THE CONTRARY AUNT AUGUSTA I VE NOW REALISED FOR THE FIRST
   TIME IN MY LIFE THE VITAL IMPORTANCE OF BEING EARNEST
$ tail +2 earnest.txt
THE ROOM IS LUXURIOUSLY AND ARTISTICALLY FURNISHED
THE SOUND OF A PIANO IS HEARD IN THE ADJOINING ROOM
...
ON THE CONTRARY AUNT AUGUSTA I VE NOW REALISED FOR THE FIRST
   TIME IN MY LIFE THE VITAL IMPORTANCE OF BEING EARNEST
```

- The `tail` utility has a special `-f` flag which waits for additional input, and so is good for "monitoring" a file.

```
$ tail -f program.log
...
Starting epoch  2.
Epoch  2 accuracy: 0.9061.
Epoch  2 time elapsed: 3962s.
Starting epoch  3.
...
```

- To redirect the output of a program to a file, use `>`.

```
$ head -1 earnest.txt > top
$ tail -1 earnest.txt > bottom
```

  UNIX systems have three pre-defined communication channels ("standard streams"): standard input (`stdin`; file descriptor 0), standard output (`stdout`; file descriptor 1), and standard error (`stderr`; file descriptor 2). The `>` literally means "redirect information intended for standard output and put it in the following file instead".

- To combine two or more files, use `cat` ("concatenate").

```
$ cat top bottom > ends
```

  The `cat` utility is also useful for "blasting" (printing to screen) one or more text files.

```
$ cat ends
MORNING ROOM IN ALGERNON S FLAT IN HALF MOON STREET
ON THE CONTRARY AUNT AUGUSTA I VE NOW REALISED FOR THE FIRST
   TIME IN MY LIFE THE VITAL IMPORTANCE OF BEING EARNEST
```

- Here is a portable C program which prints one line to `stdout`, and another to `stderr`.

```
$ cat streams.c
#include <stdio.h>

int main(void) {
    fprintf(stdout, "Hi from stdout.\n");
    fprintf(stderr, "FAIL from stderr.\n");
    return 0;
}
```

  After compiling (via `cc` command), it prints two lines, one to `stdout` and another to `stderr`.

```
$ cc -o streams streams.c
$ ./streams > out
FAIL from stderr.
$ cat out
Hi from stdout.
```

  The former is captured and redirected using >; the latter is printed to the terminal screen. This behavior can be modified by adding prefixes to >.

```
$ ./streams 2> err
Hi from stdout.
$ cat err
FAIL from stderr.
```

- The `tee` command provides a way to send output to multiple locations simultaneously. It copies standard input to standard output, so it can be used to simultaneously print and save standard output from an earlier program on the pipeline.

```
$ ./streams | tee out
OH NO from stderr.
Hello from stdout.
$ cat out
Hello from stdout.
```

- If the file pointed to by '>' already exists, it will be overwritten. If, however, you would like to append to (i.e., add to the end of) a file, you can use >>.

```
$ cat ends >> out
$ cat out
Hello from stdout.
```

```
MORNING ROOM IN ALGERNON S FLAT IN HALF MOON STREET
ON THE CONTRARY AUNT AUGUSTA I VE NOW REALISED FOR THE FIRST
   TIME IN MY LIFE THE VITAL IMPORTANCE OF BEING EARNEST
```

Appending is the source of *many* bugs, so use it with care.

- Much like how '>' redirects standard output to a file, '<' sends a file to standard input. This is useful for commands which read from standard input, like `tr`. The `tr` command replaces one character (or character class) with another (or another character class of the same length).

```
$ tr M m < ends
mORNING ROOm IN ALGERNON S FLAT IN HALF mOON STREET
ON THE CONTRARY AUNT AUGUSTA I VE NOW REALISED FOR THE FIRST
   TImE IN mY LIFE THE VITAL ImPORTANCE OF BEING EARNEST
$ tr [:upper:] [:lower:] < ends
morning room in algernon s flat in half moon street
on the contrary aunt augusta i ve now realised for the first
   time in my life the vital importance of being earnest
```

- Instead than redirecting standard output to a file, we can use the '`|`' ("pipe") to redirect the standard output of one program to the standard input of another; this is probably what McIlroy had in mind when he said "[w]rite programs to work together". For example, `grep` ("global regular expression print") prints lines which match a certain string (or regular expression) argument.

```
$ head -100 earnest.txt | grep ALGERNON
MORNING ROOM IN ALGERNON S FLAT IN HALF MOON STREET
```

The command `grep -v` ("inVert") prints all lines that *don't* match the argument string.

```
$ grep -v E earnest.txt | head
OH
THANK YOU SIR
WHAT BRINGS YOU UP TO TOWN
MAY I ASK WHY
GIRLS DON T THINK IT RIGHT
THAT IS ABSURD
IT S ON YOUR CARDS
NOW GO ON
NOW GO ON
DON T TRY IT
```

Note that `grep` can either read data from standard input (as in the former example), or from a file argument (as in the latter).

2.5 Text munging

- To perform a regular-expression substitution, use `perl -p -e` ("evaluate and print").

```
perl -p -e 's/\s+/\n/g' < earnest.txt > words
```

Some people suggest using `sed` for this sort of thing, but I think Perl does this sort of thing much better.

- The `sort` command sorts a file (lexicographically).

```
$ sort words | tail
YOUR
YOUR
YOUR
YOURS
YOURS
YOURS
YOURS
YOURSELF
YOURSELF
YOURSELF
```

- The `uniq` utility removes duplicate lines, *assuming the file has already been sorted*.

```
$ sort words | uniq | tail
YES
YESTERDAY
YET
YEW
YOU
YOUNG
YOUNGER
YOUR
YOURS
YOURSELF
```

The command `uniq -c` ("count") also counts the number of repeated lines. This can be combined with `sort -n` ("numeric sort") to get a list of words sorted by increasing frequency.

```
$ sort words | uniq -c | sort -n
 268 IT
 272 IN
 324 THAT
 371 A
 377 OF
 400 IS
 535 TO
 540 YOU
 543 THE
 823 I
```

- The `wc` command counts lines (`-l`), words (`-w`), and characters (`-c`).

```
$ wc earnest.txt
    1688   17897   91184 earnest.txt
```

  For more of this sort of thing, see K. Church, *UNIX For Poets* (link on the course website).

- The `cut` and `paste` commands are useful—but limited—tools for working with the some forms of columnar data, like CSV ("comma-separated values") files. The `cut` command prints selected columns of a file. The argument to the `-d` flag specifies the delimiter (which must be one byte wide—i.e., a single ASCII character; the default delimiter is the tab character) and the argument to the `-f` flag specifies columns by number (using one-based indexing).

```
$ cat prices
Barista,2.50,3.00
Courier Cafe,2.00,2.00
Floyd's,1.50,2.00
Glyph,2.25,2.25
Starbucks,1.75,2.25
Stumptown,2.50,2.50
Southeast Grind,1.75,1.75
$ cut -d , -f 2 prices
2.50
2.00
1.50
2.25
1.75
2.50
1.75
```

  The `-f` also admits lists of column indices (e.g., `-f 1,3`) or ranges (e.g., `-f 2-3`).

```
$ cut -d , -f 1,3 prices
Barista,3.00
Courier Cafe,2.00
Floyd's,2.00
Glyph,2.25
Starbucks,2.25
Stumptown,2.50
Southeast Grind,1.75
$ cut -d , -f 2-3 prices
2.50,3.00
2.00,2.00
1.50,2.00
2.25,2.25
1.75,2.25
2.50,2.50
1.75,1.75
```

- The `paste` command can be used to put columnar data in separate files back together; here the argument `-d` flag specifies delimiter to be added.

```
$ cut -d , -f 1 prices > cafes
$ cut -d , -f 3 prices > doubleshots
$ paste -d , cafes doubleshots
Barista,3.00
Courier Cafe,2.00
Floyd's,2.00
Glyph,2.25
Starbucks,2.25
Stumptown,2.50
Southeast Grind,1.75
```

- One major limitation to both of these tools is that they break when fields contain their own delimiter; had there been a cafe called "Coffee, The Final Frontier", `cut` and `paste` would have choked. We will discuss more sophisticated tools for columnar data later in the course.

2.6 `find` & `xargs`

- Earlier (§2.3), we saw how wildcards, character ranges, and the like could be used to pick out complex subsets of files in arguments. This method has many limitations, however:
  - There is no simple way to perform a case-insensitive match (i.e., match both `prism` and `PRISM`).
  - Every UNIX-like OS imposes an upper limit on the number of characters that can go in any command. If a wildcard expression exceeds this bound, the command will be invalid.
  - Many UNIX commands are not designed to handle an unbounded list of arguments. For instance, `foo bar baz` does not always have the same effect as `foo bar; foo baz`.
- The `find` command provides a solution. It takes one mandatory argument, which specifies a path to search. With no other arguments, this will simply print the name of every file in that directory of subdirectory.

```
$ pwd
/Users/gormanky/Documents/Code/syllabify
$ find .
./README.md
./manual
./manual/Makefile
./manual/syllabify.aux
./manual/syllabify.bbl
./manual/syllabify.bib
./manual/syllabify.log
```

```
./manual/syllabify.pdf
./manual/syllabify.tex
./syllabify.py
...
```

- That's not terribly useful as is. Two additional arguments make it more useful. The first is the `-name` flag, which takes an argument (using wildcard syntax) to match against file names.

  ```
  $ find . -name 'syllab*'
  ./manual/syllabify.aux
  ./manual/syllabify.bbl
  ./manual/syllabify.bib
  ./manual/syllabify.log
  ./manual/syllabify.pdf
  ./manual/syllabify.tex
  ./syllabify.py
  ...
  ```

  There is also a `-iname` flag, which performs a case-insensitive regular expression match.

  You can also specify file-types using the `-type` flag; the `f` argument matches regular files, and the `d` matches directories.

  ```
  $ find . -type d
  ./manual
  ...
  $ find . -type f
  ./eval/results.md
  ./manual/Makefile
  ./manual/syllabify.aux
  ./manual/syllabify.bbl
  ./manual/syllabify.bib
  ./manual/syllabify.log
  ./manual/syllabify.pdf
  ./manual/syllabify.tex
  ./syllabify.py
  ...
  ```

- The `find` command has a flag `-exec` which allows an arbitrary function to be called on each file that is found. For instance, the following would delete all `*.exe` files in the current directory (and any subdirectories).

  ```
  $ find . -name '*.exe' -exec rm {} \;
  ```

  The `{}` is a variable representing the filename; the last bit delimits the end of the internal command.

- There is one additional problem which `find` alone does not address: even if we have a way to execute the `foo` command on many other arguments, chances are that it will process the arguments in serial. The `xargs` command provides a simple way to run something like `-exec` statements in parallel. In one common use case, `find` is used to list file names, and `xargs` reads these names from standard input. For instance, the following would perform the deletion of `*.exe` files, four processes at a time.

  ```
  $ find . -name '*.exe' -print0 | xargs -0 -P 4 -n 1 -I rm {}
  ```

  The `-print0` flag to `find` and the `-0` flag to `xargs` eliminate a loophole to do with filenames containing whitespace characters. The argument to the `-P` flag specifies the number of simultaneous processes to use. By specifying `-n 1`, we request that `rm` be called exactly once per argument. The `-I` flag enables the use of the `{}` variable; without it, the filename(s) is simply added to the end of the command (so that `xargs rm` is shorthand for `xargs -I rm {}`). Note that, unlike `find -exec`, `xargs` does *not* take a delimiter for the end of the internal command.

2.7 Job control

- Commands executed at the command line (usually) initiate one or more *processes*. Each running process has a unique identifier—a *process ID* (PID)—associated with it. Each completed process has an integer error code, which you can inspect to ensure that the process finished normally. By convention, a non-zero error code indicates failure. The error code of the last process to finish in this shell is stored in the environmental variable `$?`.

  ```
  $ cat error.c
  int main(void) {
      return 12;
  }
  $ cc -o error error.c
  $ ./error
  $ echo $?
  12
  ```

- There are several ways to see which processes are running at the moment. The simplest is the interactive `top` command. You may want to specify how `top` is to sort the running processes using an argument to the `-o` flag. Some options include `cpu` (processor utilization), `mem` (memory utilization), and `time` (time elapsed).

- Another option is the (non-interactive) `ps` ("process status") command. Without any arguments, `ps` displays the processes you own which are running in some shell; other flags can be used to reveal additional running processes.

  ```
  $ ps
  34732 ttys000    0:00.09 -bash
  39103 ttys001    0:00.01 -bash
  39848 ttys002    0:02.17 vi L1-2.tex
  39961 ttys006    0:00.00 man ps
  ```

- By default, a process runs in the *foreground*, meaning that nothing else executes in that shell until the process is complete, and that anything you type in the shell is sent to that process. To make a process run in the *background*, simply add an ampersand (`&`) to the end of the command.

```
$ sleep 20 &
[1] 39860
$ sleep 30 &
[2] 39861
...
[1]-  Done                    sleep 20
[2]+  Done                    sleep 30
```

- Once you've backgrounded a process, you can bring it back to the foreground using the `fg` ("foreground") command. By default, `fg` foregrounds the last process, but you can also specify a process by PID.

- You also use the `wait` command to delay further execution until multiple backgrounded processes are complete. By default, `wait` waits for all backgrounded processes by default, but you can specify PIDs to wait on.

```
$ ./job0.sh &
[1] 39921
$ ./job1.sh &
[2] 39922
$ ./job2.sh &
[3] 39923
$ wait
$ echo "All done."
All Done.
```

- Processes can also be terminated from the command line. The `kill` command takes one or more PID argument and attempts to kill the associated process(es).

```
$ sleep 30 &
[1] 97074
$ kill 97074
[1]+  Terminated: 15          sleep 30
```

  The `killall` takes one or more command arguments and attempts to kill any processes running that command.

```
$ python job0.py &
$ python job1.py &
$ python job2.py &
$ killall python
[1]   Terminated: 15          python job0.py
[2]   Terminated: 15          python job1.py
```

```
[3]+  Terminated: 15          python job2.py
```

Both `kill` and `killall` take an optional numeric argument which sends the associated "signal" to the process. For instance, the following command sends signal 2 (`SIGINT`) to all running `python` processes.

```
$ killall -2 python
```

See `man signal` for additional information on UNIX signals.

- If you are working over a remote connection and the connection is lost, the running processes will usually be terminated. The `nohup` ("no hang-up") command prevents this behavior.

```
$ ssh bigbird12.cslu.ohsu.edu
...
$ nohup ./jobs.sh > jobs.log &
[1] 40022
$ exit
...
$ ssh bigbird12.cslu.ohsu.edu
...
$ cat jobs.log
Training...
Testing...
Accuracy = 0.9523.
```

- The `screen` program allows not just for persistent execution, but persistent shell sessions. The `tmux` program is an increasingly popular alternative to `screen`. There are many tutorials about these tools available online.

- If you use your Macintosh for long-running experiments, you may have noticed that it will "sleep" after so long without any interaction. While you could just disable power management features (in System Preferences → Energy Saver), a more ecological solution is to use the `caffeinate` command, which prevents the system from sleeping. The user can either specify the number of seconds to prevent sleep (with an argument to the `-t` flag), or that the system should stay awake until a command is done executing.

```
$ caffeinate -t 3600 &
$ caffeinate ./train.sh
```

2.8 Ownership & permissions

- Every file is *owned* by at least one user (e.g., `gormanky`) as well as to at least one named group of users (`autism`). The `chown` ("change ownership") command changes file ownership.

```
$ ls -l -h
-rw-r--r--  1 gormanky  staff    34B Sep 29 13:31 myscript.py
$ chown rstites:staff myscript.py
```

```
$ ls -l -h
-rw-r--r--  1 rstites  staff    34B Sep 29 13:35 myscript.py
```

- Every file also has a set of *permissions*. These determine whether someone can:
  - `r`: read
  - `w`: write
  - `x`: execute (i.e., run it as a program)

  By convention, you must have execute privileges to navigate to a directory or any of its subdirectories. You must also have read privileges to list a directory's contents, and you must have write privileges to create a new file in it.
- There are three sets of permissions; one for the user, one for the group or groups, and one set of "global" permissions, which apply to all users.
- The `chmod` ("change access modes") command changes permissions. A common use is to mark a script as an executable:

  ```
  $ ls -l -h myscript.py
  -rw-r--r--  1 gormanky  staff    34B Sep 29 13:31 myscript.py
  $ chmod +x myscript.py
  -rwxr-xr-x  1 gormanky  staff    34B Sep 29 13:31 myscript.py
  ```

  Before the `chmod` command was executed, the script could be modified by user `gormanky` and read by any user. After the command was executed, any user could also execute the command (i.e., run `./myscript.py`).

2.9 Text editors

Consider learning at least one command-line text editor. The three most common.

- Nano (`nano`), a clone of `pico`; type Ctrl+x to quit.
- Emacs (`emacs`), a clear example of the evils of MIT style (you can use it to play Tetris); type Ctrl+x, then Ctrl+c, to quit.
- Vim (`vim` or `vi`), a fork of `vi`; type Esc, then `:q`, then return, to quit.

If you're not sure which you'd like to learn, let me suggest Vim. It has been around in some form or another since the Ford administration, and with good reason. To learn more, work through the `vimtutor` program which should be installed on your UNIX-like machine.