EXERCISES IN COMPUTATIONAL ANALYTICS

UNIVERSITY OF SAN FRANCISCO

Master of Science in Analytics

TERENCE PARR

parrt@cs.usfca.edu

http://parrt.cs.usfca.edu

A BONKERS THE CAT PRODUCTION

Contents

Part I

Introduction

Audience and Summary

Welcome to MSAN501, the computational analytics boot camp at the University of San Francisco! This exercise book collects all of the labs you must complete by the end of the boot camp in order to pass. The labs start out as very simple tasks or step-by-step recipes but then accelerate in difficulty, culminating with an interesting text analysis project. You will build all projects in Python (2.7.x).

This course is specifically designed as an introduction to analytics programming for those who are not yet skilled programmers. The course also explores many concepts from math and statistics, but in an empirical fashion rather than symbolically as one would do in a math class. Consequently, this course is also useful to programmers who'd like to strengthen their understanding of numerical methods.

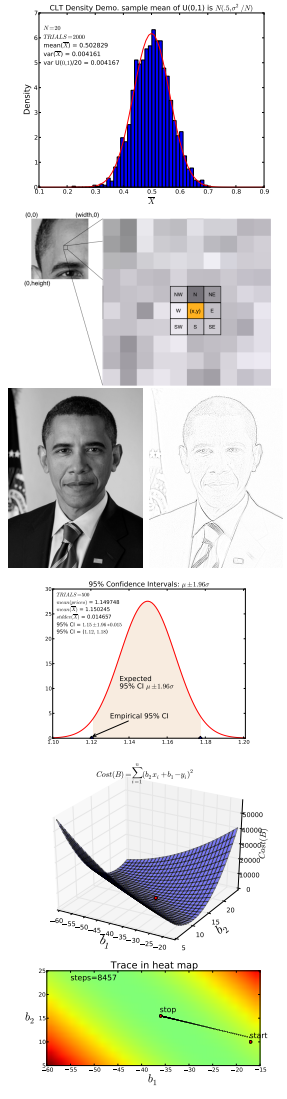
The exercises are grouped into multiple parts. We begin by looking at our fundamental tools, such as the command line and PyCharm. Then we'll study how computers represent data and images, learn simple data structures, and build some visualizations. Next, we'll study how to solve a number of math and statistics problems related to analytics. The empirical statistics part strives to give an intuitive feel for random variables, density functions, the central limit theorem, hypothesis testing, and confidence intervals. It's one thing to learn about their formal definitions, but to get a really solid grasp of these concepts, it really helps to observe statistics in action. All of the techniques we'll use in empirical statistics rely on the ability to generate random values from a particular distribution. We can do it all from a uniform random number generator.

The next set of exercises deal with function optimization. Given a particular function, $f(x)$, optimizing it generally means finding its minimum or maximum, which occur when the derivative goes flat: $f'(x) = 0$. When the function's derivative cannot be derived symbolically, we're left with a general technique called *gradient descent* that searches for minima.

Next, you'll get an introduction to text analysis. We will compute something called *TFIDF* that indicates how well that word distinguishes a document from other documents in a corpus. That score is used broadly in text analytics.

Finally, we'll launch some machines in the cloud and perform some small distributed computing tasks using hadoop and Python.

As you progress through these exercises, you'll learn a great deal about Python and the following libraries: `matplotlib`, `numpy`, `scipy`, and `py.test`. You'll also learn how to run jobs on computers in the cloud.



Grading

You will be using `github.com` to submit your projects (and we'll learn more about `git` shortly). Once you have completed the tasks and they pass any test files I've provided, you will notify me that projects are ready for grading adding a "tag" to your repository. I will review your code and make sure that it passes the tests. Usually I will have some comments and commit changes and comments back to the repository. You must fix all of that and notify me by adding another tag to your repository. Once you have passed, I will add a tag to let you know it has been graded.

This class is pass/fail and all of the projects are pass/fail. You simply have to get past the tests and my code review for each exercise in this book.

Here are the tags that you are to enter when you complete each task:

hist Histograms Using matplotlib

images Image Processing

graphs Graph Adjacency Lists and Matrices

stats All exercises in Empirical statistics

descent Iterative Optimization Via Gradient Descent

regression Predicting Murder Rates With Gradient Descent

tfidf Summarizing Reuters Articles with TFIDF

I will tag the repository with `pass` *X* for your tag *X*. For repeated submissions of *X*, use *X2*, *X3* etc...

Problem solving

Some questions to get you warmed up

Q. You intercept an encrypted message from another student and you would like to decrypt it because you are nosy. Let's assume that the student mapped each letter of the alphabet to a different, unique letter chosen at random. How would you go about decrypting this message? Is it even possible if the mapping is truly random?

Figure 1: From
http://www.sacred-texts.com/eso/sta/sta42.htm

Q. Geophysicists like to explode dynamite and measure the direct and reflection waves using a "pen on rolling paper" device like a seismograph. You find yourself in a remote location examining the signal from such a device and need to compute the area under the curve. The only tools you have with you are spare paper, pencil, sharp razor, ruler, and a sensitive scale. How can you get a reasonably accurate measure of the area under the seismograph curve in between two points?

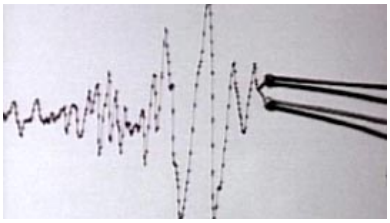
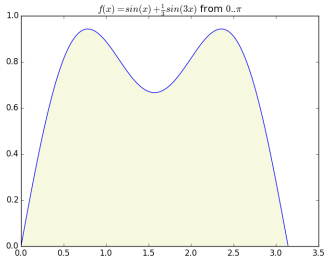


Figure 2: From
http://www.pbs.org/wgbh/nova/education/earth

Q. Imagine there is no symbolic solution to the area under the curve of $f(x) = \sin(x) + \frac{1}{3}\sin(3x)$ from 0 to π. How could you estimate the area? Hint: do you like playing darts?



Q. How can you sort N integers in $[0, M]$ if you can only look at each number once?

Figure 3: From
https://en.wikipedia.org/wiki/Pigeonholing

Part II

Fundamental Tools

Using the PyCharm IDE

Get and install

https://www.jetbrains.com/pycharm/download/

Why an IDE?

As I help students poke around in the pycharm development environment, I often notice that many treat it like a text editor. For example, you copy code out of your Python script and paste it into a console in order to run your programs. Don't do that.

Development environments like the one we're using in this class are the culmination of decades of experimentation into making programmers as productive as possible. Using it as a text editor is silly because you might as well just use a text editor, right? There's no point in installing some massive development environment on your machine. There are plenty of good text editors and you are free to use them. A bit of history will help use the development environment productively.

In the bad old days, programmers literally had to flip switches on the front of the machine to insert a boot program so the machine would actually come up. Then we had punchcards. Then we had line oriented editors like "edit line 34." Yes, my life was that bad in 1980. Tedious but better than physical punch cards. Then we got visual editors where we used cursor keys to move around (enter vi, emacs etc...). We would edit code, exit the editor, and then run our program. A text editor does absolutely nothing for you except let you edit characters in the program.

A development environment **has** a text editor built-in but it knows that you are typing in Python code. Hence you will see that it identifies errors in your code such as syntax errors and often type errors before you even run the program. You don't cut and paste the code into a console somewhere, you just say "run this program." The console is for playing around and experimentation, not testing your program. As the programs get bigger, there's just no way to use the console. Some of the good stuff with an IDE:

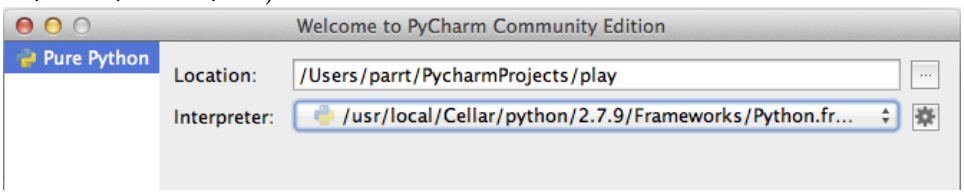
- The development environment can jump to the definition of a function, even if it's in a built-in library.
- You can rename variables and functions using knowledge of Python syntax rather than brain-dead string replace, which could introduce errors. Re-factoring is one of the critical tasks of a programmer.
- You can select code and say *extract method*. You can select an expression and say *introduce variable* etc.
- The development environment knows how to reformat your code.
- When you get a runtime error, you can click on it to go that point in the file that crashed.

Most importantly the development environment has a debugger where you can single step through your program to see variables change and understand what has gone wrong.

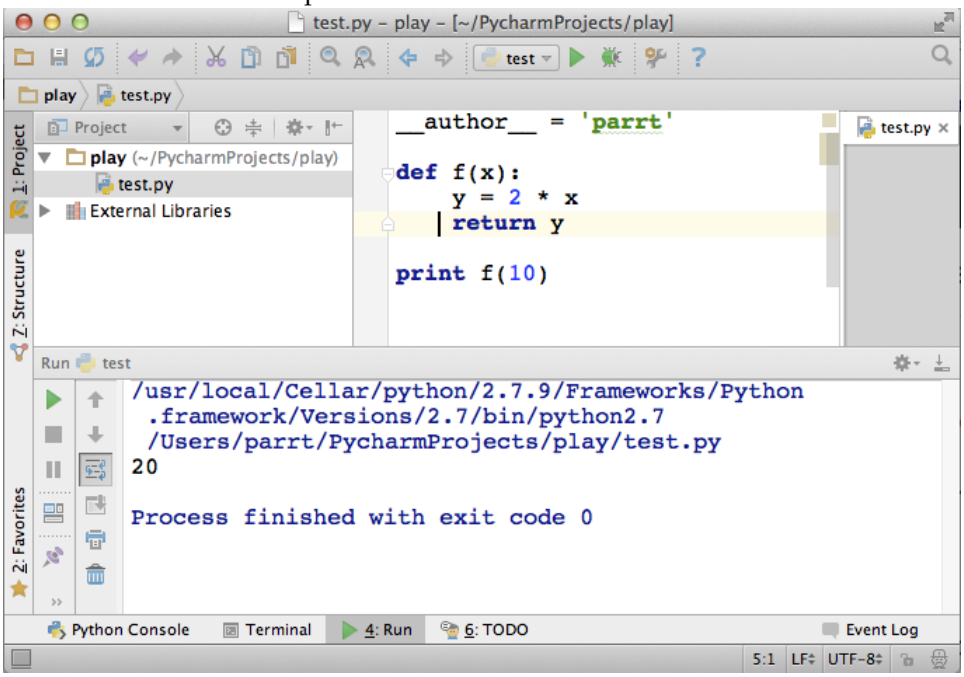
The development environment helps you understand your program as a whole, particularly when the program spans multiple files. As the programs get bigger, you will find the development environment to be very helpful. Part of my job in this class is to give you good programming habits to prepare you for the coming year and your future jobs.

Give it a test drive

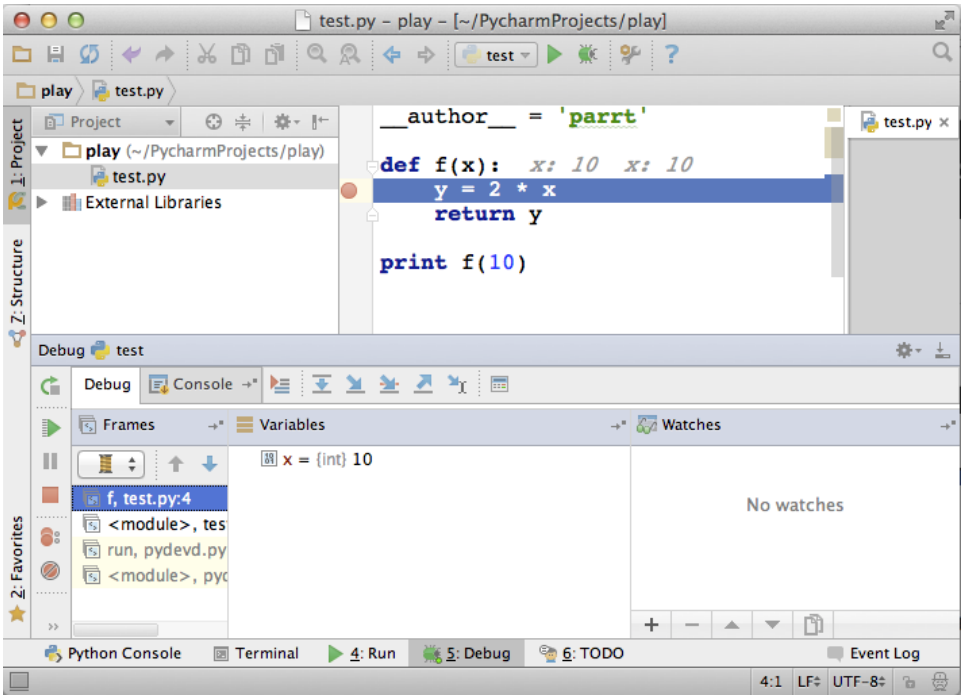
Create a sample project, making sure that you are using the latest interpreter from homebrew installation (`/usr/local/Cellar/...`):



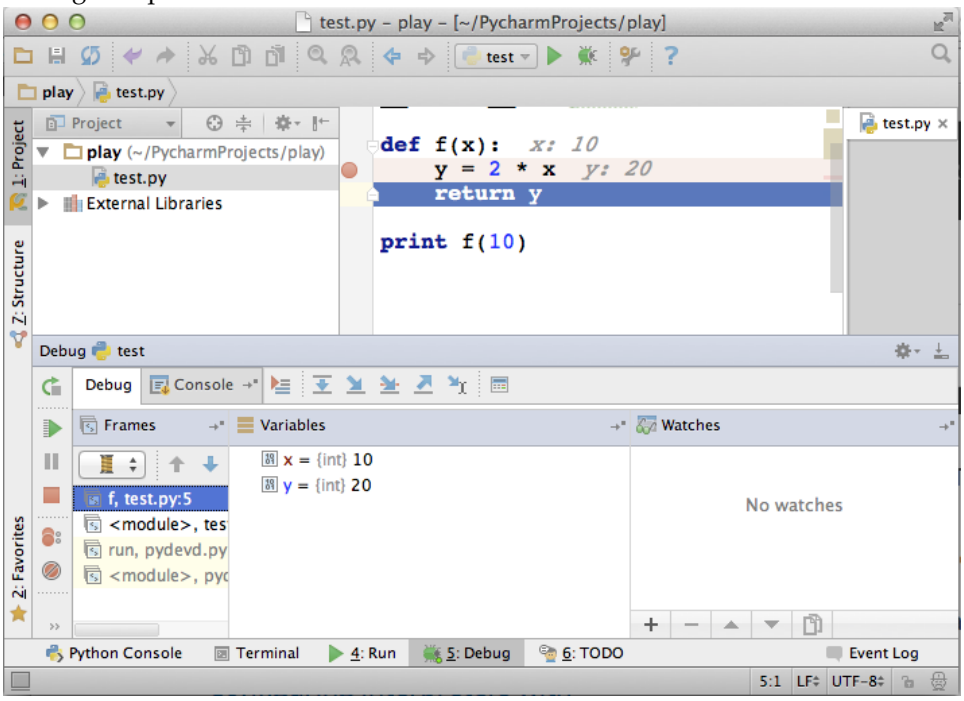
Create a new file with a simple function and run it:



Start up the debugger by clicking the bug icon in the toolbar:



Now single step to the next statement:



Git on it

For our purposes in MSAN, we're going to ignore most of the nontrivial capabilities that programmers use routinely, such as branching and merging. Git is extremely complicated and would not be my first choice if it weren't for the excellent `github.com`.

Introduction to revision control

Revision control is a mechanism to track all changes to a set of files, which we typically associate with the project. The file status call a *repository* and at any given time, my computer has lots and lots of these repositories.

A `git` repository instance is just a directory on your disk that also happens to have a `.git` (hidden) directory, which is effectively a complete database of everything that's happened to the repository since it was created with `git init` (or you `clone`'d it from somewhere).

If you want to throw out the repository, just remove the entire subtree from your disk. There is no central server to notify. Every repository instance is a complete copy so you could have, for example, 10 versions of the repository cloned from an original sitting on the same disk.

After you create a repository, you can create all sorts of files in the local directories managed by git, but git ignores them until you `add` them. When you add files or modify files already known to git, they are in the so-called staging area (this used to be called the index). You can have whatever other files you want laying around, such as development environment preference files. Git will simply ignore them unless you `add` them. This is different than other revision control systems that insist upon knowing about and managing everything under a particular subtree. I like that feature.

Does a solo programmer need revision control?

If you are working solo, from a single machine, and you have a regular backup mechanism in your development environment or from the operating system like Time Machine (OS X), you can get away without a formal revision system.

There are lots of important operations that can be faked without a revision system. It's a good idea to keep track of versions of the software that work or other milestones. In the old days, people would

make a copy of their project directory corresponding to important milestones like "Added feature X and it seems to work." You can do comparisons using a diff tool in between directories.

Whether your IDE does it or a revision control system does it, I find it very important to look back at recent changes to see what changes have introduced a bug. Or I decide to abandon a small piece of what's going on and flip a file back to an old version.

A good example of use of a repository is the repository for this course:

https://github.com/parrt/msan501

It contains all the changes that I've made since I started teaching this course.

These days, revision control systems are meant to be used among multiple computers and multiple developers, but they are still useful even on a single machine.

Solo programmer, sharing across machines

In order to work on that software from your home machine and a laptop for example, you have to make copies. That introduces the possibility that you will overwrite the good version of your software. Or, you will forget that you had made changes on your laptop but have now made a bunch of changes on your desktop. Resolving things can be tricky and error-prone.

As a side benefit, pushing your repository to a remote server gives you a backup automatically.

Multiple programmers

When you add another person to the project, people end up mailing code around but it's difficult to perform a merge. My experience watching students do this reveals that two versions of the software always appear. Both students shout that their version is better and that the other version should be abandoned.

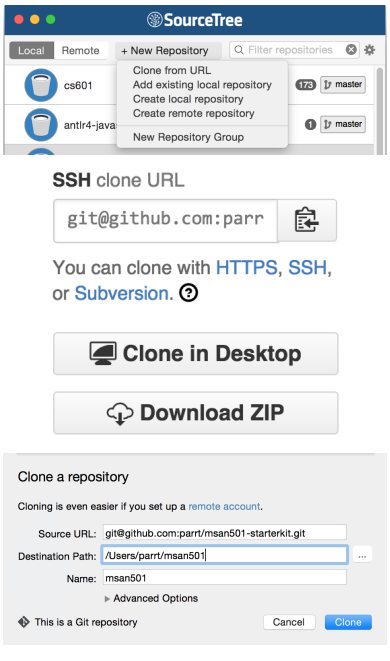
In my experience, no matter how you try to fake multiple states of the source code and share, merging changes to work on the same code base is a nightmare.

Once in a while I go back and I look at the history of changes. Sometimes I want to know who screwed this up or I want to see the sequence of changes that I made or that were made by somebody else.

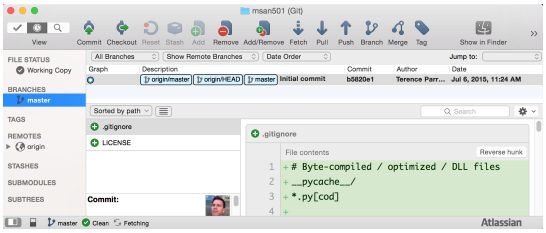
Every single commercial developer I know uses revision control at work. Every company you will encounter uses it. For that reason alone, you need to learn revision control to be functional in a commercial setting.

Cloning an existing repository

Fire up SourceTree and use the file menu `New / Clone` option. You can clone that starter kit into any directory you like, but you should probably name it `msan501` or something similar. Note that I'm still in the process of setting up student repositories as I'm writing this prior to receiving a list of github ids. The github URL you will clone will not be similar to tt msan-501-starterkit but instead *YourGihubUserid-msan501*.



You'll then see the status of the repository; click on the `master` branch in the left gutter:



Naturally, this is all much more explicit and quicker from the commandline:

```
$ cd ~
$ git clone git@github.com:parrt/msan501-starterkit.git msan501
```

PyCharm + git = no problem

Most of the time, you can use PyCharm to avoid all of the unpleasantness with the specialized git commands. For example, opening PyCharm on your `msan501` directory after you clone it from github should automatically detect that the directory is a repository. But, you can check this in the preferences. Then, when you create a new file, say, `t.py`, PyCharm will automatically ask if you want to add it to the repository. The two snapshots in the gutter show you what it looks like. You can click the checkbox so that it always just adds files. If you want to manually add them later it's okay. You can do so under the VCS (version control system) menu.

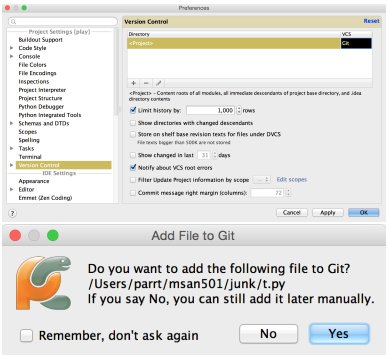


Figure 4: Git config in PyCharm

To figure out what changes you have made to the existing repository, click on the Changes tab at the bottom of PyCharm. You will see that t.py is a change (addition) to the repository.

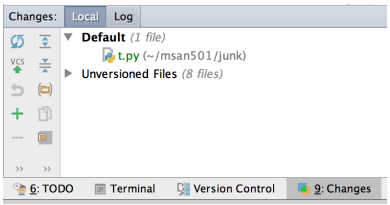


Figure 5: View changes to repo

Use menu option VCS > Commit Changes to commit your changes to the local repository. As part of the commit, you can push to the `origin`, which will bring up the Git Push dialog box.

Tagging. As part of delivering your projects, you will signal to me that a particular task is ready for review by tagging. The specific tag name will identify which project I should review and then I will add a tag to indicate I have graded it. Going back to SourceTree, we can see the recent commit of t.py. Right clicking on that line in the GUI, brings up a menu where we can add a tag. The tag will be added to the local repository and pushed to the origin if you click the appropriate checkbox. I will not be able to see it until he goes back to the origin.

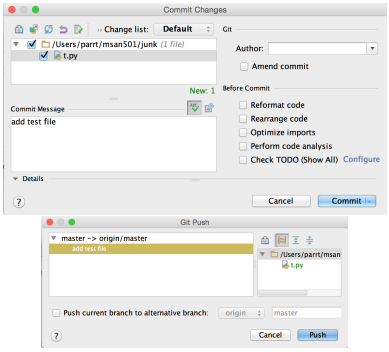


Figure 6: Commit and push

Git from the command line

Adding to repo. To add files, just create them and do a `commit`:

```
$ cd ~/msan501
$ mkdir junk
$ cd junk
... create t.py ...
$ git add t.py
$ git commit -a -m 'initial add'
```

Make changes. Then you can make changes and do another commit. Make sure use the `-a` command. By the way, deleting a file is also considered a change but you can also use `git rm` *filename*.

Checking differences with repo. If you make a change and want to know how it's different from the current repository version, just use diff:

```
$ ... tweak t.py ...
$ git diff t.py
...
```

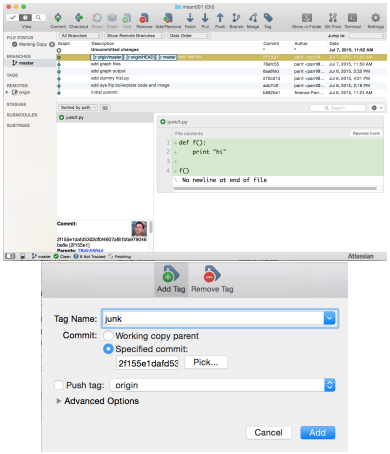


Figure 7: SourceTree view, tagging

Reverting. If you screw up and want to toss out everything from the last commit, do a reset and make sure you use the hard option:

```
$ ... tweak whatever you want ...
$ git reset --hard HEAD
```

which throws out all changes since the last commit. If all you want to do is revert uncommitted changes to a single file, you can run this:

```
$ git checkout -- filename
```

I think they call that funny dash-dash option "sparse mode." See? Git is the assembly code of revision systems. blech.

Correcting commit message. One of the other things I often have to do is to fix the commit message that I just wrote in a commit command.

```
$ git commit --amend -m "I really wanted to say this instead"
```

Adding file you forgot to commit. If you forgot to add one of the files and you wanted in a previous commit, you can also use amend. Just add the file and use amend:

```
$ git add t2.py
$ git commit --amend --no-edit
```

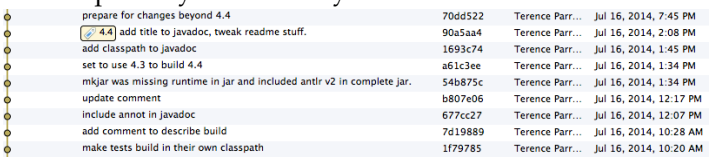
Checking working dir and staging area vs repo. Finally, if you want to figure out what changes you have made such as adding, deleting, or editing files, you can run:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    t3.py

nothing added to commit but untracked files present (use "git add" to track)
```

Don't fear `commit`*ment*

Every time you *commit* a change (file edit, file add, file delete, ...) to the repository, the revision control system tracks a patch called the *diff* that indicates essentially how to edit the first file version in order to arrive at the current file version. Storing just the differences is very space efficient and lets the revision control system apply the same set of changes to a file on a different computer to keep them both in sync.

Having a complete list of changes is extremely useful. For example, here is a chunk taken out of the middle of my commits on the ANTLR repository as shown by SourceTree:



That should look very much like Time Machine on OS X to you. You can go back and look at changes made to the repository for any commit.

You will also notice that I have tagged a particular commit as 4.4 with the tag command. This makes it easy for me to flip the repository back to a specific commit with a name rather than one of those funky commit checksums.

You should commit only logical chunks like feature additions, bug fixes, or comment updates across the project, etc.

Commit messages are important. Do not use meaningless messages, as I see students sometimes do:

```
Add t.py
Alter t.py
Change t.py
...
```

I have even seen git commit messages: `one`, `two`, `three`. Nothing spells laziness or academic dishonesty than that.

In rare cases, when I'm working alone, I sometimes use a private repository as a means of sharing files across multiple computers like dropbox. In this case, my commits are just to take a snapshot to pull it down on another machine. The commit message doesn't matter (though I might still look back through the changes at some point). When doing this for a real project, it's best to use stash per the next section.

With a remote server like github

When you're working by yourself (and without branches), a remote server acts like a central server that you can push and pull from. For example, I push from my work machine and pull to my home machine or my laptop. And then reverse the process with changes I make at home over the weekend.

For example, once I have commit all of my changes that work and I'm ready to go home, I push to the origin:

```
$ git push origin master
```

From home, I do:

```
$ git pull origin master
```

The origin is the alias for the original server we cloned from and master is our master branch, which we can ignore until we look at branches.

To look at the remote system alias(es), we use:

```
$ git remote -v
origin git@github.com:parrt/msan501.git (fetch)
origin git@github.com:parrt/msan501.git (push)
```

Linux command line

UNIX shell is an interactive domain specific language used to control and monitor the UNIX operating system, which includes processes, devices, ram, cpus, disks etc. It is also a programming language, though we'll use it mostly to do scripting: lists of commands. If you have to use a Windows machine, the shell is useless so try to install a UNIX shell.

You need to get comfortable on the UNIX command line because many companies use Linux on their servers, which in my opinion, is best used from the command line for ultimate control over the server or cluster. For example, in the next lab we will launch Hadoop jobs from the shell. Facility with the show marks you as a more sophisticated programmer.

Everything is a stream

The first thing you need to know is that UNIX is based upon the idea of a stream. Everything is a stream, or appears to be. Device drivers look like streams, terminals look like streams, processes communicate via streams, etc... The input and output of a program are streams that you can redirect into a device, a file, or another program.

Here is an example device, the null device, that lets you throw output away. For example, you might want to run a program but ignore the output.

```
$ ls > /dev/null # ignore output of ls
```

where "# ignore output of ls" is a comment.

Most of the commands covered in this lecture process stdin and send results to stdout. In this manner, you can incrementally process a data stream by hooking the output of one tool to the input of another via a pipe. For example, the following piped sequence prints the number of files in the current directory modified in August.

```
$ ls -l | grep Aug | wc -l
```

Imagine how long it would take you to write the equivalent C or Java program. You can become an extremely productive UNIX programmer if you learn to combine the simple command-line tools.

The basics

UNIX disk structure: http://www.thegeekstuff.com/2010/09/linux-file-system-structure/

`~parrt` is my home directory, `/home/parrt`, as is `~`.

```
$ls /
Applications
Library
Network
System
Users
```

```
Volumes
bin
cores
dev
etc
home
mach_kernel
net
opt
private
sbin
tmp
usr
var
```

Like when were typing in the Python shell, each command is terminated by newline. The first thing we type is the command followed by parameters (separated by whitespace):

```
$ foo arg1 arg2 ... argN
```

That is why whitespace in filenames sucks:

```
$ ls house\textbackslash of\textbackslash pancakes
```

But we can use this:

```
$ ls 'house of pancakes'
```

The commands can be built into the shell or they can be programs that we write and invoke. For example, here's how you ask which program is being executed when you type a command:

```
$which ls python
/bin/ls
/usr/local/bin/python
```

The Python interpreter is a program installed on our disk and when we say `python` at the shell, it finds the program using an ordered list of directories in `PATH` environment variable and executes it.

Next, we pass information around using streams and we can shunt that data into a file or pull data from a file using special operators. You can pretend these are like operators in a programming language like addition and multiplication. Each program has standard input, standard output, and standard error; three streams.

We can set the standard input of a process using > character:

```
$ls / > /tmp/foo
```

Here is how to type something directly into a text file:

```
$ cat > /tmp/foo
the first line of the file
the second line of the file
^D
$
```

The `^D` means control-D, which means end of file. `cat` is reading from standard input and writing to the file. The way it knows we are done is when we signal in the file with control-D.

We can set the standard input of a process to the contents of a file and redirect the output of a process to a file.

```
$wc < /tmp/foo
      19      19     118
```

or

```
$wc /tmp/foo
      19      19     118 /tmp/foo
```

We can connect to the output of one program to the input of another using pipes: `|`.

```
$ls / | wc # count files are in the root directory
      19      19     118
```

Here is a simple pipe (show first 5 lines of the text we stored in foo):

```
$cat /tmp/foo | head -5
Applications
Library
Network
System
Users
```

So, some programs take filenames on the command line and some expect standard input. For example, the `tr` translation command expects standard input and writes to standard output

```
$ls / | tr -d e # delete all 'e' char from output
Applications
Library
Ntwork
Systm
Usrs
Volums
bin
cors
dv
tc
hom
mach_krnl
nt
opt
privat
sbin
tmp
usr
var
```

Misc

man, help, apropos

ls, cd, pushd, popd

cp, scp

cat, more

head, tail

The most useful incantation of tail prints the last few lines of a file and then waits, printing new lines as they are appended to the file. This is great for watching a log file:

```
$ tail -f /var/log/mail.log
```

wc

Searching streams

One of the most useful tools available on UNIX and the one you may use the most is grep. This tool matches regular expressions (which includes simple words) and prints matching lines to stdout.

The simplest incantation looks for a particular character sequence in a set of files. Here is an example that looks for any reference to System in the java files in the current directory.

```
$ grep System *.java
```

You may find the dot '.' regular expression useful. It matches any single character but is typically combined with the star, which matches zero or more of the preceding item. Be careful to enclose the expression in single quotes so the command-line expansion doesn't modify the argument. The following example, looks for references to any a forum page in a server log file:

```
$ grep '/forum/.*' /home/public/cs601/unix/access.log
```

or equivalently:

```
$ cat /home/public/cs601/unix/access.log | grep '/forum/.*'
```

The second form is useful when you want to process a collection of files as a single stream as in:

```
$ cat /home/public/cs601/unix/access*.log | grep '/forum/.*'
```

If you need to look for a string at the beginning of a line, use caret '^':

```
$ grep '^195.77.105.200' /home/public/cs601/unix/access*.log
```

This finds all lines in all access logs that begin with IP address 195.77.105.200.

If you would like to invert the pattern matching to find lines that do not match a pattern, use -v. Here is an example that finds references to non image GETs in a log file:

```
$ cat /home/public/cs601/unix/access.log | grep -v '/images'
```

Now imagine that you have an http log file and you would like to filter out page requests made by nonhuman spiders. If you have a file called spider.IPs, you can find all nonspider page views via:

```
$ cat /home/public/cs601/unix/access.log | grep -v -f /tmp/spider.IPs
```

Finally, to ignore the case of the input stream, use -i.

Basics of file processing

cut, paste

```
$cat ../data/coffee
3 parrt
2 jcoker
8 tombu
```

cut grabs one or more fields according to a delimiter like strip in Python. It's also like SQL `select f1, f2 from file`.

```
$cut -d ' ' -f 1 ../data/coffee > /tmp/1
cut -d ' ' -f 2 ../data/coffee > /tmp/2

$cat /tmp/1
3
2
8

$cat /tmp/2
parrt
jcoker
tombu
```

paste combines files as if they were columns:

```
$paste /tmp/1 /tmp/2
3       parrt
2       jcoker
8       tombu

$paste -d ', ' /tmp/1 /tmp/2
3,parrt
2,jcoker
8,tombu
```

Get first and third column from names file

```
$ cut -d ' ' -f 1,3 names
```

`join` is like an INNER JOIN in SQL. (zip() in python) first column must be sorted.

```
$cat ../data/phones
parrt 5707
tombu 5001
jcoker 5099

$cat ../data/salary
parrt 50$
tombu  51$
jcoker 99$

$join ../data/phones ../data/salary
parrt 5707 50$
tombu 5001 51$
jcoker 5099 99$
```

Here is how I email around all the coupons for Amazon Web services without having to do it manually:

```
$ paste students aws-coupons
jim@usfca.edu X
kay@usfca.edu Y
sriram@usfca.edu Z
...
```

and here is a little Python script to process those lines:

```
import os
import sys
for line in sys.stdin.readlines():
```

```
p = line.split('\t')
p = (p[0].strip(), p[1].strip())
print "echo '' | mail -s 'AWS coupon "+p[1]+"' "+p[0]
os.system("echo '' | mail -s 'AWS coupon "+p[1]+"' "+p[0])
```

and here's how you run it:

```
$ paste students aws-coupons | python email_coupons.py
```

Processing log files

```
$ cut -d ' ' -f 1 access.log | sort | uniq -c | sort -r -n|head
```

Get unique list of IPs.
Find out who is hitting your site by getting histogram.
How many hits to the images directory?
How many total hits to the website?
Histogram of URLs.

Python programs

```
$python code/linux/printargs.py hi mom
args: hi mom
```

That Python code:

```
import sys
print "args:", sys.argv[1], sys.argv[2]
```

We can use those arguments as filenames to open or we can read from standard input:

```
import sys
print sys.stdin.readlines()
```

Print coffee data out

```
$python code/linux/mycat.py < ../data/coffee
['3 parrt\n', '2 jcoker\n', '8 tombu\n']
```

Histograms Using `matplotlib`

Discussion

The goal of this lab is to teach you the basics of using matplotlib to display probability mass functions, otherwise known as histograms. In this lab we will use the uniform distribution. Use filename `hist.py`.

Steps

1. import the proper libraries

```
import matplotlib.pyplot as plt
import numpy as np
```

2. Get a sample of uniform random variables in $U(0,1)$

```
N = 1000
X = [np.random.uniform(0,1) for i in range(N)] # U(0,1)
# or np.random.uniform(0,1,N)
```

3. Display a histogram using matplotlib (in a separate window)

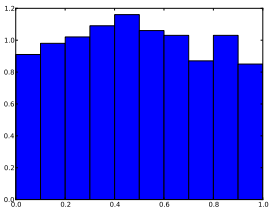
```
plt.hist(X, normed=1)
plt.show()
```

4. Run it.

5. Now, save the image as `unif-0-1-density.pdf` to the same directory by inserting a save command in between the histogram and the show method:

```
plt.hist(X, normed=1)
plt.savefig('unif-0-1-density.pdf', format="pdf")
plt.show()
```

6. Run it. Your hist.pdf file should look like



7. Graphs should always have the axes labeled. Let's do that as well as add a title and set the range of the graph. Put this code right before the `savefig()`.

```
plt.title("U(0,1) Density Demo")
plt.xlabel("X", fontsize=16)
plt.ylabel("Density", fontsize=16)
plt.axis([0, 1, 0, 2])
```

8. Run it.

9. It's also common to add some annotations inside the graph to explain more about what we are seeing. First, we need to get access to the figure itself and then has to figure about its axes. (We need this in order to specify coordinates within the graph.) Put the following code before the `hist()` call.

```
fig = plt.figure()        # get a handle on the figure object itself
ax = fig.add_subplot(111) # weird stuff to get the Axes object within figure
```

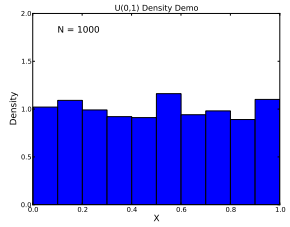
Then, before the `savefig()`, add the following to display some text above the histogram within the graph. The coordinates are from 0..1 where 0 is the left/bottom edge and 1 is the right/top edge.

```
# put N=... at top left
plt.text(.1,.9, 'N = %d' % N, fontsize=16, transform = ax.transAxes)
```

10. Also, let's change the file name slightly so we can keep our original graph plus our fancy one:

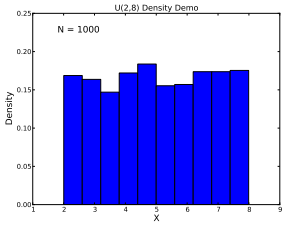
```
plt.savefig('unif-0-1-density-fancy.pdf', format="pdf")
```

11. Run it.



To understand distributions, it's a great idea to start messing around with the parameters of the density or mass function.

12. Change $U(0,1)$ to $U(2,8)$ and save the results in `unif-2-8-density-fancy.pdf`. You will have to alter the axis() to use different ranges. (Or let the plotting software do the work for you and get rid of the axis() call.) Run it. You should see something like the following.



Deliverables. Make sure that `hist/hist.py`, `hist/unif-0-1-density.pdf`, `hist/unif-0-1-density-fancy.pdf`, `hist/unif-2-8-density-fancy.pdf` are correctly committed to your repository and pushed to github. Tag when completed with `hist`.

Part III

Python Programming and Data Structures

Representing Data

Digitizing our world

Everything in the computer is stored as a number. This includes numbers of course as well as letters, audio files, movie files, etc.

Unary

"*One if by Land Two if by Sea*" –Paul Revere

A single symbol or digit: 1. It's the way we count with tick marks in groups of 5.

Unary digits	Value
	0
1	1
1 1	2

Binary

Numbers are encoded in the machine using binary, 0 or 1 which corresponds to usually 0 and +5 Volts in the hardware. The fundamental element within a computer is a switch that can be either on or off just like a lightbulb. If I have one lightbulb, I can have two states: on or off, 0 or 1, lo or hi, whatever you want to call it. If I have two lightbulbs, how many states can I have? 4

Binary digits	Value
0 0	0
0 1	1
1 0	2
1 1	3

What about three? $2^3 = 8$:

Binary digits	Value	Value
0 0 0	0	1
0 0 1	1	2
0 1 0	2	3
0 1 1	3	4
1 0 0	4	5
1 0 1	5	6
1 1 0	6	7
1 1 1	7	8

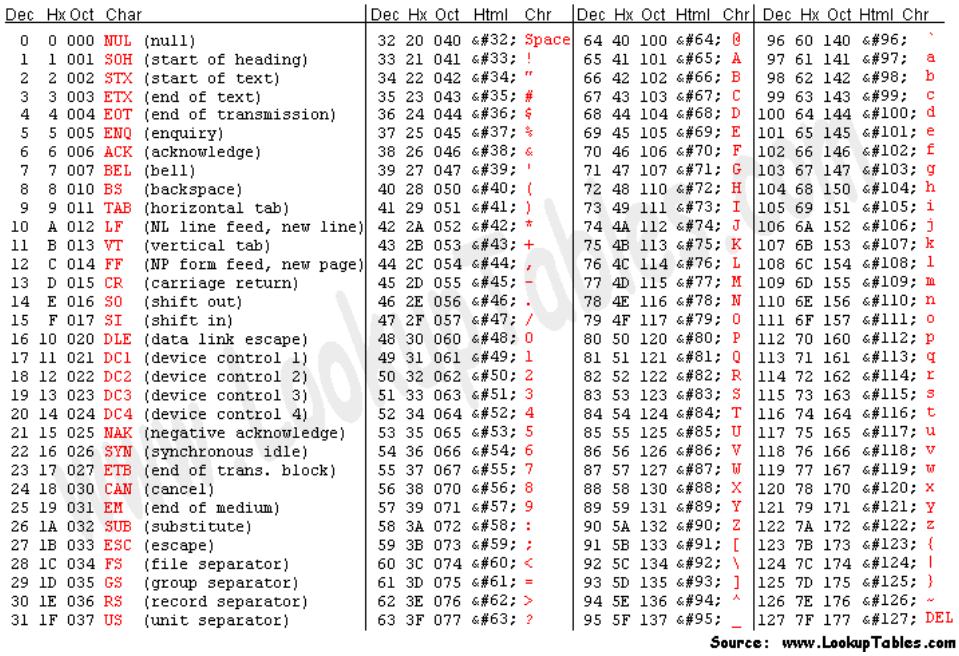
These are two and three bit numbers, which indicates their maximum value.

A **byte** is 8 bits. $2^8 = 256$ states or values 0..255. A **word** is typically the size of the microprocessors primary register. These days that will be 64 bits.

Binary numbers get long pretty quickly: `1010011010` binary is 666 decimal.

Characters

Everything is a number, so how to represent letters? (For now we we'll stick with the American character set). We assign a unique number to each letter:



Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

You can think of this is kind of an encryption. Instead of saying "Hi" you might say "73 105."

There are multiple standards but the clear winner is ASCII. In the old days there was also EBCDIC.

As we will see shortly, a phrase or sentence or word is just a sequence of characters hence a sequence of numbers stored in the machine.

```
$ cat sentence.txt
As we will see shortly, a phrase or sentence or word is just a sequence
of characters hence a sequence of numbers stored in the machine.
$ od -b sentence.txt
0000000   101 163 040 167 145 040 167 151 154 154 040 163 145 145 040 163
0000020   150 157 162 164 154 171 054 040 141 040 160 150 162 141 163 145
0000040   040 157 162 040 163 145 156 164 145 156 143 145 040 157 162 040
0000060   167 157 162 144 040 151 163 040 152 165 163 164 040 141 040 163
0000100   145 161 165 145 156 143 145 012 157 146 040 143 150 141 162 141
0000120   143 164 145 162 163 040 150 145 156 143 145 040 141 040 163 145
0000140   161 165 145 156 143 145 040 157 146 040 156 165 155 142 145 162
0000160   163 040 163 164 157 162 145 144 040 151 156 040 164 150 145 040
0000200   155 141 143 150 151 156 145 056 012
0000211
$ od -c -b sentence.txt
0000000   101 163 040 167 145 040 167 151 154 154 040 163 145 145 040 163
           A   s       w   e       w   i   l   l       s   e   e       s
0000020   150 157 162 164 154 171 054 040 141 040 160 150 162 141 163 145
           h   o   r   t   l   y   ,       a       p   h   r   a   s   e
0000040   040 157 162 040 163 145 156 164 145 156 143 145 040 157 162 040
               o   r       s   e   n   t   e   n   c   e       o   r
```

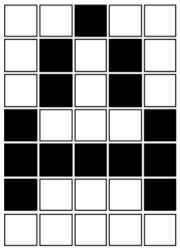
```
0000060   167 157 162 144 040 151 163 040 152 165 163 164 040 141 040 163
           w   o   r   d       i   s       j   u   s   t       a       s
0000100   145 161 165 145 156 143 145 012 157 146 040 143 150 141 162 141
           e   q   u   e   n   c   e  \n   o   f       c   h   a   r   a
0000120   143 164 145 162 163 040 150 145 156 143 145 040 141 040 163 145
           c   t   e   r   s       h   e   n   c   e       a       s   e
0000140   161 165 145 156 143 145 040 157 146 040 156 165 155 142 145 162
           q   u   e   n   c   e       o   f       n   u   m   b   e   r
0000160   163 040 163 164 157 162 145 144 040 151 156 040 164 150 145 040
           s       s   t   o   r   e   d       i   n       t   h   e
0000200   155 141 143 150 151 156 145 056 012
           m   a   c   h   i   n   e   .  \n
0000211
```

Images

Images are stored as numbers also. Each pixel on the screen is typically represented by three numbers, it's (red, green, blue) RGB values. For example:

white: 255 255 255 (yes they are one byte each)
black: 0 0 0
blue: 0 0 255 (blue is saturated)
Yellow is a mix of red and green: 255 255 0
(Play with omnigraffle or other application that will show color)

Or, we can use a single bit to represent a black-and-white image where zero means off and one means on:



The bit sequence is:

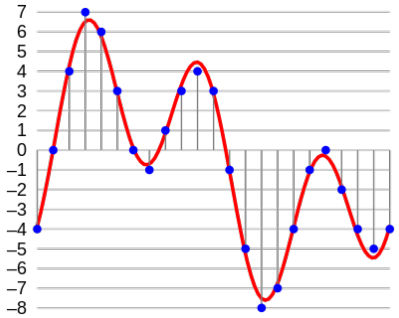
```
00100 01010 01010 10001 11111 10001 00000
```

If you stack vertically, you can see the image sort of:

```
00100
01010
01010
10001
11111
10001
00000
```

Audio

Audio that you listen to are also represented as just a sequence of numbers where each number represents the amplitude of the signal at discrete locations and time. From Wikipedia page on Digital audio:



Entropy

It could be a good time to mention the term entropy, which is a measure of the chaos or disorder of a model or system. In the real world, systems always increase entropy. For example, the state of my kitchen approaches maximum entropy two weeks after the maid has cleaned up. You will see entropy again when you look at algorithms to construct random forests. Entropy in information theory describes how much information is in a signal.

Q. Does it take more bits or fewer bits to store random noise compared to, for example, a pure tone at a particular frequency?

If you need to store a random variable that can take n values with equal probability, you need $log(n)$ bits to represent that variable/number. On the other hand, if we know for sure that the random variable is always, say, 9345 then it only takes a single bit to represent that random variable (9345 is present or not).

Python's atomic elements

There are a few basic or atomic elements in Python and each kind of element knows not only its value but also its type. Is very important to learn the difference between value and type.

Integers

These are just numbers that were used for counting; whole numbers like -9, 0, 3, 1023023823. You can make these things as big as you like:

```
>>> type(32324132123412341234812309132 4)
<type 'long'>
```

But if they are smaller they go into a different type. `type` is just another function call that asks the type of something.

```
>>> type(3232)
<type 'int'>
```

Real numbers

Real numbers, not whole numbers, are of finite precision but can hold some very large and very small numbers.

```
>>> 3.14159
3.14159
>>> 0.000000000001
1e-12
>>> 23e100
2.3e+101
```

The e stuff is the scientific notation and represents the exponent. We call these floating-point numbers.

The Python tutorial on floating-point numbers is something you should look at to learn more about floating-point numbers. The fact that they have finite precision, so-called "double precision," means you can get some odd results:

```
>>> 0.1 + 0.2
0.30000000000000004
```

This is because 0.1 is actually represented as 0.0001100110011..., a repeating fraction. It has no nice representation in binary fractions, but numbers like 0.125 do: 0.001 in binary with value $\frac{0}{2^1} + \frac{0}{2^2} + \frac{1}{2^3}$.

If you need floating-point numbers that trade precision for efficiency, use the decimal module:

```
>>> from decimal import Decimal
>>> Decimal(1)/Decimal(10) + Decimal(2)/Decimal(10)
Decimal('0.3')
```

One last thing on Floating-point numbers. Be aware that subtraction can destroy precision. It is considered an ill conditioned operation because subtracting two numbers that are almost equal can give you very imprecise answers.

Boolean values

A Boolean value is either true or false, but Python also allows a number of other things to represent true and false such as 1 can be true and 0 can be false.

```
>>> True
True
>>> False
False
>>> bool(1)
True
>>> bool(0)
False
>>> bool(36)
True
>>> bool("hi")
True
```

Strings

Single, double, or triple quotes. 0 or more characters in between delimiters. We call these literals or hard-coded values.

```
>>> 'a'  # a single character string is sometimes called a character
'a'
>>> 'hi'
'hi'
>>> "hi"
'hi'
>>> """hi"""
'hi'
```

When you need to actually include quotes of some kind in the string, then you surround it with different quotes like `"Bob's house"`.

Special characters

`\n` is the newline character, `\t` is the tab character

```
>>> print "Cars:\n\tBMW\n\tAudi"
Cars:
    BMW
    Audi
```

which is the same thing as

```
>>> print "Cars:"
Cars:
>>> print "\tBMW"
    BMW
>>> print "\tAudi"
    Audi
```

or

```
>>> print "Cars:"
Cars:
>>> print "    BMW"
    BMW
>>> print "    Audi"
    Audi
```

Conversions

We can convert between numbers

```
>>> float(3)
3.0
>>> int(3.14159)
3
```

and numbers and characters

```
>>> chr(100)
'd'
>>> chr(105)
'i'
>>> ord('H')
72
>>> str(234)
'234'
```

Image Processing

The goal of this chapter is to exercise your understanding of all of the major components in Python: **assignments, expressions, if and loop statements, functions, lists, and libraries**. To make things interesting, we will perform some cool image processing tasks: **flipping horizontally, blurring, removing salt-and-pepper image noise, finding edges within images, and image sharpening**. For example, Figure 8 demonstrates image sharpening.

Figure 8: Bonkers the cat sharpened using `sharpen.py`.

Installing Pillow

For image processing, we'll use Pillow, so please follow the instructions carefully to install Pillow, a fork of PIL.

Install stuff on a mac

First make sure `brew` is installed; try this from Terminal app:

```
$ brew
```

If not there, go here: `http://brew.sh` and it'll say to install do this:

```
$ ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

then

```
$ brew install python
```

That should give you pip as well. Ignore warning about having 2 pythons installed. Make sure python is using the brew version of python. All brew stuff is in `/usr/local/Cellar/*` so you should see:

```
$ which python
/usr/local/bin/python
$ ls -l `which python`
lrwxr-xr-x 1 parrt admin 33 Jul 5 12:19 /usr/local/bin/python@ -> ../Cellar/python/2.7.9/bin/python
```

Now, install all of the software you need before installing Pillow:

```
$ brew install libtiff libjpeg webp little-cms2
```

then

```
$ pip install Pillow
```

If you get an error that ends with:

```
...
ImportError: cannot import name HTTPSHandler
```

then try reinstalling python:

```
$ brew reinstall python
```

Test it:

```
$ python
Python 2.7.9 (default, Jul 5 2015, 12:18:15)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import PIL
>>>
```

there should be no error when you type `import PIL`

Getting started

To get started, review how to access the command-line arguments that every program running on a computer can accept. Here is a small program, `view.py`, that accepts a filename as a command-line argument.

```
import sys
from PIL import Image
if len(sys.argv) != 2:
    print "$ python view.py imagefilename"
    sys.exit(1)
filename = sys.argv[1] # get the argument passed to us by operating system
img = Image.open(filename) # load file specified on the command line
img = img.convert("L") # grayscale
img.show()
```

To run this program, launch a terminal on Mac or Linux. Then, move to the directory that contains `view.py` that you saved/typed-in using terminal command `cd` (change directory). For example, if you are using directory `msan501/images` under your home directory for this project, then type this for Mac:

```
cd /Users/YOURID/msan501/images
```

and this for Linux:

```
cd /home/YOURID/msan501/images
```

You can get a bunch of sample images that I use for these notes from github:

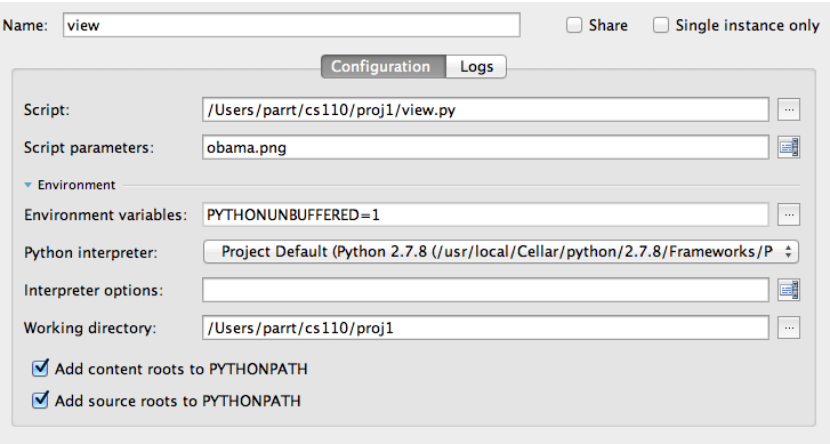
`https://github.com/parrt/msan501/tree/master/labs/figures`

but of course you can play around with whatever images you want.[1] Now, we can execute our `view.py` script and pass it an argument of `obama.png`:

```
$ python view.py obama.png
```

[1] Remember, however, that all images used in this class and those stored on University equipment must be "safe for work." Keep it G-rated please, with no offensive images popping up on your laptops or machines during lab etc.

If you are using PyCharm, then you need to use the "`Edit configuration`" dialog under the `Run` menu. All file names that we specify in the script parameters area, or from the command line, are relative to the *current working directory*. That is why we used `cd` to change our working directory in the example above. For convenience, let's keep all of images and scripts in the same directory. For our purposes, let's assume that the directory is always `/Users/YOURID/msan501/images` (Mac) or `/home/YOURID/msan501/images` (Linux).



The "Script parameters" text field is where you provide the "command-line arguments," despite the fact we are not actually using a command-line. That is why PyCharm calls it script parameters instead of command-line parameters.

Task 1. Flipping an image horizontally

As a first task, you must create a script called `flip.py` that shows the image provided as a program (command-line) argument in its original form and then flipped horizontally. For example, Figure 9 shows the result of running script `flip.py` on image `eye.png`.

Figure 9: Flipping an image horizontally; the original is on the left.

Boilerplate code

Here's the boilerplate or "skeleton" code (already in repo as `images/flip.py`) that we already know how to do but with a hole where you need to define a function called `flip` and a hole where you need to call that function to perform the flipping:

```
import sys
from PIL import Image

# define your flip function here
...
if len(sys.argv)<=1:
```

```
    print "missing image filename"
    sys.exit(1)
filename = sys.argv[1]
img = Image.open(filename)
img = img.convert("L")
img.show()

# call your flip function here
...
img.show()
```

If you are using PyCharm, your project file list area should look like the list in Figure 10. It assumes that you have downloaded two image files as well.

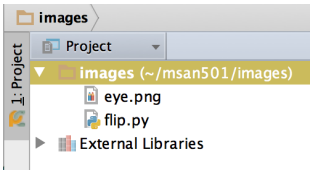


Figure 10: Project file list after building the flip task

The files are already in the repository because I did the following:

```
$ cd ~/msan501/images
$ git add eye.png
$ git add flip.py
$ git commit -a -m 'add eye flip boilerplate code and image'
[master adcf1df] add eye flip boilerplate code and image
 2 files changed, 16 insertions(+)
 create mode 100644 images/eye.png
 create mode 100644 images/flip.py
$ git push origin master
Counting objects: 5, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (5/5), 238.35 KiB | 0 bytes/s, done.
Total 5 (delta 0), reused 0 (delta 0)
To git@github.com:parrt/msan501-starterkit.git
  b5820e1..adcf1df master -> master
```

Three new Pillow pieces

To write your `flip` function, you will need three basic pieces:

- `img.size` returns a tuple containing the width and height of image `img` so you can write code like this:

  ```
  width, height = img.size
  ```

  You'll need the width and height to iterate over the pixels of the image.

- `img.copy()` duplicates image `img`. For our `flip` function, it would be hard to modify the image in place because we would be overwriting pixels we would need to flip later. It's easier to create a copy of the image in flipped position. You can write code like this:
  `imgdup = img.copy()`

- `img.load()` is yet another weird name from PIL that actually returns an object that looks like a two-dimensional matrix. We have previously seen two-dimensional lists, which are really like lists of lists such as `m = [[1,2], [3, 4]]` or reformatted to look like the matrix:

  ```
  m = [[1, 2],
       [3, 4]]
  ```

  To get element `3`, we would use list index expression `m[1][0]` because we want the list at index `1`, `m[1]`, and then element `0` within that list. The two-dimensional object returned by `load()` uses similar notation. If we ask for the "matrix" with:

  ```
  m = img.load()
  ```

  then we would use notation `m[x,y]` to get the pixel at position (`x`, `y`).

You will use these functions for the remaining tasks so keep them in mind.

Iterating over the image matrix

Define function `flip` using the familiar function definition syntax and have it take a parameter called `img`, which will be the image we want to flip. The goal is to create a copy of this image, flip it, and return a copy so that we do not alter the incoming original image. To create `flip`, write code that implements the following steps.

1. Use `size` to define local variables `width` and `height`
2. Use `copy()` to make a copy of the incoming image `img` and save it in a local variable
3. Use `load()` to get the two-dimensional pixel matrix out of the incoming image and the copy of the image. Store these results in two new local variables.
4. To walk over the two-dimensional image, we've learned we need every combination of `x` and `y`. That means we need a nested `for` loop. Create a nested for loop that iterates over all `x` and all `y` values within the `width` and `height` of the image.
5. Within the inner loop, set pixels in the image copy to the appropriate pixel copied from the original image
6. At the end of the function, return the flipped image

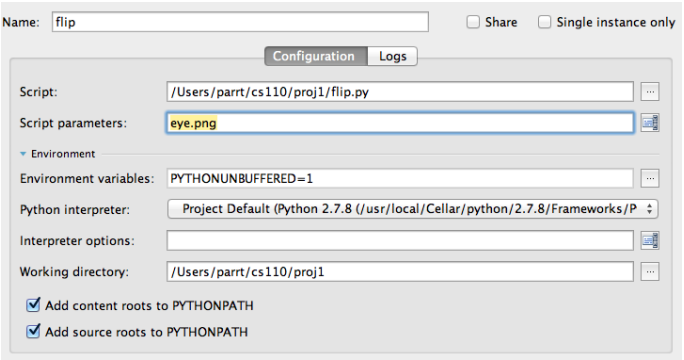
The only remaining issue is determining which pixel from the original image to copy into the (`x`, `y`) position in the image copy. The `y` index will be the same since we are flipping horizontally. The `x` index in the flipped image is index `width-x-1` from the original image. Trying out a few sample indexes shows that this works well. For example, a flipped image with `width=10` has its pixel at `x=0` pulled from index `x=10-0-1=9` in the original image. That's great, because it takes the image from all in the right in the original and copies it to the far left of the copy. Checking the opposite extreme, `x=9` in the flipped image should be `x=10-9-1=0` from the original image.

Running your flip script

To run the script from the command line, make sure you are in the `msan501/images` directory containing your scripts and images then do:

```
$ cd msan501/images
$ python flip.py eye.png
```

From PyCharm, I right-click and then select "Run." It will do nothing but print "`missing image filename`" because we have not given it a parameter yet, but we need to do this so that PyCharm creates a run configuration. Now, use the edit configuration dialog to specify `eye.png` as a parameter or any other image. (*Remember that the filename must refer to a file in the current working directory or must be fully qualified.*) Click "Run" now and it should bring out the two images.



What to do when the program doesn't work

If you have problems, follow these steps:

1. Don't Panic! Relax and realize that you will solve this problem, even if it takes a little bit of messing around. Banging your head against the computer is part of your job. Remember that the

computer is doing precisely what you tell it to do. There is no mystery.

2. Determine precisely what is going on. Did you get an error message from Python? Is it a syntax error? If so, review the syntax of all your statements and expressions. PyCharm is your friend here and should highlight erroneous things with a red squiggly underline. If you got an error message that has what we call a stack trace, a number of things could be wrong. For example, if I misspell `show()` as `shower()`, I get the following message:

```
Traceback (most recent call last):
  File "/Users/parrt/msan501/images/flip.py", line 26, in <module>
    img.shower()
  File "/usr/local/lib/python2.7/site-packages/PIL/Image.py", line 605, in __getattr__
    raise AttributeError(name)
AttributeError: shower
```

   In PyCharm, the "`/Users/parrt/msan501/images/flip.py`" part of the trace will be a blue link that you can click on. It will take you to the exact location in your script where there is a problem. Look for anything that refers to that file.

3. If it does not look like it some simple misspelling, you might get lucky and find something in Google if you cut-and-paste that error message.

4. If your script shows the original image but not the flipped image, then you likely have a problem with your `flip` function.

5. If your program is at least running and doing something, then insert print statements to figure out what the variables are and how far into the program you get before a craps out. That often tells you what the problem is.

6. Try using the debugger to step through your program in PyCharm. Set a breakpoint on for example the line `filename = sys.argv[1]` by clicking in the gutter to the left of that line. A red dot should appear, indicating there is a breakpoint there. Then click the little bug icon instead of the green triangle (which is run button). That will start execution and then stop at that line. You can look at all of the variables at that point. Then you can step forward line by line. Read how to use the debugger online.

7. Definitely try to solve the problem yourself, but don't waste too much time. I can typically help you out quickly so you can move forward.

Deliverables. Make sure that `images/flip.py` is correctly committed to your repository and pushed to github.

Task 2. Blurring

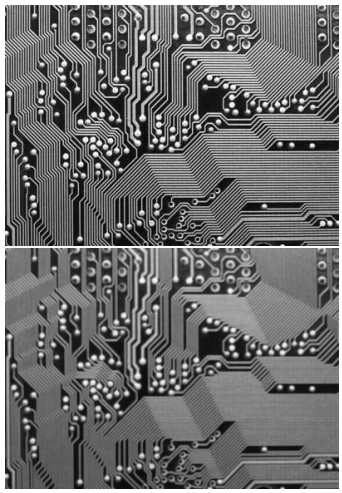


Figure 11: Blurring of a circuit board; the original is on top.

In this task, we want to blur an image by removing detail as shown in Figure 11. We will do this by creating a new image whose pixels are the average of the surrounding pixels for which we will use a 3x3 region as shown in Figure 12. The pixel in the center of the region is the region to compute as we slide the region around an image. In other words, `pixel[x,y]` is the sum of `pixel[x,y]` and all surrounding pixels divided by 9, the total number of pixels.

To implement this, start with the boilerplate from the previous section, which you should put into script `blur.py`. The only difference is that you must call soon-to-be-created function `blur` not `flip` as you had before. Now, let's start at the courses level of functionality and realize that we have to walk over every pixel in the image. (This is called *top-down design*.)

Blurring function

Define function `blur` to take an `img` parameter, just like the `flip` function in the previous task. In a manner very similar to `flip`, write code in `blur` to accomplish these steps:

1. Define local variables `width` and `height`.
2. Make a copy of the incoming image `img` and save it in a local variable.
3. Get the two-dimensional pixel matrix out of the image copy. Store it in a new local variable called `pixels`.
4. Create a nested for loop that iterates over all `x` and all `y` values within the `width` and `height` of the image.
5. Within the inner loop:
   (a) Call to-be-created function `region3x3` with arguments `img`, `x`, and `y` in store into local variable `r`.
   (b) Set `pixels[x,y]` in the image copy to the result of calling to-be-created function `avg` with an argument of `r`.
6. At the end of the function, return the flipped image.

Following the top-down design strategy, let's **define function** `avg` since it's the easiest. Define `avg` to take an argument called `data` or another of your choice. This will be the list of 9 pixels returned by function `region3x3`. The average of a set of numbers is their total divided by how many numbers there are. Python provides two useful functions here: `sum(data)` and `len(data)`. (Naturally, `sum` simply walks the list and accumulates values using a pattern we are familiar with.)

Image regions

Now we need to **define function** `region3x3`. Have it take three parameters as described above. This function creates and **return a list of nine pixels**. The list includes the center pixel at `x`, `y` and the 8 adjacent pixels at N, S, E, W, ... as shown in Figure 12. Create a series of assignments that look like this:

```
me = getpixel(img, x, y)
N = getpixel(img, x, y - 1)
...
```

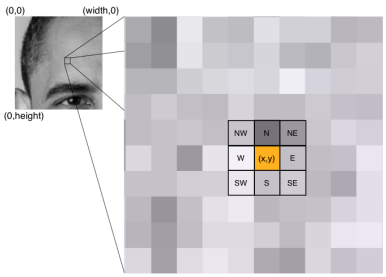


Figure 12: Hyper-zoom of Obama's forehead showing 3x3 region.

where function `getpixel(img, x, y)` gets the pixel at `x`, `y` in image `img`. We can't use the more readable expression `pixels[x,y]` in this case, as we'll see in a second. Collect all those pixel values into a list using `[a,b,c,...]` list literal notation and return it. Make sure that this list is a list of integers and exactly 9 elements long and that you keep in mind the order in which you add these pixels to the list. Any function that we create to operate on a region naturally needs to know the order so we can properly extract pixels from the list. For example, my implementation always puts the pixel at `x` and `y` first, then North, etc...

Safely examining region pixels

We need to `define a function` `getpixel` instead of directly accessing pixels because some of the pixels in our 3x3 region will be outside of the image as we shift the region around. For example, when we start out at `x=0`, `y=0`, 5 of the pixels will be out of range, as shown in Figure 13. Accessing `pixels[-1,-1]` will trigger:

```
IndexError: image index out of range
```

and stop the program. To avoid this error and provide a suitable definition for the ill-defined pixels on the edges, we will use a function that ensures all indices are within range.

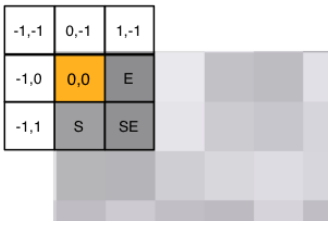


Figure 13: Our 3x3 region has pixels outside of the image boundaries as we slide it around the image along the edges.

Define function `getpixel` with the appropriate parameters. Its functionality is as follows:

1. Get the width and height into local variables.
2. If the `x` value is less than 0, set it to 0.
3. If the `x` value is greater than or equal to the width, set it to the width minus 1 (the last valid pixel on the far right).
4. If the `y` value is less than 0, set it to 0.
5. If the `y` value is greater than or equal to the height, set it to the height minus 1 (the last felt pixel on the bottom).
6. Return the pixel at `x`, `y`. You will need to use the `img.load()` function again to get the 2D `pixels` matrix as you did in function `blur`. Make sure you returning pixel and not the coordinates of the pixel from `getpixel`.

Testing your blur code

That is a lot of code to enter and so more than likely it won't work the first time.[2] That means we should test the pieces. It's generally a good idea to do top-down design but *bottom-up testing*. In other words, let's test the simple low-level functions first and make sure that works before testing the functions that call those functions and so on until we reach the outermost script.

[2] It never does, dang it!

With that in mind, lets test `avg` by passing it a fixed list of numbers to see if we get the right number. Add this to your script before it does any of the file loading stuff:

```
print avg([1,2,3,4,5])
```

Then run `blur.py` with any old image; the `apple.png` file is a good one because it's small:

```
$ python blur.py apple.png
3
$
```

If it does not print $(1+2+3+4+5)/5 = 3$, then you know you have a problem in `avg`.

Now test `getpixel`. You will have to insert some code after loading and converting the image to grayscale because `getpixel` takes an image parameter:

```
img = Image.open(filename)
img = img.convert("L")
print getpixel(img, 0, 0)
print getpixel(img, 0, 1)
print getpixel(img, 10, 20)
```

That should print: 96, 96, and 255. The upper left corner is gray and pixel 10, 20 is somewhere in the middle of the white Apple logo. If you don't get those numbers, then you have a problem with `getpixel`. Worse, if you don't get simple numbers, then you really have a problem with `getpixel`.

Before getting to `blur`, we should also **test** `region3x3` to ensure it gets the proper region surrounding a pixel. Replace those `getpixel` calls in the `print getpixel` statements with calls to `region3x3`. Use the `x`, `y` of the upper left-hand corner and somewhere near the upper left of the white section of the logo such as:

```
print region3x3(img, 0, 0)
print region3x3(img, 7, 12)
```

That checks whether we get an out of range error at the margins and that we get the correct region from somewhere in the middle. Running the script should give you the following numbers:

```
$ python blur.py apple.png
[96, 96, 96, 96, 96, 96, 96, 96, 96]
[255, 176, 255, 255, 215, 96, 245, 255, 255]
$
```

That assumes order: current pixel, N, S, E, W, NW, NE, SE, SW.

When you have verified that all of these functions work, it's time to check function `blur` itself. Try the printed circuit board image:

```
$ python blur.py pcb.png
```

That should pop up the original circuit board and the blurred version. It might take 10 seconds or more to compute and display the blurred image, depending on how fast your computer is.

Make sure to remove all of your debugging code before submitting your scripts. Submitting a project with a bunch of random debugging output is considered sloppy, like submitting an English paper with a bunch of handwritten edits.

Deliverables. Make sure that `images/blur.py` is correctly committed to your repository and pushed to github.

Task 3. Removing noise

For our next task, we are going to de-noise (remove noise) from an image as shown in Figure 14. It does a shockingly good job consider-

ing the simplicity of our approach. To blur, we used the average of all pixels in the region. To denoise, we will use the median, which is just the middle value in a list of ordered numbers.

Believe it or not, we can implement the noise by copying `blur.py` into a new script called `denoise.py` and then changing a few lines. We also have to remove the no-longer-used `avg` function and replace it with a `denoise` function. Of course, instead of calling `blur`, we'll call function `denoise` with the usual `img` argument. The only difference between `denoise` and `blur` is that you will set the pixel to the `median` not `avg`. Hint: you need to tweak one statement in the inner loop that moves over all pixel values.

Now define function `median` that, like `avg`, takes a list of 9 numbers called `data`. Sort the list using Python's `sorted` function that takes a list and returns a sorted version of that list. Then compute the index of the middle list element, which is just the length of the list divided by two. If the length is even, dividing by 2 (not 2.0) will round it down to the nearest index. Once you have this index, return the element at that index.

Let's give it a test:

```
$ python denoise.py guesswho.png
```

That should pop up the noisy Obama and the cleaned up version. You can save the cleaned up version and run `denoise.py` on that one to really improve it.[3] Running `denoise.py` twice, gives the cleaned up image (c) from Figure 14.

Deliverables. Make sure that `images/denoise.py` is correctly committed to your repository and pushed to github.

Figure 14: Denoising an image of President Obama with lots of salt-and-pepper noise. (a) the noisy image, (b) denoised as computed by `denoise.py`, (c) denoised 2x, (d) original.

Task 4. Re-factoring to improve code quality

As I mentioned in the last task, `blur.py` and `denoise.py` are virtually identical, meaning that we have a lot of code in common. Figure 15 demonstrates this visually. One of the most important principles of computer science is to reduce code duplication. We always want exactly one place to change a particular bit of functionality. In this case, we have the following common code:

1. Functions `getpixel` and `region3x3`.
2. The "main" part of the script that loads the original image.

[3] Hint: **To save an image with PIL**, use `img.save("filename.png")`.

Figure 15: Visual difference between scripts `blur.py` and `denoise.py`. The files are identical except for orange marks.

3. Functions `blur` and `denoise` are identical except for the function called to compute a new pixel in the image copy from a 3x3 region in the original (`avg` or `median`).

The goal of this task is to make new versions, `blur2.py` and `denoise2.py`, that share as much code as possible. The functionality will be the same, but they will be much smaller and we will get warm feeling that our code is well structured. To share code, we need to (a) put all common code in a file that these new scripts can import and (b) create a generic function called `filter` that will work for both blurring and de-noising in an image. Once we get the common code into a single file, which we will call `filter.py`, we can import it into `blur2.py` and `denoise2.py` like this:[4]

[4] Do not confuse script names with function names; `filter.py` can contain anything we want and it so happens that we will also put a function in there with the same name, `filter`.

```
from filter import *
```

That statement asks Python to look in the current directory (among other places we don't care about for now) for file called `filter.py` and import all of the functions. You can think of it as a formalized cut-and-paste.

Let's start by creating new "library" file `filter.py` and placing our usual imports at the top:

```
import sys
from PIL import Image
```

Now, copy functions `getpixel` and `region3x3` into it.

We can also create a function called `open` to hide all of the messiness that checks for command-line arguments and opens the indicated image:

```
def open(argv):
    if len(argv)<=1:
        print "missing image filename"
        sys.exit(1)
    img = Image.open(argv[1])
    img = img.convert("L")  # make greyscale if not already (luminance)
    return img
```

The only tricky bit[5] is to create a single generic `filter` function that can reproduce the functionality we have in functions `blur` and `denoise`.

[5] Pun intended

1. Define function `filter` to take `img` and `f` parameters.
2. Copy the body of function `blur` into your new `filter` function.
3. Replace the call to `avg(r)` with `f(r)`.

As we discussed in class, functions are objects in Python just like any strings, lists, and so on. That means we can pass them around as function arguments.[6] To use our new generic `filter` function, we pass it an image as usual but also the name of a function:

```
blurred  = filter(img, avg)
denoised = filter(img, median)
```

[6] Don't confuse the name of a function with an expression that calls it. Assignment `f = avg` makes variable `f` refer to function `avg`. `f = avg()` **calls** function `avg` and stores the return value in variable `f`. Using `f = avg`, we can call `avg` with expression `f()`. You can think of `f` as an alias for `avg`.

In the end, your `filter.py` script file should have 4 functions: `getpixel`, `region3x3`, `filter`, and open.

Armed with this awesome new common file, our entire `blur2.py` file shrinks to a tiny script:[7]

```
from filter import *
# Your avg function goes here (copy from blur.py)
...
img = open(sys.argv)
img.show()
img = filter(img, avg)        # blur me please
img.show()
```

[7] You might be wondering why we don't have to include the usual `sys` and `PIL` imports at the start of our new files. That is because we import our `filter.py` file, which in turn imports those files.

The `denoise2.py` script is also tiny:[8]

```
from filter import *
# Your median function goes here (copy from denoise.py)
...
img = open(sys.argv)
img.show()
img = filter(img, median)    # denoise me please
img.show()
```

[8] Yep, these files are identical except for the fact that we call `filter` with different function names. If you wanted to get really fancy, you could replace both of these scripts with a single script that took a function name as a second argument (after the image filename). With some magic incantations, you'd then ask Python to lookup the function with the indicated name and pass it to function `filter` instead of hard coding.

Before finishing this task, be a thorough programmer and test your new scripts to see that they work:

```
$ python blur2.py pcb.png
$ python denoise2.py guesswho.png
```

They *should* work, but unfortunately that is never good enough in the programming world. Lot of little things can go wrong. *Certainty* is always better than *likelihood*.

We will import file `filter.py` into all of our future scripts. You have created your first useful library. **Good job!** ☺

Deliverables. Make sure that `images/blur2.py`, `images/denoise2.py`, and `images/filter.py` are correctly committed to your repository and pushed to github.

Task 5. Highlighting image edges

Now that we have some basic machinery in `filter.py`, we can easily build new functionality. In this task, we want to highlight edges found within an image. It is surprisingly easy to capture all of the important edges in an image, as shown in image (b) from Figure 17.

The mechanism we're going to use is derived from some serious calculus kung fu called the *Laplacian*, but which, in the end, reduces to 4 additions and a subtraction! The intuition behind the Laplacian is that abrupt changes in brightness indicate edges, such as the transition from the darkness of a uniform to the brightness of a windshield edge. As we did for blurring and denoising, we are going to slide a 3x3 region around the image to create new pixels at each `x`, `y`. That means we can reuse our `filter` function—we just need a `laplace` function to pass to `filter`.

To get started, here is the boilerplate code copied from `denoise2.py` but with function name `laplace` (the object of this task) passed as an argument to function `filter`:

```
from filter import *
# define function laplace here
...
img = open(sys.argv)
img.show()
edges = filter(img, laplace)
edges.show()
```

Create function `laplace` that takes region `data` as an argument as usual. Have the function body return the sum of the North, South, East, and West pixels minus 4 times the middle pixel from our usual region:

NW	N	NE
W	(x,y)	E
SW	S	SE

(a) (b)

Figure 16: Edges of an old photograph from World War II. (a) original, (b) edges as computed by `edges.py`.

That computation effectively compares the strength of the current pixel with those around it.

For those familiar with calculus, we are using the second partial derivative (i.e., acceleration) in x and y directions. The first derivative would detect edges even for gradual changes but the second derivative detects only really sharp changes. For a pixel fetching function f operating on a 3x3 region around (x, y), "applying the *Laplacian*" means computing a filtered image pixel at x, y as: $f(x+1,y)+f(x-1,y)+f(x,y+1)+f(x,y-1)-4f(x,y)$ where $f(x,y)$ is equivalent to our `pixels[x,y]`.

For example, imagine a region centered over a vertical white line. The region might look like:

	255	
0	255 (x,y)	0
	255	

The `laplace` function would return $255 + 255 + 0 + 0 - 4 \times 255 = -510$.

Compare that to the opposite extreme where values are almost the same:

	18	
21	10 (x,y)	15
	19	

The `laplace` function would return $18 + 19 + 15 + 21 - 4 \times 10 = 33$.

Once you have implemented your `laplace` function, give it a try with some of the sample images you have such as the jeep or Obama:

```
$ python edges.py obama.png
```

It actually does a really good job capturing Obama's outline:

Be aware of something that Pillow is doing for us automatically when we store values into an image with `pixels[x,y] = v`. **If `v` is out of range 0..255, Pillow clips** `v`. So, for example, `pixels[x,y] = -510` behaves like `pixels[x,y] = 0` and `pixels[x,y] = 510` behaves like `pixels[x,y] = 255`. It doesn't affect edge detection or any of our other operations in future tasks but I wanted to point out that in a more advanced class we would **scale** these pixel values instead of clipping them. Clipping has the effect of reducing contrast.

Deliverables. Make sure that `images/edges.py` is correctly committed to your repository and pushed to github.

Task 6. Sharpening

Sharpening an image is a matter of highlighting the edges, which we know how to compute from the previous tasks. Script `edges.py` computes just the edges so, to highlight the original image, we *subtract* that white-on-black edges image from the original. You might imagine that *adding* the edges back in would be more appropriate and it sort of works, but the edges are slightly off. We get a better image by subtracting the high-valued light pixels because that darkens the edges in the original image, such as between the uniform and the windshield. Let's start with the easy stuff:

(a) (b)

Figure 17: Bonkers the cat portrait. (a) original and (b) sharpened as computed by `sharpen.py`.

1. Copy your previous `edges.py` file to new script `sharpen.py`.
2. After `edges = filter(img, laplace)`, add a line that calls a function we'll create shortly called `minus`. `minus` takes two image parameters, `A` and `B` and returns `A-B`. In our case, pass in the original image and the image you get back from calling `filter(img, laplace)`.
3. Show the result of the `minus` function.

That only leaves the task of **creating function** `minus` to subtract the pixels of one image from the pixels of another image like a 2-D matrix subtraction. As we did before, we will return a modified version of a copy of an incoming image parameter. (In my solution, I arbitrarily chose to create and return a copy of `A`.) Because you are getting really good at creating functions to manipulate images, the instructions for creating `minus` in this task are less specific than in previous tasks. You need to fill in the body of this function:

```
# Return a new image whose pixels are A[x,y] - B[x,y]
def minus(A, B):
    ...
```

The mechanism is the same as before: iterating loop variables `x` and `y` across the entire image and processing the pixel at each location. The only difference between this function and `filter` is that we want to operate on individual pixels not 3x3 regions. In the inner

loop, set `pixels[x,y]` to the value of pixel `A[x,y]` minus pixel `B[x,y]`. Don't forget to return the image you filled in.

Here's how to run `sharpen.py` on Bonkers the cat:

```
$ python sharpen.py bonkers-bw.png
```

Figure 18, Figure 19, and Figure 20 show some sample transformation sequences with original, *Laplacian*, and sharpened images.

Deliverables. Make sure that `images/sharpen.py` is correctly committed to your repository and pushed to github. Tag when completed with `images`.

Figure 18: Sharpening of a Bonkers the cat. Clockwise: (a) original, (b) edges as computed by `edges.py`, (c) the sharpened image as computed by `sharpen.py`.

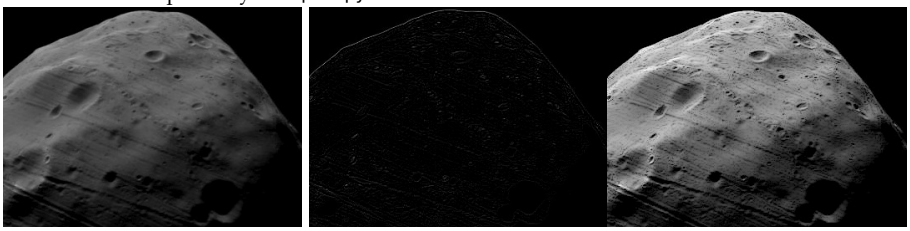


Figure 19: Sharpening of Phobos asteroid from NASA. Clockwise: (a) original, (b) edges as computed by `edges.py`, (c) the sharpened image as computed by `sharpen.py`.

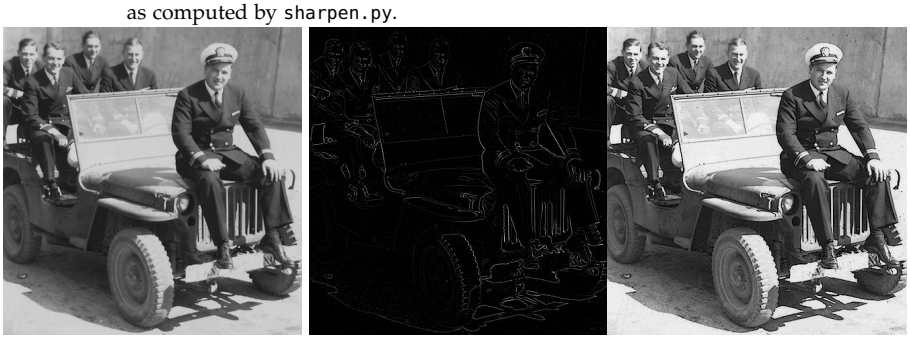


Figure 20: Sharpening of an old photograph from World War II. Clockwise: (a) original, (b) edges as computed by `edges.py`, (c) the sharpened image as computed by `sharpen.py`.

Graph Adjacency Lists and Matrices

Goal

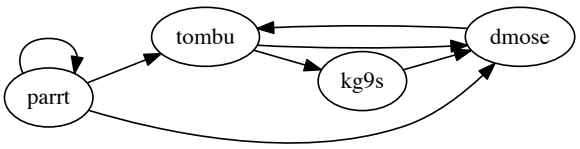
The goal of this task is to teach you about the implementation of graphs in Python, how to implement a few simple related algorithms, and do some simple data loading. As part of this exercise, you will also learn to transform data, which is an important data preparation skill you will need as an analyst.

Discussion

In this project, you will convert a string representation of a graph that looks like this:

```
parrt: tombu, dmose, parrt
tombu: dmose, kg9s
dmose: tombu
kg9s: dmose
```

to an adjacency list representation and ultimately generate a visual representation via graphviz/dot:



For fun, you will also create an edge matrix representation:

$$\begin{array}{c} \\ parrt \\ tombu \\ dmose \\ kg9s \end{array} \begin{array}{c} \begin{array}{cccc} parrt & tombu & dmose & kg9s \end{array} \\ \begin{pmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \end{array}$$

where the nodes have the following indexes (all we really care about here is the order):

$$\begin{bmatrix} 0 : parrt \\ 1 : tombu \\ 2 : dmose \\ 3 : kg9s \end{bmatrix}$$

The following sections describe the functions you must create in `graph.py`. Your starter kit should have some boilerplate code set up for you.

Processing an adjacency list string

First, you have to process a string representation of an adjacency list and create an internal data structure:

```
def adjlist(adj_list):
    """
    Read in adj list and store in form of dict mapping node
    name to list of outgoing edges. Preserve the order you find
    for the nodes.
    """
    ...
```

You will use an ordered dictionary (`OrderedDict`) that maps node name x to a list of target nodes. x will be a string and the target list will be a list of strings. For example, from line in string adj_list

`parrt: tombu, dmose, parrt`

you will create an entry in the dictionary with key `parrt` and value:

```
['tombu', 'dmose', 'parrt']
```

To process the text, you must split the incoming string into lines and then process them one at a time as each line represents an adjacency list. You will use string functions `split` and (likely) `strip` to process the text. The goal here is to learn how to process text so don't look for built-in functions that do all of this for you.
Printing the adjacency list dictionary from `adjlist`, we should all get the following output:

```
OrderedDict([('parrt', ['tombu', 'dmose', 'parrt']),
 ('tombu', ['dmose', 'kg9s']),
 ('dmose', ['tombu']),
 ('kg9s', ['dmose'])])
```

Adjacency list to adjacency matrix

Given an adjacency list stored as a dictionary per adjlist(), create a function that converts it to an adjacency matrix:

```
def adjmatrix(adj):
    """
    From an adjacency list, return the adjacency matrix with entries in {0,1}.
    The order of nodes in adj is assumed to be same as they were read in.
    """
    ...
```

The matrix should look like the one shown above.

Getting a list of all nodes

A very useful function to have is the following that returns a list of all nodes visited starting at a particular node in a graph.

```
def nodes(adj, start_node):
    """
    Walk every node in graph described by adj list starting at start_node
    using a breadth-first search.  Return a list of all nodes found (in
    any order). Include the start_node.
    """
    ...
```

Do not build a recursive function as you must do a breadth-first search. (Recursive functions are much more useful when doing a depth-first search.) The basic algorithm looks like this:

```
visited = [];
add the start node to a work list;
while more work do
  node = remove a node from work list;
  add node to visited list;
  targets = adjacency_list[node];
  add all unvisited targets to work list;
end
return visited;
```

Generating DOT output

In order to visualize the graph you have read in, create the following function that dumps valid Graphviz DOT code, given an adjacency list. Then cut-and-paste the output and put it into Graphviz to display it.

```
def gendot(adj):
    """
    Return a string representing the graph in Graphviz DOT format
    with all p->q edges. Parameter adj is an adjacency list.
    """
    ...
```

Or, to amaze your family and friends, you can directly from the command line on a mac or unix box:

```
$ python test_dot.py | dot -Tpdf > /tmp/graph.pdf; open /tmp/graph.pdf
```

Here is a simple test rig, `test_dot.py`, that translates an input string description to DOT and prints it out.

```
from graph import *

# test dot
g = \
"""
parrt: tombu, dmose, parrt
tombu: dmose, kg9s
dmose: tombu
```

```
kg9s: dmose
"""
list = adjlist(g)
dot = gendot(list)
print dot
```

For the adjacency list shown at the start of this assignment, you should to generate the following DOT code:

```
digraph g {
  rankdir=LR;
  parrt -> tombu;
  parrt -> dmose;
  parrt -> parrt;
  tombu -> dmose;
  tombu -> kg9s;
  dmose -> tombu;
  kg9s -> dmose;
}
```

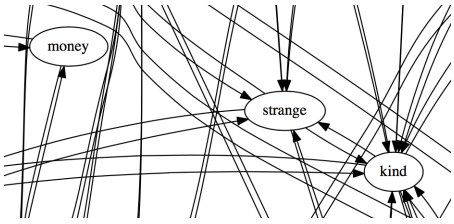
Testing

I have provided `test_graph.py` and `test_dot.py` test rigs that exercise the required functions using the sample adjacency list described above. Please make sure that your library works with this test rig.

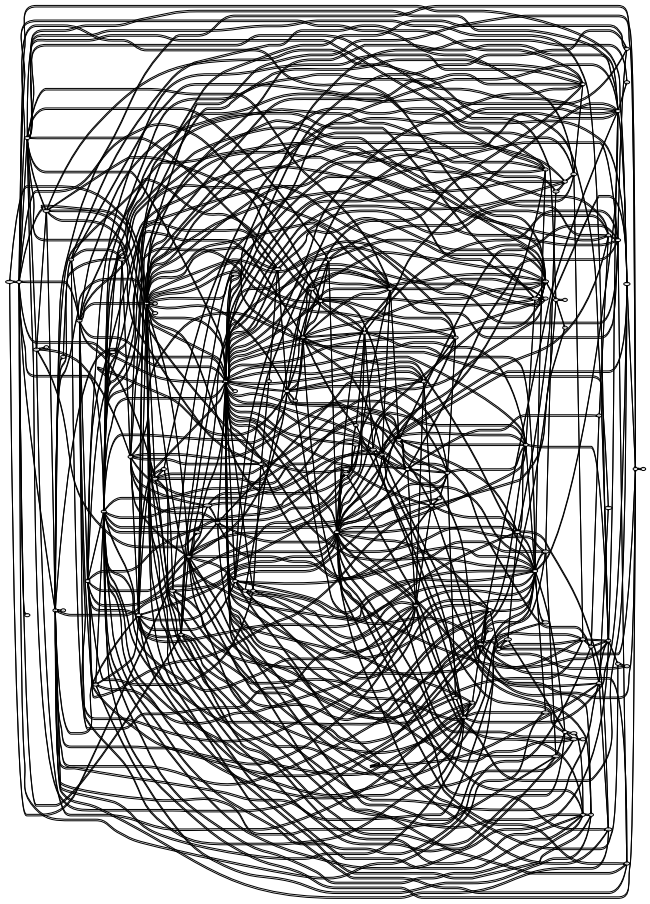
Real data

EXTRA CREDIT

And, now, let's try something real though still pretty small. Get the Word adjacencies in the book David Copperfield as a graph in `.gml` (graph modeling language) format from M. E. J. Newman, *Finding community structure in networks using the eigenvectors of matrices*, Preprint physics/0605087 (2006). Visually, the graph looks like this:



Or, in its full glory:



You will need to install package NetworkX to read in the graph modeling language file format:

```
$ sudo pip install NetworkX
```

The data looks like this in text format:

```
Creator "Mark Newman on Fri Jul 21 13:00:02 2006"
graph
[
  node
  [
    id 0
    label "agreeable"
    value 0
  ]
  node
  [
    id 1
    label "man"
    value 1
  ]
...
  edge
  [
```

```
    source 32
    target 15
  ]
...
```

(By the way, I had to install NetworkX and pyparsing within PyCharm because it did not see the `pip` installations I did from the command line.)

You must create `wordgraph.py` that reads accepts a .gml file as a commandline argument (i.e., our downloaded `adjnoun.gml` file), converts it to an adjacency list with a `gml2adjlistt()` function, and then generates DOT with your `gendot()` function. Make sure to keep the same order in the adjacency list you create as found in the .gml file. Here is the boilerplate function in the `wordgraph.py` starter kit in the repository:

```
def gml2adjlist(G):
    """
    Return a dict mapping word to adjacent nodes. G.node dict in memory
    looks like:

    {0: {'id': 0, 'value': 0, 'label': 'agreeable'},
    1: {'id': 1, 'value': 1, 'label': 'man'}, ... }

    and G.edge dict looks like:

    {0: {1: {}, 2: {}, 3: {}}, 1: {0: {}, 19: {}, 2: {}, 102: {}, ...}, ...}

    and we need:

    {agreeable:['man', 'old', 'person'], man:[['agreeable', 'best', 'old', ...], ...}
    """
    words = collections.OrderedDict()  # keep stuff in order read from GML
    ...
    return words
```

The code is not too tricky, but you have to figure out how to extract the `label` from each node to get the node list. The edges start out as node ids not words so you need to convert those to words in order to create an appropriate adjacency list.

Save the DOT output to a file and compare it to the `wordgraph.dot` file I provide in the repository. I.e., the following command should generate no output because there should be no difference between the files:

```
$ diff wordgraph.dot yourfile.dot
```

Deliverables. Make sure that `graphs/graph.py` (the functions inside should emit no output at all, just return data as specified), `graphs/wordgraph.py` are correctly committed to your repository and pushed to github. Tag when completed with `graphs`.

Part IV

Empirical statistics

Watch out for these issues

in progress. warnings for newbies.

`-1/2` is `-1` and `1/2` is `0` in Python. There is no automatic promotion when you send an integer to a function that is expecting a floating-point number.

(-1/lambduh)*(np.log(1-u)) vs -(np.log(1-u))/lambduh. The latter is probably better because it does fewer floating-point operations and hence probably has fewer errors.

This doesn't do what you think: X = [[0] * N] * TRIALS

Don't do this:

```
# defining function open here
def open(argv):
    ...
x = flip() # call flip
```

Computing Point Statistics

Discussion

The goal of this task is to refresh your memory of a few point statistics.

Stats

This exercise involves writing functions to compute sample mean, variance, and covariance from a data set (list of values). In mathematics notation, the sample estimates are:

$$\bar{x} = \frac{1}{N}\sum_{i=1}^{N} x_i \qquad \text{(Sample mean)}$$

$$s^2 = \frac{1}{N-1}\sum_{i=1}^{N}(x_i - m)^2 \qquad \text{(Unbiased sample variance)}$$

$$cov(x,y) = \frac{1}{N-1}\sum_{i=1}^{N}(x_i - \bar{x})(y_i - \bar{y}) \qquad \text{(Unbiased sample covariance)}$$

In Python, you must define functions `mean(x)`, `var(x)`, `cov(x,y)` where `x` and `y` are objects that behave like a list or iterator. The functions return a floating-point value based on the above mathematics notation. If the length of the incoming vectors to `cov` are not the same, return 0.

Libraries

While we're at it let's learn about creating and importing our own libraries. You'll notice that `test_point_stats.py` references your code like this:

```
from stats import *
```

That lets us directly access the functions defined in the `stats.py` file you are going to create.

You can test the correctness of the functions by using the `numpy` lib, make sure you ask for the sample population statistics by using parameter `ddof=1` for `var()` and `cov()`. E.g., `np.var(data, ddof=1)`. Be careful not to confuse function names; e.g., `numpy` has functions with the same names (although cov() returns a covariance matrix).

```
import numpy as np  # np is an alias for the numpy library
x = ...
y = ...
print np.cov(x,y)[0][1] # np.cov returns cov matrix
```

We now have the kernel of a small statistics library in `stats.py` and we will continue to add functions to this as we go along.

Testing

In computer science, programmers recognize two primary kinds of tests: *unit tests* and *functional tests*. A unit test is really just testing a function or a few functions whereas functional tests test the overall functionality of the program. In file `test_point_stats.py`, I have provided a set of unit tests that you can use for basic sanity checking of your project.

To make the unit tests work, make sure that you install py.test, which is usually just a matter of:

```
easy_install -U pytest
```

Test your code using the following command line (with your `stats.py` is in the same directory):

```
$ python -m pytest test_point_stats.py
============================= test session starts ==============================
platform darwin -- Python 2.7.6 -- py-1.4.30 -- pytest-2.7.2
rootdir: /Users/parrt/msan501/stats, inifile:
collected 3 items

test_point_stats.py ...
=========================== 3 passed in 0.01 seconds ===========================
```

If you don't see all tests passing, and there is a problem at a basic level with your software.

You may not use sum() or any other built-in functions for this project to compute the point statistics. The whole point of the exercise is to learn to build your own for loops. Obviously.

Deliverables. Make sure that `stats/stats.py` (the functions inside should emit no output at all, just return data as specified) is correctly committed to your repository and pushed to github.

Approximating $\sqrt{n}$ with the Babylonian Method

Motivation

This lab shows how to encode and solve a recurrence relation from mathematics using iteration in Python to approximate a real-world, real valued function. It also teaches how to quickly prototype something in Excel (if warranted).

Discussion

To approximate square root, $\sqrt{n}$, the idea is to pick an initial estimate, x_0, and then iterate with better and better estimates, x_i, using the recurrence relation:

$$x_{i+1} = \frac{1}{2}(x_i + \frac{n}{x_i})$$

To see how this works, jump into Excel (yes, a spreadsheet) and crank through a few iterations by defining cells with n and your initial estimate x_0, which can be anything you want. (It's sometimes easier to play around without having to deal with a programming language.) Then you need to define a cell that computes the above better approximation using x_i as the cell above it. I hardcoded the names in column A and the first two rows of column B. Cell B3 should be a formula that computes B4 based upon B3. Then you can extend the formula down and watch it converge on the final (correct) value for $\sqrt{125348}$. My spreadsheet looks like this:

	A	B
1	n	125348
2	x_0	20
3	x_1	3143.7
4	x_2	1591.78638
5	...	835.266564
6		492.668011
7		373.547461
8		354.554285
9		354.04556
10		354.045195
11		354.045195

Try out any nonnegative number and you'll see that it still converges, though at different rates.

There's a great deal on the web you can read to learn more about why this process works but it relies on the average (midpoint) of x and n/x getting us closer to $\sqrt{n}$. It can be shown that if x is above $\sqrt{n}$ then n/x is below $\sqrt{n}$ and the reverse is true if x is below the root. The iteration converges and does so quickly. Informally, as shown in Wikipedia, we can represent the true square root by adding an error term to our estimate:

$$\sqrt{n} = x + \epsilon$$

or,

$$n = (x + \epsilon)^2$$

Expanding, we get:

$$n = x^2 + 2x\epsilon + \epsilon^2$$

Solving for ϵ:

$$n - x^2 = \epsilon(2x + \epsilon)$$

$$\epsilon = \frac{n - x^2}{2x + \epsilon}$$

Because ϵ is much smaller than x, we can drop it from the denominator leaving us with an estimate of epsilon:

$$\epsilon = \frac{n - x^2}{2x}$$

Then we can plug it back into $x + \epsilon$ and get:

$$x := x + \epsilon = x + \frac{n - x^2}{2x} = \frac{2x^2}{2x} + \frac{n - x^2}{2x} = \frac{1}{2}\frac{x^2 + n}{x} = \frac{1}{2}(x + \frac{n}{x})$$

Which gets its back to the Babylonian formula. Since we dropped an ϵ term, this formula for x is inexact but it gets us closer to $\sqrt{n}$.

Now that you understand how this estimate works, your goal is to implement a simple Python method called sqrt() that uses the Babylonian method to approximate the square root. File `sqrt.py` is in you are repository with some starter code. You will also find unit tests in `stats/test_sqrt.py`, which you can run with:

```
$ python -m pytest test_sqrt.py
```

You may not use math.sqrt() for implementing your function, but you may use it for testing the results. Obviously.

Deliverables. Make sure that `stats/sqrt.py` is correctly committed to your repository and pushed to github.

Generating Uniform Random Numbers

Q: *How to generate pure random string*?
A: *Put a fresh student in front of vi editor and ask him to quit.*
— Emiliano Lourbet (@taitooz)

Discussion

To perform computer-based simulation we need to be able to generate random numbers. Generating random numbers following a uniform distribution are the easiest to generate and are what comes out of the standard programming language "give me a random number" function. Here's a sample Python session:

```
>>> import random
>>> print random.random()
0.0818355341735
>>> print random.random()
0.660487346254
>>> print random.random()
0.365559490915
```

We could generate real random numbers by accessing, for example, noise on the ethernet network device but that noise might not be uniformly distributed. We typically generate pseudorandom numbers that aren't really random but look like they are. From Ross' *Simulation* book, we see a very easy recursive mechanism (recurrence relation) that generates values in $[0, m-1]$:

$x_n = ax_{n-1}$ modulo m

That is recursive (or iterative and not *closed form*) because x_n is a function of a prior value:

$x_1 = ax_0$ modulo m
$x_2 = ax_1$ modulo m
$x_3 = ax_2$ modulo m
$x_4 = ax_3$ modulo m
...

To get random numbers between 0 and 1, we use x_n/m.

We must pick a value for a and m that make x_n seem random. Ross suggests choosing a large prime number for m that fits in our integer word size, e.g., $m = 2^{31} - 1$, and $a = 7^5 = 16807$.

Initially we set a value for x_0, called the *random seed* (it is not the first random number). Every seed leads to a different sequence of pseudorandom numbers. (In Python, you can set the seed of the standard library by using `random.seed([x])`.)

Your goal is to take that simple recursive formula and use it to generate uniform random numbers. Please add the following functions file to the same `stats/stats.py` file from previous exercises:

```
# U(0,1)
def runif01():
```

```
    ...

# U(a,b)
def runif(a,b):
    ...

def setseed(s): # updates the seed global variable
    ...
```

Functions `runif01()` and `runif()` return a new random value per call. Use $m = 2^{31} - 1$, $a = 7^5 = 16807$, and an initial seed of $x_0 = 666$. *You may not use random.random() or any other built-in random number generators for this project. Obviously.*

Use `test_runif.py` to test values.

Deliverables. Make sure that you update `stats/stats.py` (the functions inside should emit no output at all, just return data as specified) and it is correctly committed to your repository and pushed to github.

Generating Binomial Random variables

Goal

The goal of this lab is to simulate a binomial distribution using repeated Bernoulli trials and then compare it against the theoretical binomial distribution. Please add new function `rbinomial()` to the existing `stats/stats.py` file so that we can continue to build our library. To draw the graphs in this exercise, create file `stats/plot_rbinomial.py`.

Discussion

1. First, define a function in file `stats/stats.py` that performs n Bernoulli trials with probability p of success. It should return the number of successes out of n:

```
def binomial(n,p):
    "Sim with prob p, n bernoulli trials; return number of successes"
    ...
```

The code is just a loop that goes around n times and uses a variable from $U(0,1)$, using your `runif01()` function, to check for success or failure. For example, my solution assumes failure if the uniform random variable is greater than p.

2. Next, in `stats/plot_rbinomial.py`, import your uniform random number generator library (which include your new `binomial()` function) and set the seed of the random number generator. (Otherwise you will always get the same Bernoulli trials.)

```
from stats import *

# other stuff we need:
import time
import matplotlib.pyplot as plt
import scipy.misc as misc

# I defined a function to hide implementation details
setseed( int(round(time.time() * 1000)) )
```

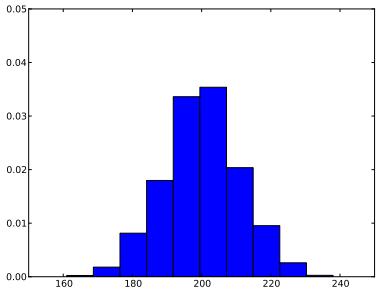
In this case, we're using the current time in milliseconds as the random seed so that it is different every time you run the program. (remember this trick.)

3. Now that we can know how to get a binomial random variable, we can examine the binomial distribution. All we have to do is grab a vector of, say, $SAMPLES$ binomial random values and the plot a histogram. The density function at k is just how many successes out of $SAMPLES$ there were ($k/SAMPLES$ probability).

Let me introduce you to something called a *list comprehension* in Python, which is a for loop that results in a list. It's also considered a *map* function ala *map-reduce*. Get list X as $SAMPLES$ binomial values with parameters $n = 500$ and $p = 0.4$. Do that by simply calling the `binomial` function N times.

```
X = [binomial(n,p) for t in range(SAMPLES)]
```

4. Plot the histogram normalized (normed=1) and run it. (You'll need `hist()`.) You should see a graph similar to the following:



5. We could use the built-in binomial mass function but let's define our own since it's easy:

$$\binom{n}{k} p^k (1-p)^{n-k} \qquad \text{(Binomial mass function)}$$

That's the probability that there are k successes in n trials with probability p of success. In `stats/stats.py`, define a function like this:

```
def binom(n, k, p):
    """
    If we run n trials with p prob for each trial of success,
    what is probability of having k successes? You can use scipy.misc.comb() if you want.
    """
    ...
```

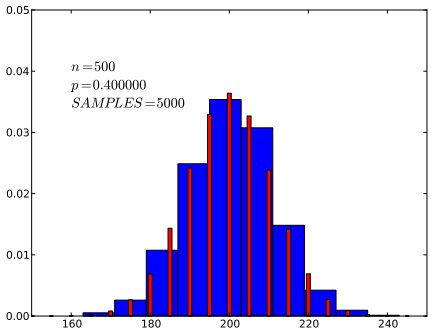
You may use function `scipy.misc.comb()` to compute $\binom{n}{k}$, but otherwise do the arithmetic yourself. (There is no loop in this function.)

6. To show the real distribution on top, we need to iterate k across the range $0..n$ used in our empirical simulation above. Since this is a mass function not a smooth density function, we can use every fifth value in the range. Let's also add some text to describe the parameters.

```
Y = [binom(n, k, p) for k in range(0,n+1,5)]
plt.bar(range(0,n+1,5), Y, color='red', align='center', width=1)
plt.axis([150,250,0,.05]) # set the axes so that we get a close-up
plt.text(160,0.04, '$n = %d$' % N, fontsize=16)
plt.text(160,0.037, '$p = %f$' % P, fontsize=16)
plt.text(160,0.034, '$SAMPLES = %d$' % SAMPLES, fontsize=16)
```

In this case I am not using 0..1 for the axes coordinates of the text; the default is the values of the graph itself. sometimes this is useful.

7. Run it and you should see something like the following:



Note that we use a bar chart for the binomial theoretical distribution and not a smooth graph because this is a mass function not a density function.

Deliverables. Make sure that `stats/stats.py` has the new functions and that `stats/plot_rbinomial.py` ($n = 500, p = 0.4, SAMPLES = 5000$) is correctly committed to your repository and pushed to github.

Generating Exponential Random Variables

Discussion

The goal of this lab is to generate random values from the exponential distribution using the *inverse transform method*. You will show the histogram of the random values and then show the theoretical exponential distribution on top to verify your results. You will reuse your exponential distribution random variable generator for the central limit theorem lab later. Use filename `stats/plot_rexp.py` for the plotting code, but you will stick your exponential functions into the usual `stats/stats.py` file.

Steps

1. First, create a function called `rexp(lambduh)` in `stats/stats.py` that returns a random value from the exponential distribution using the inverse transform method. To do that, you need the inverse *cumulative distribution function* (CDF) for the exponential distribution $Exp(\lambda)$. The *probability density function* for the exponential distribution is:

$$p = F(x;\lambda) = \lambda e^{-\lambda x}$$

Therefore the inverse function to get the x value associated with a probability p, we use

$$x = F^{-1}(p;\lambda) = -\frac{ln(1-p)}{\lambda}$$

Your function should look like the following:

```
def rexp(lambduh): # lambduh mispelled to avoid clash with lambda in python
    # u = get value from U(0,1) then
    # return F^-1(u) for exp cdf F^-1
```

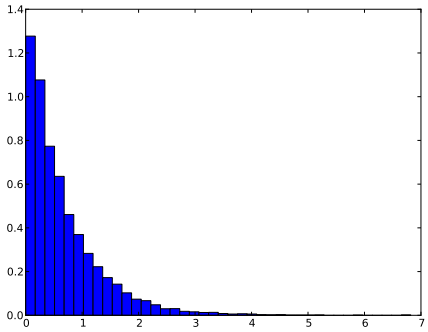
Use your `runif01()` function from previous labs.

2. To plot things, create file `stats/plot_rexp.py` and get a sample of exponential random variables into variable `X` of size N from $Exp(1.5)$ using your `rexp()`. I usually define constants to make the code more readable:

```
N = 1000
LAMBDUH = 1.5
```

then I can call `rexp(LAMBDUH)` and change LAMBDUH everywhere in my code by just changing the constant. In this case, there's no real need but it's good practice.

3. Plot a histogram of you are sample with `bins=40, normed=1`. You should see something like this:



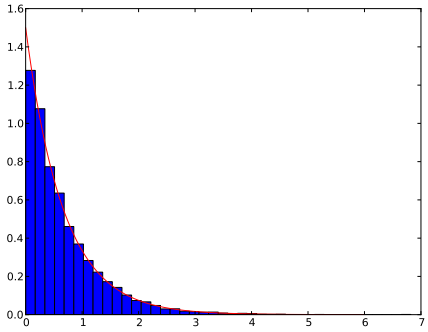
How do we know that this accurately represents the exponential distribution? We plot the theoretical distribution on top with a red line.

4. Since it's easy, let's define our own exponential probability density function in `stats/stats.py` as follows:

```
def exppdf(x, lambduh):
    ...
```

When you call it, make sure use the same lambduh.

5. Now, before the call to function `show()`, plot the theoretical distribution so that we can see both at once:



Deliverables. Make sure that `stats/stats.py` has the new functions and that `stats/plot_rexpr.py` is correctly committed to your repository and pushed to github.

The Central Limit Theorem in Action

Discussion

The goal of this lab is to observe how the sample means of uniform and exponential random variables have normal distributions with $N(\mu, \sigma^2/n)$ where σ^2 is the variance of the underlying distribution and n is the sample size whose mean we compute. Use filenames `plot_clt_unif.py`, `plot_clt_exp.py` for this lab.

Discussion

The CLT in a nutshell says that the sample mean, $X\_$, of samples X of size n from lots of distributions follows the normal distribution,specifically, $N(\mu, \sigma^2/n)$ for sample size n. In this lab will use $U(0,1)$ and the exponential distribution with $\lambda = 1.5$ and verify that using the mean as a random variable, the histogram shows a normal distribution of $N(0.5, 1/12)$ for the uniform and $N(\lambda^{-1}, \lambda^{-2}/n)$ for the exponential distribution. The mean of the uniform distribution is $\frac{a+b}{2}$ and the variance is $\frac{(b-a)^2}{12}$. The mean of the exponential distribution is $\mu = \lambda^{-1}$ and its variance is $\sigma^2 = \lambda^{-2}/n$.

The key thing here is to note that not only is the distribution of the mean random variable normal, but its variance gets tighter as we increase the sample size.

The law of large numbers says that the average of a large number of trials should approach the theoretical mean. That means that our sample mean:

$$\bar{X} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

will converge to μ as n approaches infinity with probability 1.

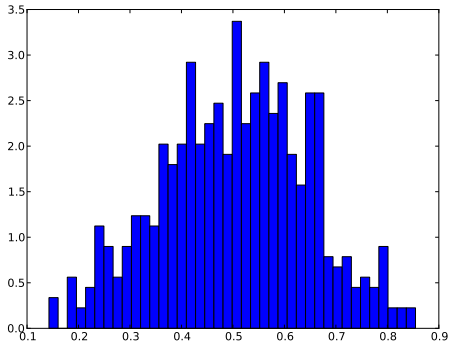
Also note that the number of trials we do improves the resolution of our normal distribution but doesn't change the variance.

CLT applied to uniform random variables

Steps

1. Get 500 samples X of size $LEN(X) = 4$ from the uniform distribution $U(0,1)$ using your `runif01()` function; i.e., $N = 4$ and $TRIALS = 500$. Compute the mean of each X vector and add it to the end of a different array $X\_$.

2. Plot the histogram of `X_` with `bins=40, normed=1`. Your histogram should look like this



Cool. It looks kind of like a normal distribution to me.

3. Let's add the theoretical normal distribution on top. To do that we need the appropriate parameters of $Normal(\mu, \sigma^2/n)$. The mean μ of uniform samples should be midway between a and b from $U(a,b)$. In our case, that's 0.5 since we are doing $U(0,1)$. The variance of the uniform distribution is $(b-a)^2/12$ and we need the variance divided by sample size N. In case we need it later, define a function in `stats/stats.py` that returns the variance of uniform distribution $U(a,b)$:

```
def unifvar(a, b):
    ...
```

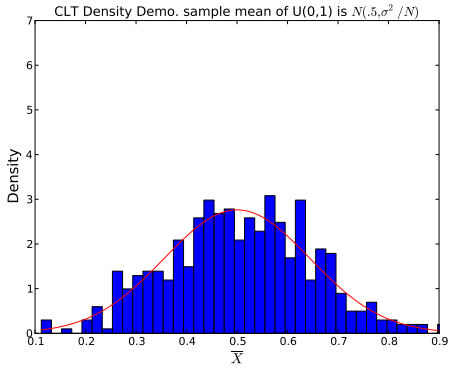
4. To get the theoretical distribution, let's define it ourselves in `stats/stats.py`:

```
def normpdf(x, mu, sigma): # sigma is the standard deviation, sigma^2 is the variance
    """
    Accept either a floating-point number or a numpy ndarray, such as what you get
    from arange().  You do not need a loop in the code does not change here
    because 2 * ndarray is another ndarray automatically. In this respect,
    numpy is very convenient and behaves like R.
    """
    ...
```

The function in math notation is:

$$P(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2 / 2\sigma^2}$$

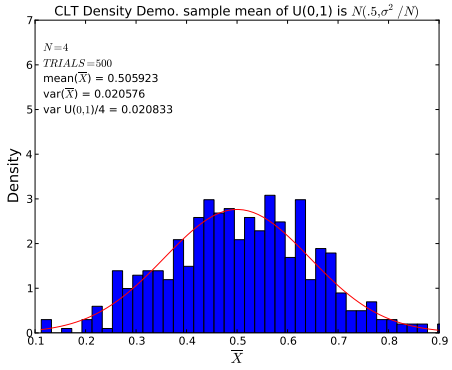
5. Then, plot the theoretical normal distribution on top of the histogram and set the axes so that we can use the same range throughout the next series of tests to see how the distribution changes. Note that the usual normal density function provided above expects the **standard deviation not the variance** and so we need to pass `normpdf()` the square root of the expected variance. The resulting graph should look like this



6. Now, display some important parameters in the graph using `text()`. You will need to do that `fig.add_subplot(111)` thing again early in your script. The text in between the $ symbols is latex and lets us display nice math symbols (e.g., the title), although I'm not doing much with it here.

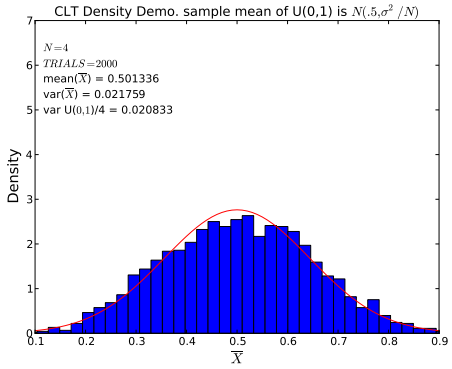
```
plt.text(.02,.9, '$N = %d$' % N, transform = ax.transAxes)
plt.text(.02,.85,'$TRIALS = %d$' % TRIALS, transform = ax.transAxes)
plt.text(.02,.8, 'mean($\\overline{X}$) = %f' % np.mean(X_), transform = ax.transAxes)
plt.text(.02,.75,'var($\\overline{X}$) = %f' % np.var(X_), transform = ax.transAxes)
plt.text(.02,.7, 'var U($0,1$)/%d = %f' % (N,varunif(0,1)/N), transform = ax.transAxes)

plt.title("CLT Density Demo. sample mean of U(0,1) is $N(.5, \sigma^2/N)$")
plt.xlabel("$\\overline{X}$", fontsize=16)
plt.ylabel("Density", fontsize=16)
```

7. Run it. The resulting graph should look like this

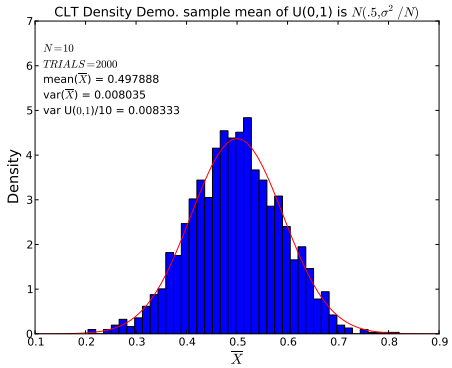


Notice how the mean is close to the expected 0.5 and that the variance of the sample mean is close to the theoretical variance.

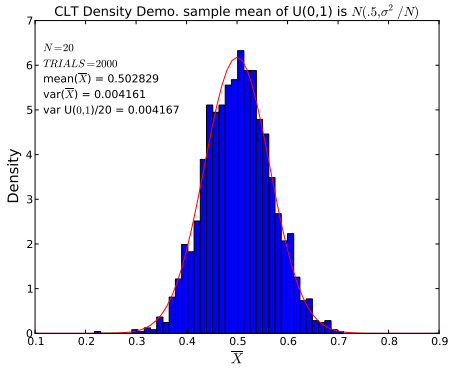
8. Increasing the number of trials two 2000 shows much higher resolution but does not change the variance/tightness of the distribution at all. Run it and see the following:



9. Now, if we increase the sample size to $N = 10$, we get a much tighter variance on the mean of $\overline{X}$. Run it:



10. Increasing to 20 we get:



CLT applied to exponential random variables

Now let's look at how the central limit theorem still gives us a normal distribution even when we pull random variables from a skewed distribution like the exponential. Create and edit a new file `plot_clt_exp.py`.

Steps

11. Make sure the `rexp(lambduh)` function you wrote for the previous lab to get exponential random variables is available via `import` and start out with the following constants:

```
N = 10
TRIALS = 4000
LAMBDUH = 1.5
```

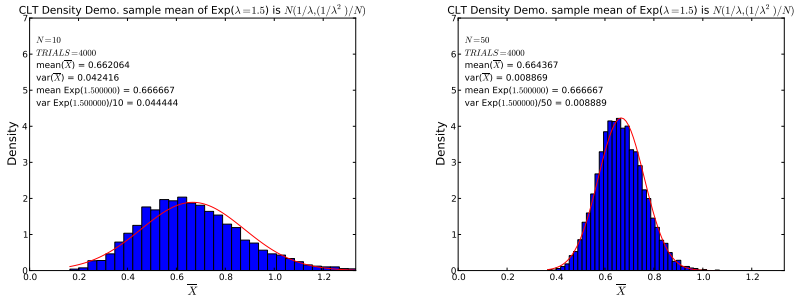
12. Repeat the loop we did above to get the mean of a bunch of samples into $X\_$, but this time from the exponential distribution `rexp(LAMBDUH)` instead of the uniform distribution function. Plot the histogram of $X\_$ as you did before, using a bin size of 40.

13. Plot the theoretical normal distribution on top using your `normpdf()`. The mean of the exponential distribution is $\mu = \lambda^{-1}$ and its variance is $\sigma^2 = \lambda^{-2}$.

14. Here are the appropriate text annotations:

```
plt.text(.02,.9, '$N = %d$' % N, transform = ax.transAxes)
plt.text(.02,.85, '$TRIALS = %d$' % TRIALS, transform = ax.transAxes)
plt.text(.02,.8,   'mean($\\overline{X}$) = %f' % np.mean(X_), transform = ax.transAxes)
plt.text(.02,.75, 'var($\\overline{X}$) = %f' % np.var(X_), transform = ax.transAxes)
plt.text(.02,.7, 'mean Exp($%f$) = %f' % (LAMBDUH,1/LAMBDUH), transform = ax.transAxes)
plt.text(.02,.65, 'var Exp($%f$)/%d = %f' % (LAMBDUH,N,(1/LAMBDUH**2)/N), transform = ax.transAxes)

plt.title("CLT Density Demo. sample mean of Exp($\lambda=1.5$) is $N(1/\lambda, (1/\lambda^2)/N)$")
plt.xlabel("$\\overline{X}$", fontsize=16)
plt.ylabel("Density", fontsize=16)
plt.axis([0,1.333,0,5])
plt.savefig('clt_exp-'+str(TRIALS)+'-'+str(N)+'.pdf', format="pdf")
```

15. Run it and you should see the following two graphs according to the value of N:



Notice that there is a slight leftward bias in that the normal distribution is a little bit to the right it looks like. This is to be expected. You really need to bump up N before you see it converge to the proper alignment.

16. Play around with other values of lambda and N.

Deliverables. Make sure that you update `stats/stats.py` and `clt_unif.py`, `clt_exp.py` are correctly committed to your repository and pushed to github. Also commit PDFs created via `savefig()` above for:

- $N = 20$, $TRIALS = 2000$ for CLT $U(0, 1)$
- $N = 50$, $TRIALS = 4000$, $\lambda = 1.5$ for CLT $Exp(\lambda)$

Generating Normal Random Variables

Discussion

The goal of this lab is to generate normal random variables but using the Central limit theorem instead of the inverse transform or the accept reject method. I'm not recommending this as the most efficient method, but it is a great practical application of the central limit theorem. The hard part about all of this is using the right variance and shifting from $N(0,1)$ to the general $N(\mu,\sigma^2)$. Use filename `stats/plot_rnorm.py`.

Steps

1. First, let's define some constants $N = 100$, $TRIALS = 4000$.

2. Now, to define a function that generates normal random variables in $N(0,1)$, we rely on the fact that the sample mean, $\overline{X}$ from a sample X of uniform distribution values, is a normal random variable. This gives us as many normal random values as we want, one per sample X. We just have to tweak things so that the mean of the distribution is zero-centered and has variance 1. That shifted and scaled value is what we return from `rnorm01()` that you must create in `stats/stats.py`:

```
def rnorm01():
    "return a value from N(0,1)"
    ...
```

The process looks like this:

A. Get N uniform random values from $U(0,1)$ into X using your `runif01()` function.

B. Then compute the mean $\overline{X}$.

C. Shift that value so that is zero-centered and call it rv.

D. We know from the CLT lab that the variance of random variable $\overline{X}$ is σ^2/N, where σ^2 is the variance of the underlying distribution $U(0,1)$, but we need the variance to be 1. Scale rv so that it has variance 1. Note that a "standard normal" variable can be created from an arbitrary normal X via $Z = (X - \mu)/\sigma$. Z is effectively a shifted and scaled version of the original. Interestingly, it really just measures how many standard deviations X is from $N(0,1)$.

3. Now, let's fill in the code we need to draw a histogram and the theoretical distribution on top using the `normpdf()` from the CLT labs:

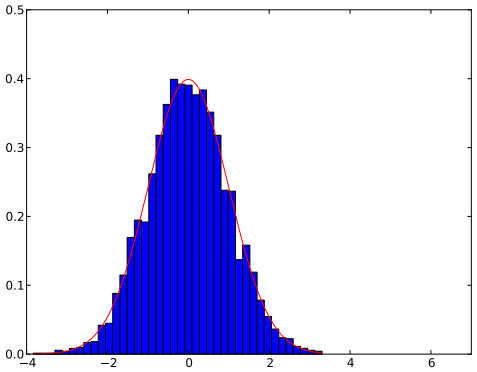
```
# Get X taken from TRIALS trials, plot histogram normalized to density func
X = ...
plt.axis([-4, 7, 0, 0.5]) # let's keep the same access across plot for this lab
plt.hist(X, bins=40, normed=1) # histogram should look standard normal
```

4. Plot the real normal curve on top like we did in the CLT lab.

5. Save a PDF:

```
plt.savefig('rnorm01-%d-%d.pdf' % (TRIALS,N), format="pdf")
plt.show()
```

which should look like:



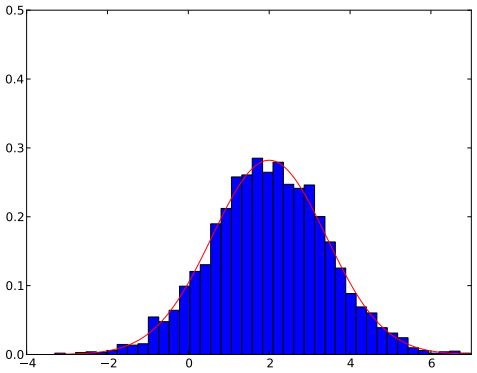
6. Now define a more general method in `stats/stats.py` that accepts a desired mean and variance (*not the mean and the standard deviation*):

```
def rnorm(mean, variance):
    "return a value from N(mean,variance)"
    ...
```

We know how to get a standard normal random variable, Z, as we just defined `rnorm01()`. To get a normal random variable with different mean and variance, we reverse the process we used to get a standard normal via $Z = (X - \mu)/\sigma$. Dust off your high school algebra and solve for X. That tells you how to shift and scale properly: $X = \mu + Z\sigma$.

7. And test as before but this time use $\mu = 2$ and $\sigma^2 = 2$. Save a PDF

```
plt.savefig('rnorm-%d-%d-%d-%d.pdf' % (MEAN,VARIANCE,TRIALS,N), format="pdf")
```



Deliverables. Make sure that you update `stats/stats.py` and `plot_rnorm.py` is correctly committed to your repository and pushed to github. Also commit PDFs of the graphs shown above for $N(0,1)$ and $N(\mu = 2, \sigma^2 = 2)$.

Confidence Intervals for Price of Hostess Twinkies

Goal

The goal of this lab is to learn how to compute an empirical 95% confidence interval for the sample mean price for Hostess Twinkies using an awesome technique called *bootstrapping*. Bootstrapping allows us to assess the accuracy of a sample mean we've computed, giving us data about how sample means vary. As part of this lab, you will also learn to read in a file full of numbers. In this case, we are going to read in the price of Hostess Twinkies, a tasty snack recently returned from the dead, from around the US.

Discussion

A sample mean confidence interval of 95% tells us the range in which most (95% or 1.96σ) of the sample means fall. All we have to do is create a number of samples, X, and compute the means $\overline{X}$. If we do this lots of times (trials) then 95% of the time, we would expect the sample mean to fall within the range of 95% of the samples. We just have to order the $\overline{X}$ values and strip off the lower and top 2.5%. Then, the lowest and highest value in that stripped list represent the boundaries of the confidence interval. Cool, right?

From the central limit theorem, we know that the distribution of $\overline{X}$ is $N(\overline{X}, \sigma^2/n)$ for sample size n (not the number of trials). In this case, however, we don't know what the underlying distribution is because we just got a bunch of prices from a file. We could assume that it's normally distributed, but there's no point. The central limit theorem works on any underlying distribution we care about here but we do need the variance. For that, we can use the sample variance as an estimate of the variation in the overall Hostess Twinkies price population.

The question is how do we get lots of trials from an underlying distribution that we cannot identify? By repeated sampling from our single sample *with replacement*. This is called *bootstrapping*, which you could also call *resampling*. The idea is to randomly select N values from our known data set of size N. That gives us one trial. We can then repeatedly compute our test statistic, the mean, on each sample.

To verify that we are doing the right thing, we will draw the theoretical normal distribution expected by the Central limit theorem and then shade in the 95% theoretical confidence interval, which we know is 1.96 standard deviations on either side of the mean: $\mu \pm 1.96\sigma$.

Please do your work in filename `stats/conf.py`.

Steps

1. First, we have to get the data from a file called `prices.txt` from `https://github.com/parrt/msan501/tree/master/data`:

```
prices = []
f = open("prices.txt")
for line in f:
```

```
    v = float(line.strip())
    prices.append(v)
```

When debugging or during development, you can print those numbers out to verify they look okay.

2. Now, we need a function that lets us sample *with replacement* from that raw data set. In other words, we need a function that gets n values at random from a data parameter (a list of numbers). It should allow repeated grabbing of the same value (that's what we call with replacement).

```
def sample(data):
    """
    Return a random sample of data values with replacement.
    The returned array has same length as data.
    """
```

The idea is to get an array of random numbers from $U(0, n)$ for `n=len(data)`. These then are a set of indices into the data array so just loop through this index array grabbing values from data according to the index. For example if you have indexes = [3,9] for a 2-element data array, then return a new array [data[3], data[9]. My solution has two lines in it.

3. Now define TRIALS=20 and perform that many samplings of prices. For each sample, create the sample mean and add it to an X_ list.

4. Sort that list and get the values from indices TRIALS*0.025..TRIALS*0.975 in X_ and call it `inside`.

5. Print the first and last value of the `inside` array as that will tell you what the bounds of your 95% confidence interval are

```
print inside[0], inside[-1]
```

You might get something like (there will be a lot of variation):

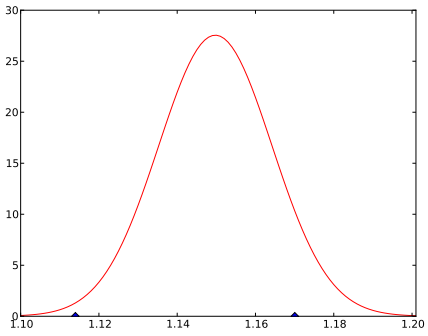
```
1.12295362319 1.16113333333
```

6. Add code to plot diamonds on the graph at those locations.

7. Now plot the normal curve using your amazing new understanding of the central limit theorem. Use the following range and also set the overall graph range:

```
x = np.arange(1.05, 1.25, 0.001)
plt.axis([1.10, 1.201, 0, 30])
```

8. Run it and you should get the following graph:

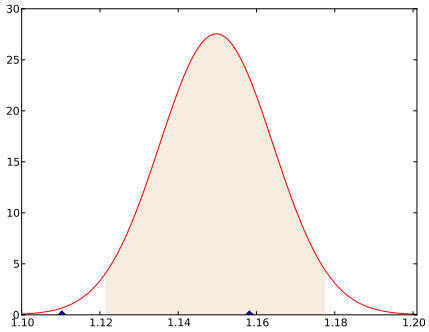


Ok, that's great but we have no idea if this is correct or not. Now, let's go nuts and show lots of stuff on the graph.

9. First, let's shade in the theoretical 95% confidence interval using your `normpdf()`.

```
mean = ...
stddev = ...
# redraw normal but only shade in 95% CI
left  = ...
right = ...

ci_x = np.arange(left, right, 0.001)
ci_y = normpdf(ci_x,mean,stddev)
# shade under (ci_x,ci_y) curve
plt.fill_between(ci_x,ci_y,color="#F8ECE0")
```

Run it again to see how it shades in the graph.

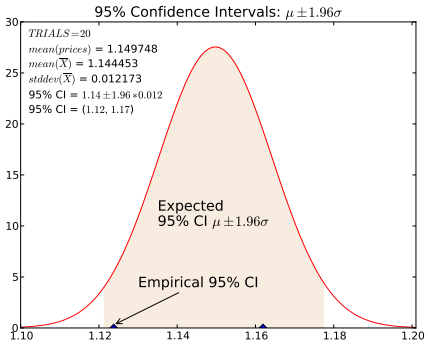


10. Now let's annotate with lots of information. Please read through and figure out what all of that stuff does to draw the nice arrows and so on.

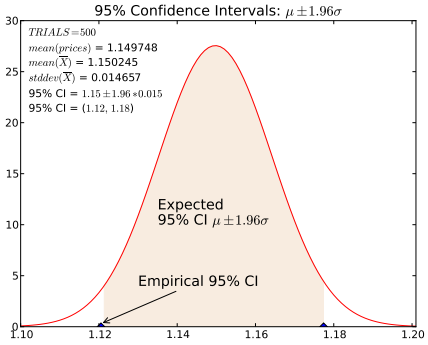
```
plt.text(.02,.95, '$TRIALS = %d$' % TRIALS, transform = ax.transAxes)
plt.text(.02,.9,  '$mean(prices)$ = %f' % np.mean(prices), transform = ax.transAxes)
plt.text(.02,.85, '$mean(\\overline{X})$ = %f' % np.mean(X_), transform = ax.transAxes)
plt.text(.02,.80, '$stddev(\\overline{X})$ = %f' %
    np.std(X_,ddof=1), transform = ax.transAxes)
plt.text(.02,.75, '95%% CI = $%1.2f \\pm 1.96*%1.3f$' %
   (np.mean(X_),np.std(X_,ddof=1)), transform = ax.transAxes)
plt.text(.02,.70, '95%% CI = ($%1.2f,\\ %1.2f$)' %
                  (np.mean(X_)-1.96*np.std(X_),
                   np.mean(X_)+1.96*np.std(X_)),
         transform = ax.transAxes)

plt.text(1.135, 11.5, "Expected", fontsize=16)
plt.text(1.135, 10, "95% CI $\\mu \\pm 1.96\\sigma$", fontsize=16)
plt.title("95% Confidence Intervals: $\\mu \\pm 1.96\\sigma$", fontsize=16)
```

```
ax.annotate("Empirical 95% CI",
            xy=(inside[0], .3),
            xycoords="data",
            xytext=(1.13,4), textcoords='data',
            arrowprops=dict(arrowstyle="->",
                            connectionstyle="arc3"),
            fontsize=16)
```

11. Run it and you should get the following graph:



12. We don't have to increase the number of trials very much before the confidence interval tightens up nicely. Try 500:



Deliverables. `stats/conf.py` and a PDF called `conf-500.pdf`, both with $TRIALS = 500$.

Is Free Beer Good For Tips?

Goal

The goal of this lab is to test a hypothesis using a variety of techniques: "eyeball" test, t-test, and bootstrapping. Use filename `stats/hyp.py`.

Discussion

Here is a typical statistics question (derived from one by Jeff "The Hammer" Hamrick) that we will solve in multiple ways.

Q. *Psychologists studied the size of the tip in a restaurant when the waitron gave the patron a free beer. Here are tips from 20 patrons, measured in percent of the total bill: 20.8, 18.7, 19.1, 20.6, 21.9, 20.4, 22.8, 21.9, 21.2, 20.3, 21.9, 18.3, 21.0, 20.3, 19.2, 20.2, 21.1, 22.1, 21.0, and 21.7. Does a beer-inspired tip exceed 20 percent or perhaps dip below 20 percent (maybe patrons get drunk and can't do math)? Use a significance level equal to $\alpha = 0.06$.*

Side note: Always pick the significance level before you run your experiment. It is really bad mojo to pick your significance after you know what the p-value is.

Before starting on this, let's interpret that question: It asks whether the mean of the specified sample differs significantly from the usual 20% tip. By "significantly" we refer to the likelihood that the usual population (with mean 20.0) could yield a sample with the observed sample mean. By "usual" we mean our control of approximately: $N(20.0, s^2/n)$ where s is the sample variance of the sample tips and $n = len(tips)$. (We can reasonably assume that tips follow a normal distribution.)

While the population mean is 20.0, the means of any resamples we take will bounce around left and right of 20.0. The question is: does this particular test sample's mean, $m = 20.725$, fall outside of the typical variability of the sample means?

More formally, we would say the following: The **null hypothesis** is that the mean for the specified sample does not differ significantly from $\mu = 20.0$. I think of this as the *control* in my experiment. The **alternate hypothesis** is that the sample mean differs significantly above or below the population mean. Formally,

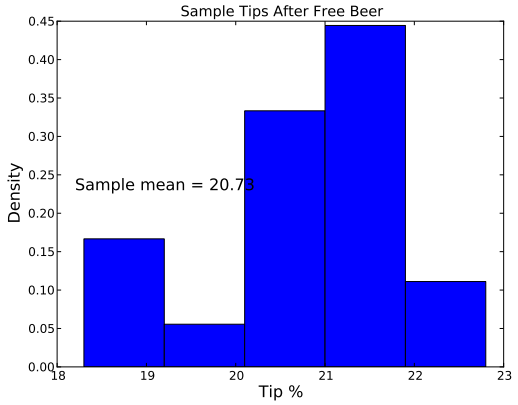
$H_0 : m = 20.0$ (non-free beer situation)
$H_1 : m \neq 20.0$ (free beer situation; two-sided alternative hypothesis)

We could also say that $H_0 : m - \mu = 0$ and $H_1 : |m - \mu| > 0$.

Steps

Eyeballing it

1. First, just draw a histogram of the tips to see what it looks like. For this exercise, create a file called `stats/hyp.py`.

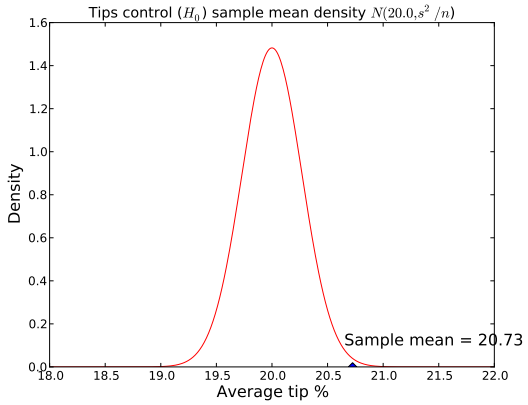


For your convenience, here are the tips in python format:

```
tips = [20.8, 18.7, 19.1, 20.6, 21.9, 20.4, 22.8,
        21.9, 21.2, 20.3, 21.9, 18.3, 21.0, 20.3,
        19.2, 20.2, 21.1, 22.1, 21.0, 21.7]
```

(Use your awesome new skills from previous labs to generate the histogram.) To me, there is a lot of "mass" to the right of the usual 20% tip but my eyeball is not a rigorous significance mechanism.

2. To get a better idea, let's simply plot the distribution of the sample means given our H_0 assumption: $N(20.0, s^2/n)$. We need to use the sample variance s^2 from our test sample because we don't know the variance of the original distribution. It safe to assume that the variance is similar. This is our "control" or the usual tipping distribution: the distribution of the set of average tips per day if H_0, the control, is true.



Looking at that graph, it seems that a sample mean of 20.73 is pretty far in the right tail of a normal curve centered at the control average 20% tip. It looks to be a few standard deviations away from the mean. My gut says that it's pretty likely that giving people a free beer increases tips significantly.

t-test

1. Let's use a *t-test* now to test for significance, just like we would do in statistics class. The t value measures the number of standard deviations a sample mean, m, is away from our presumed population mean μ:

$$t = \frac{m - \mu}{s/\sqrt{N}} \quad \text{(t-value)}$$

It's just the difference between the means scaled to be in units of standard deviations. Write some code to compute the t-value. When computing s, the sample standard deviation, note that the numpy std() function returns a biased estimate of the standard deviation. Use `np.std(tips, ddof=1)` instead of just `std(tips)`.

2. Print out the value of t. I get $t = 2.69417199392$. That means that m is about 2.7 standard deviations away from μ, which is a very significant departure.

3. To get a p-value, likelihood that we would see such a t value in the nonfree beer situation, look up t in a t-distribution c.d.f. using `1-scipy.stats.t.cdf(t,N-1)`. You should get 0.0071844. Since we need to check both tails, the probability is actually $2x$ that, or, p-value=0.014369 (1.4%). The definition of significance is $\alpha = 0.06$, which means that our sample mean is definitely significant since $1.4\% < 6\%$. There is only a 1.4% chance that the control could generate a value that extreme or beyond.

We must conclude that m differs significantly from $\mu = 20.0$ based upon the significance of $\alpha = 0.06$ and, therefore, we reject H_0 in favor of H_1. Giving out free beers is extremely likely to have increased the average tip in that experiment.

Boostrapping for empirical hypothesis testing

Ok, now, let's use bootstrapping to estimate a *p-value*. A p-value for some point statistic or value is the probability that the control (null hypothesis H_0) could generate that statistic or value. In our case, a p-value can tell us the likelihood that a normal distribution centered around $\mu = 20.0$ with $s^2 = var(tips)$ could generate a sample mean of 20.725. (We approximate the population variance with our sample variance.) *Note and we are sampling from $N(\mu = 20.0, s^2)$ to conjure up samples from the control situation. We are not resampling from the tips list as we are trying to see how the observed sample mean, 20.725, fits within the control distribution not the test distribution. We are also not generating samples from the distribution of a mean random variable, $N(\mu = 20.0, s^2/n)$.*

1. Bootstrap TRIALS=5000 samples of size $n = len(tips)$ from $N(\mu, s^2)$ using your `rnorm()` function created in a previous lab. It's very important that we use the same sample size as $len(tips)$ so we are comparing the same thing. Compute the mean of each sample, X, an add to $\overline{X}$ as you generate samples from the normal distribution.

2. Compute how many means in $\overline{X}$ are greater than or equal to mean(tips):

```
greater = np.sum(X_ >= np.mean(tips))
```

or

```
greater = sum([x>=np.mean(tips) for x in X_]) # the number of true values
```

3. The (one-sided) p-value is just the ratio of values above the observed mean, mean(tips), to the number of trials. Double that because we're doing a two-sided test. With 5000 trials, I see just 13 values greater than $m = 20.725$. That gives us a p-value of $2 * \frac{13}{5000} = 0.0052$ or .52%. That means that, empirically, we find that there is an extremely small probability that the control could generate an extreme value like $m = 20.725$. Certainly the likelihood is less than the required 6% significant value.

Note: we would expect the empirical p-value (.52%) and the p-value derived from the t-test (1.4%) to be very close to each other when the number of trials is large with bootstrapping. Our resident statistician, Jeff Hamrick, explains that the difference is not a problem with our bootstrapping solution and is ok.

"*A student t distribution with dof=19, is pretty close to a normal. But the differences are most greatly felt in the tails, and we're in the tails (rejection H_0), thus casting a little bit of sketchiness or your choice to draw the simulated raw data from a normal random variable. If we were performing this exact same operation on a data set with reasonably large size (say, 40 or 50 or 75) the differences would still exist but would be even more minute.*"

Again, we easily reject the control and conclude that giving out free beers increases tips.

Deliverables. `stats/hyp.py` and a text file call `stats/hyp_results.txt` that gives your t-value and p-value from the t-test. Also give your empirical p-value from bootstrapping with $TRIALS = 5000$. Tag when completed with all stats-related exercises with `stats`.

Part V

Optimization and Prediction

Iterative Optimization Via Gradient Descent

Goal

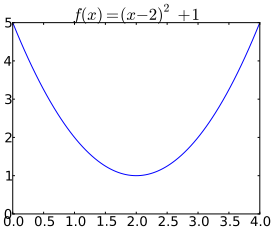
The goal of this task is to increase your programming skill by solving an iterative computation problem with nontrivial iteration and termination conditions: *gradient descent function minimization*. Please use file `opt/descent.py` for your `minimize()` function and `opt/plot_descent.py` for the code that draws the trace of the minimization in action (i.e., this is the file that has all the matplotlib junk).

Discussion

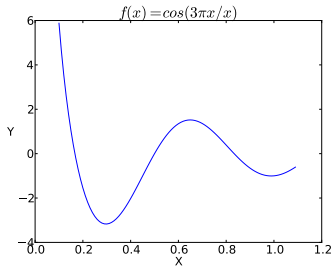
Finding x that minimizes function $f(x)$ (usually over some range) is an incredibly important operation as we use it to minimize risk and, for machine learning, to learn the parameters of our classifiers or predictors. Generally x will be a vector but we will assume x is a scalar to learn the basics. If we know that the function is convex like a quadratic polynomial, there is a unique solution and we can simply set the derivative equal to zero and solve for x:

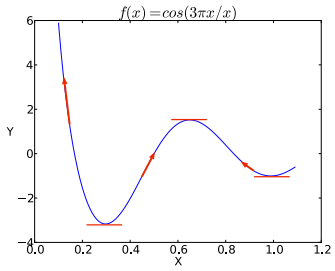
$$f'(x) = 0 \qquad \text{(Analytic solution to optimization)}$$

For example, the function $f(x) = (x-2)^2 + 1$ has $f'(x) = 2x - 4$ whose zero is $x = 2$.



We prefer to find the *global minimum* but generally have to be satisfied with a *local minimum*, which we hope is close to the global minimum. A decent approach to finding the global minimum is to find a number of local minima via random starting x_0 and just choose the minimum local minimum discovered. For example, the function $f(x) = cos(3\pi x)/x$ has two minima in $[0, 1.1]$, with one obvious global minimum:





If the function has lots of minima/maxima or is very complicated, there may be no easy analytic solution. There are many approaches to finding function minima iteratively (i.e., non-analytically), but we will

use a well-known technique called *gradient descent* or *method of steepest descent*.

Gradient descent

This technique can be used to train everything from *linear regression* models (see next lab) to *neural networks*. Gradient descent requires a starting position, x_0, the function to optimize, $f(x)$, and its derivative $f'(x)$. Recall that the derivative is just the slope of a function at a particular point. In other words, as x shifts away from a specific position, does y go up or down, and by how much? E.g., the derivative of x^2 is $2x$, which gives us a positive slope when $x > 0$ and a negative slope when $x < 0$. Gradient descent uses the derivative to iteratively pick a new value of x that gets us closer and closer to the minimum of $f(x)$. The negative of the derivative tells us the direction of the nearest minimum. For example, the graph to the right above shows a number of vectors representing derivatives at particular points. Note that the derivative is zero, i.e. flat, at the minima (same is true for maxima). The recurrence relation for updating our estimate of x that minimizes $f(x)$ is then just:

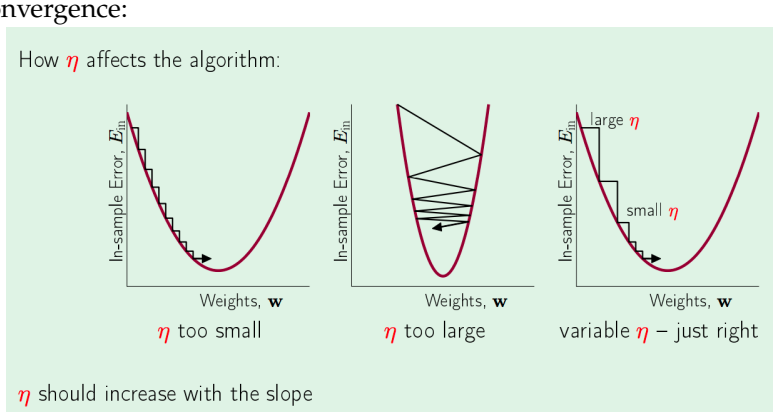
$$x_{i+1} = x_i - \eta f'(x_i)$$

where η is called the *learning rate*, which we'll discuss below. The $\eta f'(x_i)$ term represents the size of the step we take towards the minimum. The basic algorithm is:

1. Pick an initial x_0, let $x = x_0$
2. Let $x_{i+1} = x_i - \eta f'(x_i)$ until $f'(x_i) = 0$

That algorithm is extremely simple but knowing when to stop the algorithm is problematic when dealing with the finite precision of computers. Specifically, no two floating-point numbers are ever equal really. So $f'(x) = 0$ is always false. Usually we do something like $abs(x_{i+1} - x_i) < precision$ or when $abs(f(x_{i+1}) - f(x_i)) < precision$ where precision is some very small number like 0.0000001. Personally, I like the concept of stopping when there is a very small vertical change **and** $f(x_{i+1})$ is heading back up.

The steps we take are scaled by the learning rate η. Yaser S. Abu-Mostafa has some great slides and videos that you should check out. Here is his description on slide 21 of how the learning rate can affect convergence:



The domain of x also affects the learning rate magnitude. This is all a very complicated finicky business and those experienced in the field tell me it's very much an art picking the learning rate, starting positions, precision, and so on. You can start out with a low learning rate and crank it up to see if you still converge without oscillating around the minimum. An excellent description of gradient descent and other minimization techniques can be found in Numerical Recipes.

Approximating derivatives with finite differences

Sometimes, the derivative is hard, expensive, or impossible to find analytically (symbolically). For example, some functions are themselves iterative in nature or even simulations that must be optimized. There might be no closed form for $f(x)$. To get around this and to reduce the input requirements, we can approximate the derivative in the neighborhood of a particular x value. That way we can optimize any reasonably well behaved function (left and right continuity would be nice). Our minimizer then only requires a starting location and $f(x)$ but not $f'(x)$, which makes the lives of our users much simpler and our minimizer much more flexible.

To approximate the derivative, we can take several approaches. The simplest involves a comparison. Since we really just need a direction, all we have to do is compare the current $f(x_i)$ with values a small step, h, away in either direction: $f(x_i - h)$ and $f(x_i + h)$. If $f(x_i - h) < f(x_i)$, we should move x_{i+1} to the left of x_i. If $f(x_i + h) < f(x_i)$, we should move x_{i+1} to the right. This is called the forward difference but there is also backward difference and a central difference. The excellent article Stochastic Gradient Descent Tricks has a lot of practical information on computing gradients etc...

Using the direction of the slope works, but does not converge very fast. What we really want is to use the magnitude of the slope to make the algorithm go fast where it's steep and slow where it's shallow because it will be approaching a minima. So, rather than just using the sign of the finite difference, we should use the magnitude or rate of change. Using finite differences then, we get a similar formula but replace the derivative with the finite (forward) difference:

$$x_{i+1} = x_i - \eta \frac{f(x_i + h) - f(x_i)}{h} \text{ where } f'(x) \approx \frac{f(x_i + h) - f(x_i)}{h}$$

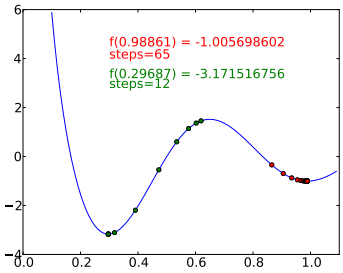
To simplify things, we can roll the step size h into the learning rate η constant as we are going to pick that anyway.

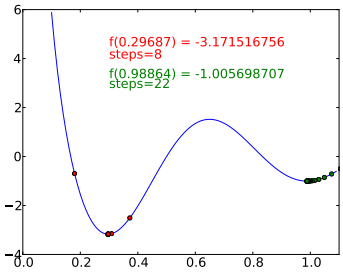
$$x_{i+1} = x_i - \eta(f(x_i + h) - f(x_i))$$

The step size is bigger when the slope is bigger and is smaller as we approach the minimum (since the region is flatter). Abu-Mostafa indicates in his slides that η should increase with the slope whereas we are keeping it fixed and allowing the finite difference to increase the step size. We are not normalizing the derivative/difference to a unit vector like he does (see his slides).

Your task

You will use gradient descent to minimize $f(x) = cos(3\pi x)/x$. To increase chances of finding the global minimum, pick **two** random locations in the range $[0.1, 1.2]$ using standard python `random.random()` and perform gradient descent with both of them. As part of your final submission, you must provide a plot of $f(x)$ with traces that indicate the steps taken by your gradient descent; use a different color for each descent. Here are two sample descents where the x and $f(x)$ values are displayed as well as the minimum of those two:





To create the dots you just need to add the x values to an array as you search for the minimum and then plot the x and $f(x)$ values with red or green dots. In your `opt/plot_descent.py` file, you'll use stuff like:

```
tracey = [f(x) for x in tracex]
plt.plot(tracex, tracey, 'ro') # plot red dots
```

Please show the information as I have shown in the graphs to make it easier to compare results and for me to grade.

Now, in your `opt/descent.py` file, define a function called `minimize` that takes the indicated parameters and returns a trace of all x values visited including the initial guess:

```
def minimize(f, x0, eta, h, precision):
    tracex = []
    tracex.append(x0)  # add starting position
    ...
    return tracex
```

Hide all of your plotting junk inside of `opt/plot_descent.py` file:

```
... code that uses minimize(), plotting ...
```

As an example, I call that function like this:

```
tracex = minimize(f, x0, ETA, STEP, PRECISION)
```

for an appropriate `f()` definition per the above cosine function. Note that Python allows us to pass a function just like any other object; we did this in our image `filter()` function. For parameter `f`, we can call that function from within `minimize()` with the usual syntax `f()`.

So that we all have the same graph structure, please use the following code (in `opt/plot_descent.py`) to plot the cosine function:

```
import matplotlib.pyplot as plt

graphx = np.arange(.1,1.1,0.01)
graphy = f(graphx)
plt.plot(graphx,graphy)
plt.axis([0,1.1,-4,6])
```

You will have to pick an appropriate step value h to get a decent approximation of the derivative through finite differences but that is large enough to avoid faulty results from lack of precision (subtracting two floating-point numbers in the computer results in a number with much less precision than the original

numbers). You want that number to be small enough so that your algorithm does not oscillate around the minimum. If the number is too big it will compute a finite difference that makes x_{i+1} leap across the minimum to the other wall of the function. You must pick a learning rate η that allows you to go as fast as you can but not so fast that it overruns the minimum back and forth. When I crank up my learning rate too far, I also see the algorithm oscillate:

```
...
f(0.491296576641) = -0.166774773584 , delta = 2.05763033375622805821
f(0.296744439739) = -3.171512867583 , delta = -3.00473809399913660556
f(0.297092626880) = -3.171512816769 , delta = 0.00000005081414267138
...
```

To help you understand what your program is doing, print out x, $f(x)$, and any other value you think is helpful to see how your program explores the curve. BUT, your code shouldn't print that out in your final submission.

To give you some idea about how fast your minimization function should converge my implementation seems to converge in less than 70 steps.

Testing

Test your code with the `opt/test_descent.py` program in the starterkit `opt` dir.

Deliverables.

- Your script `opt/descent.py` with the `minimize()` function.
- Your script `opt/plot_descent.py`.
- A PDF called `opt/traces.pdf` of your graph with two *visible* traces (sometimes they will overlap and you can't see one of them). It doesn't matter if they both are converging to the same minimum or two different ones. The graph should include the text I have on mine for x, $f(x)$, number of steps, etc...

Tag when completed with `descent`.

Predicting Murder Rates With Gradient Descent

Goal

The goal of this exercise is to extend the techniques you learned in the one-dimensional gradient descent task to a two-dimensional domain space, solving a *linear regression problem*. This problem is also known as *curve fitting*. As part of this lab, you will learn how to compute with vectors instead of scalars. Please use file `opt/regression.py` for function `opt/plot_regression.py` for the code that draws the trace of the minimization in action (i.e., this is the file that has all the matplotlib junk). You'll see starter (and test) files in the starterkit `opt` dir.

Discussion

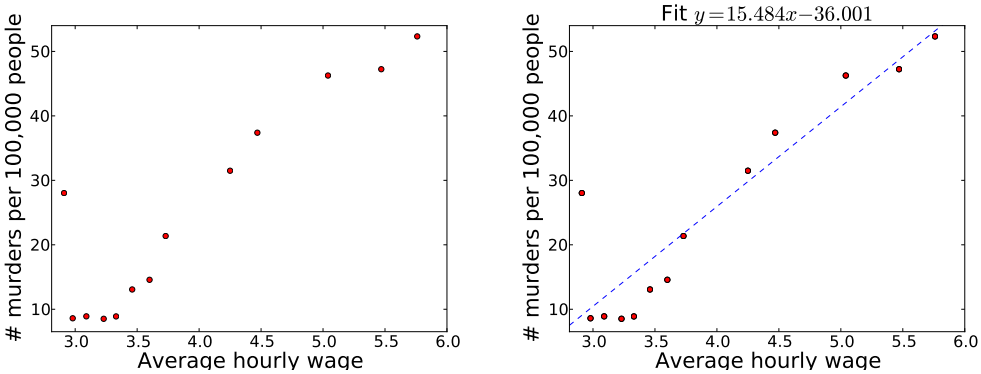
Problem statement

Given training data (x_i, y_i) for $i = 1..n$ samples with dependent variable y_i, we would like to predict y for some x's not in our training set. x_i is generally a vector of independent variables but we'll use a scalar. If we assume there is a linear relationship between x and y, then we can draw a line through the data and predict future values with that line function. To do that, we need to compute the two parameters of our model: a slope and a y intercept. (We will see the model below.)

For example, if we compare the number of murders per 100,000 people in Detroit to the average hourly wage, our eyeballs easily detect a correlation. Here is data suitable to copy and paste into Python:

```
HOURLY_WAGE = [2.98, 3.09, 3.23, 3.33, 3.46, 3.6, 3.73, 2.91, 4.25, 4.47, 5.04, 5.47, 5.76]
MURDERS = [8.6, 8.9, 8.52, 8.89, 13.07, 14.57, 21.36, 28.03, 31.49, 37.39, 46.26, 47.24, 52.33]
```

and here is a scatter plot and best fit line as determined by numpy (using `np.polyfit(HOURLY_WAGE, MURDERS,1)`).



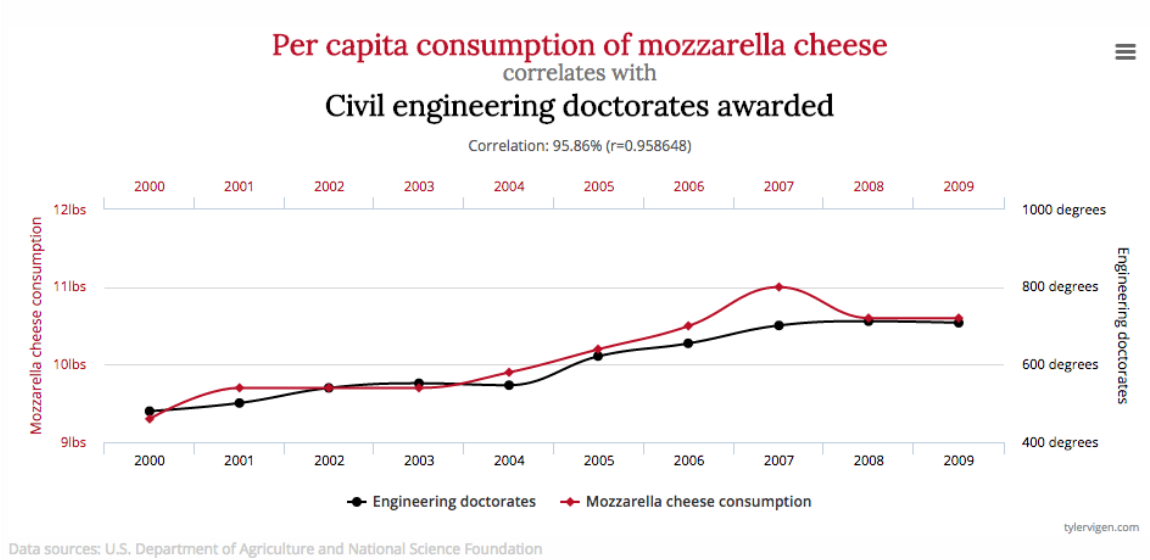
Here, for example, $x_0 = 2.98$ and $y_0 = 8.6$.

This might be a good point to remind everyone that correlation does not equal causation. I hardly think that paying people more makes them murderous, although I could see the opposite. ;) Correlation is a *necessary*

but not *sufficient* condition for causation. When you find a correlation, that gives you a candidate to check for cause-and-effect.

Here's another fun data set from `http://www.tylervigen.com/spurious-correlations`:

```
# http://www.census.gov/compendia/statab/2012/tables/12s0217.xls
MOZZ_CHEESE_PER_CAPITA_2000_2009 = [9.3, 9.7, 9.7, 9.7, 9.9, 10.2, 10.5, 11.0, 10.6, 10.6]
# http://www.nsf.gov/statistics/infbrief/nsf11305/
CIVIL_ENG_PHD_2000_2009          = [480, 501, 540, 552, 547, 622, 655, 701, 712, 708]
```



Best fit line that minimizes squared error

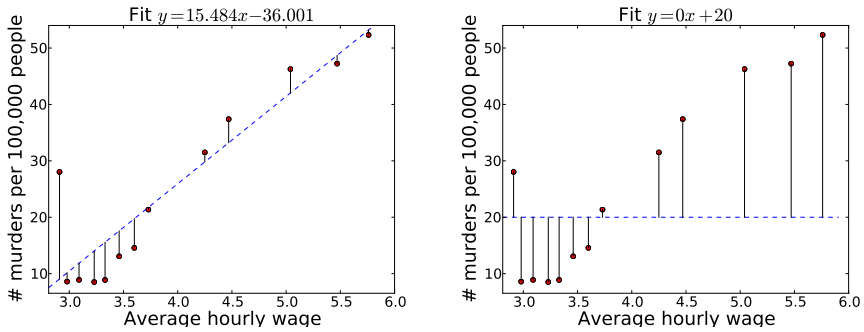
Recall the formula for a line from high school: $y = mx + b$. We normally rewrite that using elements of vector $\vec{B}$ in preparation for describing it with vector notation from linear algebra. For simplicity, though, we'll stick with scalar coefficients for now:

$$y = b_1 + b_2 x$$

The "best line" is one that minimizes some cost function that compares the known y values at x to the predicted y of the linear model that we conjure up using parameters b_1, b_2. A good measure is the *sum of squared errors*. The cost function adds up all of these squared errors to tell us how good of a fit our linear model is:

$$Cost(B) = \sum_{i=1}^{n} (\underbrace{b_1 + b_2 x_i}_{\text{linear model}} - \overbrace{y_i}^{\text{true value}})^2$$

As we change the linear model parameters, the value of the cost function will change. The following graphs shows the errors/residuals that are squared and summed to get the overall cost for two different "curve fits."



The costs are 533.82 for the left and 3563.50 for the right.

The good news is that we know the cost function is a quadratic, which is convex and has an exact solution. All we have to do is figure out where the partial derivatives of the cost function are both zero; i.e., where the cost function flattens out (at the bottom).

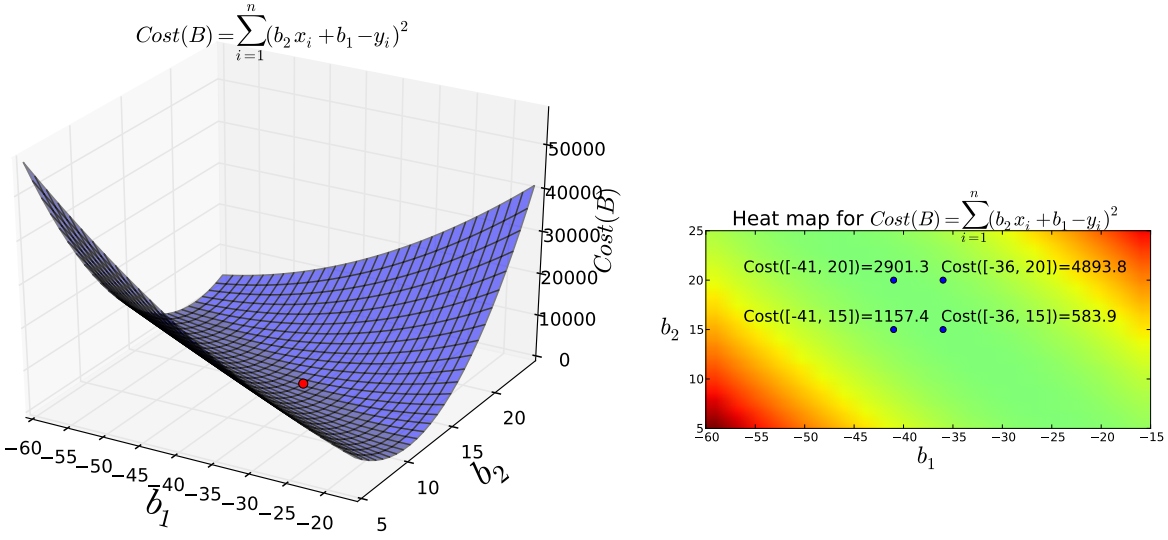
$$\nabla Cost(B) = 0 \qquad \text{(Analytic solution to optimization)}$$

For our purposes, though, we'll use gradient descent to minimize the cost function.

To show our prediction model in action, we can ask how many murders there would be in Detroit if the average salary were $4.7? (Obviously, these wages are from 30 years ago.) To make a prediction, all we have to do is plug $x = 4.7$ into $y = -36.001 + 14.484x$, which gives us 32.074 murders.

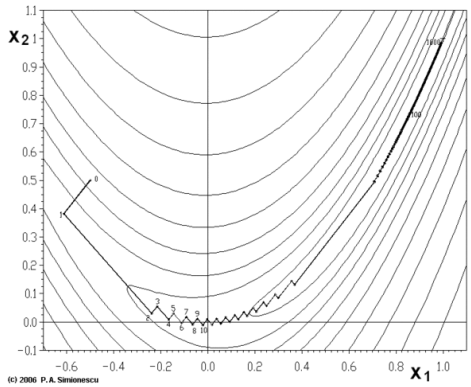
Gradient descent in 3D

Before trying to minimize the cost function, it's helpful to study what the surface looks like in three dimensions, as shown in the following two graphs. The x and y dimensions are the coefficients of our linear model and the z coordinate is the cost function.



What surprised me is that changes to the slope of the linear model's coefficient b_2, away from the optimal $b_2 = 15.484$, cost much more than tweaks to the y intercept, b_1. Regardless, the surface is convex. Unfortunately, based upon the deep trough that grows slowly along the diagonal of (b_1, b_2), gradient descent takes a while to converge to the minimum. We will examine the path of gradient descent for a

few initial starting point. Wikipedia says that the Rosenbrock function is a pathological case for traditional gradient descent and it looks pretty similar to our surface with its shallow valley in this topographic view from above:



The recurrence relation for updating our estimate of $\vec{B} = [b_1, b_2]$ that minimizes $Cost(\vec{B})$ is the same as our previous lab but with a vector instead of a scalar:

$$\vec{B}_{i+1} = \vec{B}_i - \eta \nabla Cost(\vec{B}_i)$$

where we will approximate vectors of partial derivatives with partial finite differences defined generically as:

$$\nabla F(\vec{X}) = \begin{bmatrix} \frac{\partial}{x_1} F(\vec{X}) \\ \frac{\partial}{x_2} F(\vec{X}) \end{bmatrix} \approx \begin{bmatrix} \frac{F(\begin{bmatrix} x_1+h \\ x_2 \end{bmatrix}) - F(\vec{X})}{h} \\ \frac{F(\begin{bmatrix} x_1 \\ x_2+h \end{bmatrix}) - F(\vec{X})}{h} \end{bmatrix}$$

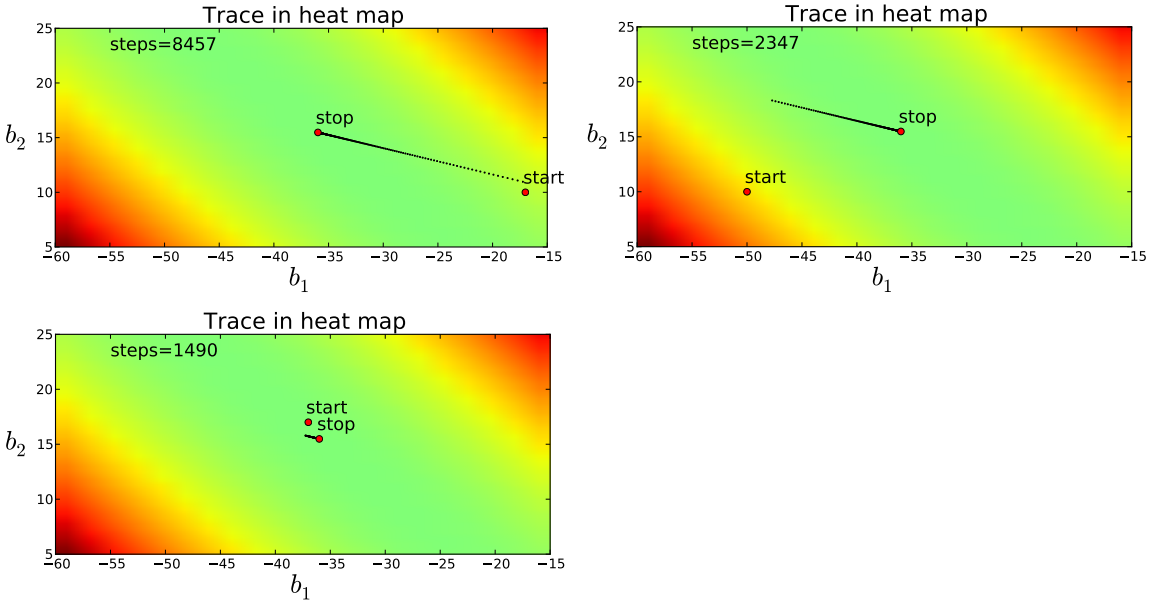
In our case, we will compute the components of a finite difference vector C' ignoring the division by the step h.

The minimization algorithm looks like:

1. Pick an initial B_0
2. Let $B = B_0$
3. $C' = \begin{bmatrix} Cost(\begin{bmatrix} b_1+h \\ b_2 \end{bmatrix}) - Cost(B_i) \\ Cost(\begin{bmatrix} b_1 \\ b_2+h \end{bmatrix}) - Cost(B_i) \end{bmatrix}$, the finite difference in each direction separately
4. Let $B_{i+1} = B_i - \eta C'$
5. Goto step 3 until $abs(Cost(B_{i+1}) - Cost(B_i)) < precision$ **and** when the *Cost* starts going back up

Using a small learning rate, my solution takes 4843 steps starting from coordinate (-45,10) using a very small step size, which gives me a fairly decent approximation of the minimum: [-36.00066933 15.48414587]

compared to the analytic solution [-36.000625 15.484375]. Starting from about the same distance away in the shallow valley at (-45,25), my solution takes 5142 steps. Cutting my step size by 5x, takes 30463 steps but gives a slightly more precise result [-36.00063497 15.48432944]. Starting at (-45,25) takes 31958 steps. Surprisingly, when I start very close to the minimum at (-36,15), my solution takes 47350 steps and does not exactly give a more accurate result. "*Your mileage may vary.*"



When I crank up the learning rate and use a very small step size, my solution converges must faster and with the same accuracy. For example, with 10 times the learning rate as before, (-45,25) converges in 3193 steps instead of 31958 steps.

Getting fancy

For the most part, the algorithm described above converges on the analytic solution with a high degree of precision, assuming that your h is small enough. In some cases however it only gets 3 digits of precision versus about six, depending on where the starting point is. To improve that accuracy, we can improve our termination condition from simply comparing previous and new value of the cost function to use the Euclidean distance between B_{i+1} and B_i. To increase convergence rate, it's a good idea to have different η learning rates per dimension, particularly in this case because we want to speed along the valley of b_1, but not jump too far on b_2 as it is already steep; the finite difference will cause it to move along quickly.

For extremely large data sets or very high dimension data sets, you can use *stochastic gradient descent* which takes a single random or a small random subset of the samples when computing the cost function. Surprisingly, this still converges and takes much less time to compute. I believe that the math says that the algorithm will converge no matter how many samples (don't quote me on that). Further, picking a random subset adds noise to the process, which has been shown to pull gradient descent out of local minima and get it moving again towards a lower minimum. In our case, there is an exact solution and therefore a single minimum so we would only get a speed benefit from stochastic gradient descent.

Your task

You will use gradient descent to solve the linear regression problem above, using the same data. As part of your final submission, you must provide heat maps with traces that indicate the steps taken by your gradient descent as I have shown above. Have your program choose two random starting B_0 vectors (you can use `random.randrange()`) to produce your heat maps. In `opt/regression.py`, define a function called `minimize` that takes the indicated parameters and returns the minimum B parameters of your linear model, the number of steps, and the trace array of intermediate B_i values.

```
def minimize(f, B0, eta, h, precision):
    trace = []
    B = B0
    steps = 0
    while True:
        steps += 1
        if steps % 10 == 0: # only capture every 10th value
            trace.append(B)
        ...
    ...
    return (B, steps, trace)
```

As an example, in my `opt/plot_regression.py` script, I call that function like this:

```
(m,steps,trace) = minimize(Cost, B0, LEARNING_RATE, h, PRECISION)
```

Use `pylab.imshow()` to draw the heat map whose b_1 are the y intercepts, b_2 coordinates are the slopes, and heat value is the cost of x, y. Hide all of your heat map construction in function in `opt/plot_regression.py`:

```
def heatmap(X, Y, trace): # trace is a list of [b1, b2] pairs
    ...
```

so we can call it like this:

```
heatmap(HOURLY_WAGE, MURDERS, Cost, trace)
show() # finally show the heatmap
```

Your `heatmap` function will create two ranges for the b_1 and b_2 coordinates and then simply compute the cost, of storing it in a matrix such as `C[b1][b2] = Cost([b1,b2],...)`. The trace is plotted on top of the map once you create it. The coordinate ranges of the heat map are not related to the trace at all. It took me a while to figure out all of the crazy methods to draw the heat maps so I'll toss you a bone and just provide it for you (it goes in your `heatmap()` function):

```
    imshow(C,
           origin='lower',
           extent=[min(b1), max(b1), min(b2), max(b2)],
           vmax=abs(C).max(), vmin=-abs(C).max()
    )
```

I plot the trace using:

```
plot(p[0], p[1], "ko", markersize=1)
```

Please show the information as I have shown in the graphs to make it easier to compare results and for me to grade.

You will have to pick an appropriately small step value h to get a decent approximation of the derivative through finite differences. You want that number to be small enough that your algorithm does not oscillate around the minimum. If the number is too big it will compute a finite difference that leads to B_{i+1} leaping across the minimum to the other wall of the function. You must pick a learning rate η that allows you to go as fast as you can but not so fast that it overruns the minimum back and forth. When I crank up my learning rate too far, I also see the algorithm go off into the weeds and stops with a minimum of [-4.86000929e+10 -2.85744570e+12].

Testing

You can run `opt/test_regression.py` to check your `miminize` function. Here's my sample run:

```
$ python test_regression.py
starting at [-23, 12]
gradient descent gives [-36.00034318 15.48430235] Cost 2373170.71366
exact is [-36.000625000000007, 15.484375]
starting at [-50, 10]
gradient descent gives [-36.00062579 15.48437039] Cost 2373166.75674
exact is [-36.000625000000007, 15.484375]
starting at [-36, 17]
gradient descent gives [-36.00062582 15.4843704 ] Cost 2373166.75627
exact is [-36.000625000000007, 15.484375]
starting at [-36, 15]
gradient descent gives [-36.00034322 15.48430236] Cost 2373170.71306
exact is [-36.000625000000007, 15.484375]
starting at [100, 17]
gradient descent gives [-36.0003431 15.48430233] Cost 2373170.71481
exact is [-36.000625000000007, 15.484375]
starting at [-15, 0]
gradient descent gives [-36.00034325 15.48430237] Cost 2373170.7127
exact is [-36.000625000000007, 15.484375]
ALL TESTS PASS
```

Resources

There is a lot of material out there on the web that can be helpful.

- Finite difference at Wikipedia
- Stochastic Gradient Descent Tricks
- Numerical recipes (See Chap 10 on minimization of functions)
- Single verbal minimization in line searches
- Andrew Ng's CS229 Lecture notes
- Data analysis with Python

You must tweak the step size and other parameters so that your results agree with the first three decimal points of the analytic solution [-36.000625000000007, 15.484375]. (My solution is much better than that, except for a couple of weird starting positions where it only gets three decimal places.)

- Your script `opt/regression.py` with the `minimize()` function.
- Your script `opt/plot_regression.py`, which must have your `heatmap` function.
- A PDF of your graph with two visible traces on two heat maps or the same heat map if the traces are clear. The graph should include the start, stop location and the number of steps as I have done on mine. As part of your PDF, please indicate the B parameters you compute with your minimize function.

Tag when completed with `regression`.

Part VI

Text Analysis

Summarizing Reuters Articles with TFIDF

Goal

The goal of this task is to learn a core technique used in text analysis called *TFIDF* or *term frequency, inverse document frequency*. We will use what is called a *bag-of-words* representation where the order of words in a document doesn't matter—we care only about the words and how often they are present. A word's TFIDF value is often used as a feature for document clustering or classification. We will use it simply as a document summary mechanism. The more a term helps to distinguish its enclosing document from other documents, the higher its TFIDF score.

This task is also an opportunity to practice organizing your Python code as a set of functions rather than an unstructured script (blob) with a bunch of global variables. You will also learn how to translate some simple algorithms written in pseudocode to Python code. As a practical matter, you will learn how to process XML files in Python. Please use file `tfidf/tfidf.py`.

Discussion

One way to summarize a text document is to list, say, the top 25 words that seem most important. That could also be used to compare documents to see if they're talking about the same thing. For example, I had to solve a problem 15 years ago to reduce noise in the forums of a Java developer's website. Users were posting stupid posts about movies and were also putting database questions in the forum on GUIs. The goal was to detect non-Java posts and also to detect misplaced posts. What does it mean to "talk about Java"? How do I know when someone is talking about databases versus GUIs? My solution was to identify the words important to Java as a whole ("Java-speak"), database, and GUI posts. Any posts that did not have words important to Java, were tossed out as irrelevant after giving them a mild smack on the snout. Similarly, posts without words relevant to databases were compared to vocabularies associated with other topics to see if another forum would be more appropriate. To make this work, I needed a precise definition of "important words." As I did for that project, you will use a classic text analysis technique called TFIDF in this project.

Certainly a word is important to a document if it's used a lot, but that would also include words like "the" so we need to discount words used frequently among our *corpus* (set of documents). So, we boost words used frequently in a document but attenuate words that are used in a lot of documents. For more on this topic, see Introduction to Information Retrieval.

The *term frequency* is just the term count within a document divided by the number of words in that document (some people use "frequency" to mean "count" but that is an affront to the gods):

$$tf(t,d) = \frac{count(t), t \in d}{|d|} \qquad \text{(Term frequency of term } t\text{, document } d\text{)}$$

A term's *document frequency* is the count of documents containing that term divided by the total number of documents:

$$df(t,N) = \frac{|\{d_i : t \in d_i,\ i = 1..N\}|}{N} \qquad \text{(Document frequency of } t \text{ in } N \text{ documents)}$$

We can think of the document frequency as the probability of seeing t in a document.

In order to attenuate the TFIDF scores for terms with high document frequencies, we need the document frequency in the denominator:

$$tfidf(t,d,N) = \frac{tf(t,d)}{df(t,N)} \qquad \text{(First approximation to TFIDF)}$$

This formula is meaningful but gives a poor term score because the document frequency tends to overwhelm the term frequency in the numerator so we take the log of the denominator first. Here's the formula slightly rewritten as it is normally shown:

$$tfidf(t,d,N) = tf(t,d) \times log(\frac{1}{df(t,N)}) \qquad \text{(TFIDF with attenuated document frequency)}$$

When t is in every document, idf is $log(1/df(t,N)) = log(1) = 0$. When t is in very few documents, such as $1/10^8$, idf is $log(10^8)$, which is about 18.4.

> **Aside.** To prevent division by 0 errors when a term does not exist in a corpus (e.g., $df(t,N) = 0$ in search applications where we pass unknown term(s) t), we can simply add 1 to the denominator. This is similar to *additive smoothing* that you will see when estimating term probabilities in document classifiers. The technique is like pretending there is an imaginary document with every unknown word (and, indeed, every possible word). To keep document frequencies in $[0..1]$, we can to bump the document count, N, as well.
>
> $$df(t,N) = \frac{|\{d_i : t \in d_i,\ i = 1..N\}| + 1}{N + 1} \qquad (df \text{ with smoothing})$$
>
> For example, if we have a vector of words in a search query $[the, apple, cat, foo]$ aggregated from 100 documents, we might get a set of document frequencies like this:
>
> $$[\frac{100}{100}, \frac{4}{100}, \frac{9}{100}, \frac{0}{100}]$$
>
> With smoothing, we would get:
>
> $$[\frac{101}{101}, \frac{5}{101}, \frac{10}{101}, \frac{1}{101}]$$
>
> This is like converting zeros to some really small number (assuming N is large) and adding that same small number to the other document frequencies.

To summarize a document, we can order its terms by *tfidf* in reverse order and look at the top 20 words, for example. To get the lexicon of a topic like databases, we can collect a known set of database posts into a single document and compute the tfidf in association with the aggregated documents of the other topics. Any word below a certain threshold, that we find by eyeballing it, is considered not relevant to that particular topic.

For example, here is a set of terms and the associated *tfidf* computed from a sample Reuters article. It's clear that it's talking about Nielsen ratings for news programs, without even looking at the original article.

Term	*tfidf*
rating	0.12332931962551781
fox	0.11911646171233138
nbc	0.11911646171233138
homes	0.11408838230482544
cbs	0.0794109744748876
audiences	0.0794109744748876
neilsen	0.0794109744748876
evening	0.06678324701893232
abc	0.06678324701893232
watching	0.0634765565309808

To compute TFIDF, we need an overall index that maps term t to document frequency $df(t, N)$ for all t in all N documents and an index that maps document d to another index that maps each $t \in d$ to $tf(t, d)$. From that, we can compute all of the TFIDF scores. That is what you will do for this project, as described in the next section.

Your task

To implement this project, you have six key functions to implement. Four of them are in the pseudocode shown in floating boxes interspersed below. You must translate them to Python in a file called `tfidf.py`.

Function: *words*(document *d*)
Input: Document *d*
Result: non-unique list of words *wordlist*
  Replace numbers, punctuation, tab, carriage return, newline with space
  wordlist = Split *d* into words
  Strip out $w \in$ *wordlist* smaller than 3 letters
  Normalize $w \in$ *wordlist* to lowercase
  return *wordlist*

You must also provide a function called `filelist(pathspec)` that returns a list of all file names that match `pathspec` and that *have non-zero file sizes*:

```
def filelist(pathspec):
    ...
    return files
```

For example, I might pass in string `../data/reuters-vol1-disk1/*.xml`. Naturally, if this doesn't work, then the rest of the code will not work as it won't get the proper data. That function should only consider the files in the specified directory, not subdirectories. You will probably want to use Python function `glob.glob()`.

To process XML files use `ElementTree`; learn about it here. Create a function called `get_text()` to open a file, load it as XML, find the title and text elements and return that combines text as a string. It's important

Function: *create_indexes*(list of $files$)
Input: List of filenames $files$
Result: (Map document name to Counter object mapping term to frequency map *tf_map*, Counter object mapping term to document count df)

```
df = Counter(); tf_map = {}
foreach f in files do
    d = get_text(f)
    words = words(d)
    n = len(words)
    tf = Counter(words)
    # walk unique word list
    foreach t ∈ tf do
        tf[t] = tf[t]/n # convert to a term freq from count
        df[t] += 1      # not currently a frequency; it's a count
    end
    tf_map[f] = tf
end
return (tf_map,df)
```

Function: *doc_tfidf*(tf, df, N)
Input: Term to frequency map tf
Input: Term to document count map df
Input: Number of documents N
Result: Map of each term in doc (tf) to TFIDF score

```
tfidf = {}
foreach t ∈ tf do
    df_t = df[t]/N
    idf_t = 1/df_t
    tfidf[t] = tf[t] × log(idf_t)
end
return tfidf
```

Function: *create_tfidf_map*($files$)
Input: List of xml filenames $files$
Result: Map from file name to map of term to TFIDF scores

```
(tf_map, df) = create_indexes(files)
tfidf_map = {}
foreach f ∈ files do
    tfidf = doc_tfidf(tf_map[f], df)
    tfidf_map[f] = tfidf
end
return tfidf_map
```

that we all follow the same text normalization so that we get the same word list and hence TFIDF scores for comparison.

```
# http://eli.thegreenplace.net/2012/03/15/processing-xml-in-python-with-elementtree/
# summary: use ElementTree, good tutorial

def get_text(fileName):
    """
    Read an xml file and return the text from <title> and <text>.
    Concatenate those two elements, putting a space in between so it doesn't
    form an incorrect compound word.
    """
    ...
    return text
```

As part of your development work you will use lots of maps that look like {dog: 36, cat: 19, ...}. Those integers, such as term counts, are easy to compute yourself but Python has an object that is effectively a histogram called `Counter`. For example, if you give it a list of words, it will return an object that maps terms to their count. When you print them out, it will do so in reverse order of term count, which is very handy for testing. Further, the unit tests I provide expect Counter objects.

For what it's worth, my implementation is just 60 lines including the `import` statements. This is not a huge project but it is tricky when messing around with all of these maps of maps and lists of things. Start by understanding the problem and working a few TFIDF examples manually. Then, build a simple functions and test them individually before moving on to the more complex functions. For example, you should start by building `filelist()` and then probably `get_text()`. My typical strategy is to design from the top down and test from the bottom up.

XML Input

As part of this project, I will provide you with a set of Reuters news articles in XML format, which will be the input to your program. From it, you will create the appropriate indexes and I will test those values against what I computed with my solution.

The format of the files doesn't matter much except that you need to pull out the `title` and `text` tags. The `p` paragraph tags inside `text` need to be collected. All of this text is what you will return from `get_text()`.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<newsitem itemid="131701" ...>
<title>German consumer confidence rises in Aug/Sept</title>
<text>
<p>German consumer confidence rose...</p>
<p>The Icon index, which...</p>
</text>
...
</newsitem>
```

The collection of Reuters articles is considered proprietary to Reuters and, to get access to the data,

the faculty had to promise Reuters the data would not be made available on a public website or given to anyone else. Please treat this data with care, do not posted to github, etc.

Testing

Test your code using the following command line (with your `tfidf.py` is in the same directory):

```
$ python -m pytest test_tfidf.py
============================= test session starts ==============================
platform darwin -- Python 2.7.7 -- py-1.4.20 -- pytest-2.5.2
collected 5 items

test_tfidf.py .....

========================== 5 passed in 0.04 seconds ===========================
```

If you don't see all tests passing, and there is a problem at a basic level with your software.

Note that the test file imports your file with:

```
from tfidf import *
```

If you name it incorrectly, the program won't work.

To test the entire corpus of Reuters articles, also run your program as follows, potentially with a different path specification.

```
$ python test_corpus.py '../data/reuters-vol1-disk1/*.xml'
```

Note that the quotations around the path specification are required to prevent the command line from expanding `*.xml`. You want that path specification to go in as a single argument, not a list of files.

The core of `test_corpus.py` is:

```
(file_to_histo, word_to_numdocs) = create_indexes(files)
for f in files:
    pairs = doc_tfidf(file_to_histo[f], word_to_numdocs, N)
    # convert map to a Counter object so we can use most_common()
    term_pair = Counter(tfmap).most_common(1)[0]
    print os.path.basename(f), "(\%s, \%1.4f)" \% (term_pair[0], term_pair[1])
```

The output, which I have provided in file `corpus_output.txt.7z`, starts with (there are actually more files than 81880, but they are mysteriously empty):

```
81880 files
131674newsML.xml (ewe, 0.7714)
131675newsML.xml (tisa, 0.2909)
131676newsML.xml (ingenico, 0.4040)
131677newsML.xml (lisbon, 0.1876)
131678newsML.xml (drachmas, 0.0844)
131679newsML.xml (satisfying, 0.1774)
13167newsML.xml (cents, 0.3350)
131680newsML.xml (tightness, 0.0891)
131681newsML.xml (tisa, 0.3626)
```

```
131682newsML.xml (intervention, 0.1766)
131683newsML.xml (nordic, 0.1249)
131684newsML.xml (oilseeds, 0.1030)
131685newsML.xml (crowns, 0.1299)
131686newsML.xml (trelleborg, 0.2399)
131687newsML.xml (nni, 0.4351)
131688newsML.xml (nantes, 0.3898)
131689newsML.xml (advances, 0.4745)
13168newsML.xml (utilicorp, 0.3165)
131690newsML.xml (sas, 0.2201)
131691newsML.xml (austria, 0.0869)
131692newsML.xml (wage, 0.1244)
131693newsML.xml (herzog, 0.2330)
...
```

My implementation takes about 1 minute 30 seconds to compute TFIDF scores for 81880 XML files loaded from an SSD on a fast machine. Loading those files takes just 5 seconds.

Resources

I provide for you the following files:

- `test_tfidf.py`: some simple tests using py.test.
- `test_corpus.py`: prints out the file, term, and TFIDF score for the highest scoring term in each file.
- `corpus_output.txt.7z`: compressed output from running test_corpus.py
- `reuters-vol1-disk1-subset.7z`: compressed directory full of XML files—the corpus. It is 385M when uncompressed. This file is in Canvas' files area: https://usfca.instructure.com/courses/1553155/files/.

Deliverables. `tfidf/tfidf.py` file. Please make sure that there is no extraneous output generated by your code. Tag when completed with `tfidf`.

Extra credit — Search Engine

In this project, we created an index from term to the number of documents that contain that term. If we extend that to be an index from term to the list of documents containing the term, we can get the same results as we did before. The benefit would be that we could also create a search engine.

Given a query such as "consumer confidence," we could merge the list of files containing those two terms and display those to a user. It's fast and works great! The only problem is that we might get 1000 documents back and we'd really like to show the most relevant documents first. Using the tf_map index, we can compute a relevance score for a query, relative to a document, by summing the TFIDF scores for each term in the query that is present in the document. The document with the highest two TFIDF scores for "consumer confidence," would be the first document we displayed.

You need to modify the df map from above, use additive smoothing to handle unknown words, and then implement the following function.

```
def search(query): # query is a string with a list of words
    docs = []
    # find list of documents for each term in query
    # docs = intersection of these files
    # compute sum of TFIDF scores for each term in query relative to each document in docs
    # sort documents by reverse score
    # Returns a list of document filenames in reverse TFIDF order
    return docs
```

Part VII

A Taste of Distributed Computing

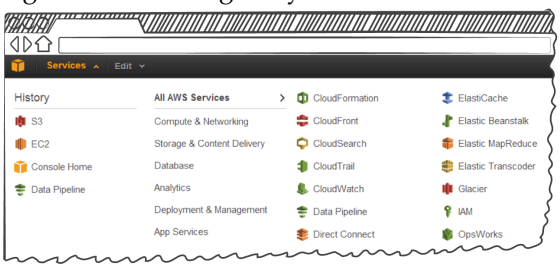
Launching a Virtual Machine at Amazon Web Services

Discussion

The goal of this lab is to teach you to create a Linux machine at Amazon Web Services, login, copy some data to that machine, and run a simple Python program on that data.

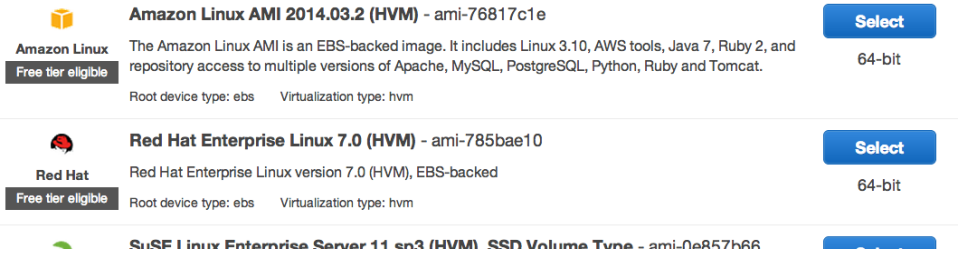
Steps

1. Login to AWS and go to your AWS console.

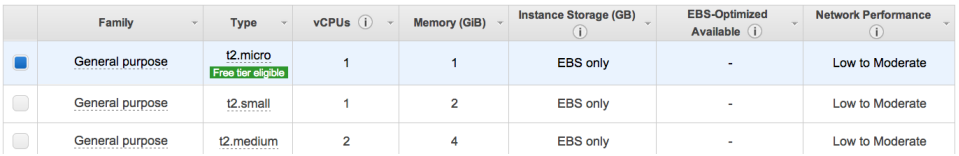


2. Click "Launch Instance", which will start the process to create a virtual machine in the cloud. An instance is just a virtual machine.

3. Select the "Amazon Linux AMI" server, which should be the first one. This is a commonly-used *image* that results in a Linux machine that contains lots of useful goodies as you can see from that list, such as Python and MySQL. An image is just a snapshot of the disk after someone carefully installs software properly on a Linux machine. This means we don't have to install software every time we create a new machine.

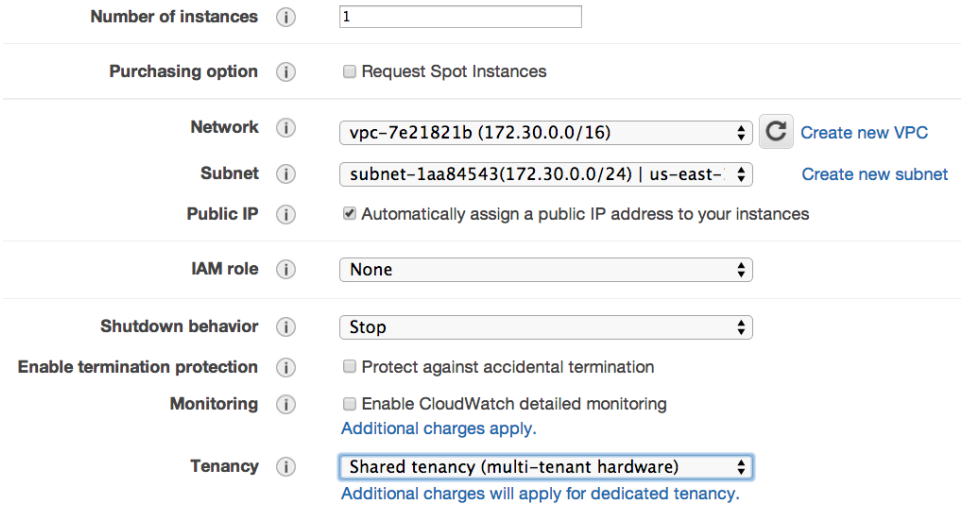


4. Select instance type "m1.micro," which should be the first machine type listed. This machine is very low powered but is sufficient for playing around. Click "Next: configure instance details."



	Family	Type	vCPUs	Memory (GiB)	Instance Storage (GB)	EBS-Optimized Available	Network Performance
☑	General purpose	t2.micro Free tier eligible	1	1	EBS only	-	Low to Moderate
☐	General purpose	t2.small	1	2	EBS only	-	Low to Moderate
☐	General purpose	t2.medium	2	4	EBS only	-	Low to Moderate

5. You can leave the configuration details as-is:



6. Ignore the network interface set up and advanced details. Click "Next: Add storage."

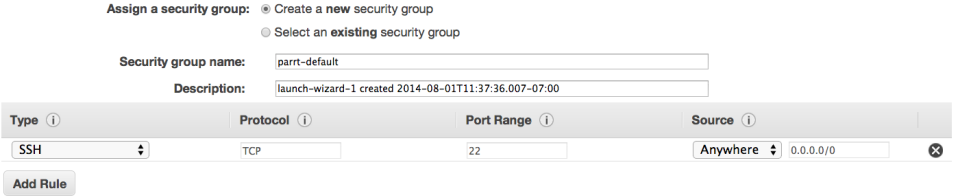
7. It shows that it will give us 8G of disk storage on a magnetic disk by default, which is good enough for our testing purposes. Click "Next: Tag instance."

Type	Device	Snapshot	Size (GiB)	Volume Type	IOPS	Delete on Termination	Encrypted
Root	/dev/xvda	snap-810ffc56	8	Magnetic	N/A	☑	Not Encrypted

8. For the key named "Name", change the value to something like *youruserid*-linux or something like that so that you can identify it later if you have multiple machines going. Click "Next: Configure security group." Then click on the group whose name is "default." Your list of security groups might not be the same.

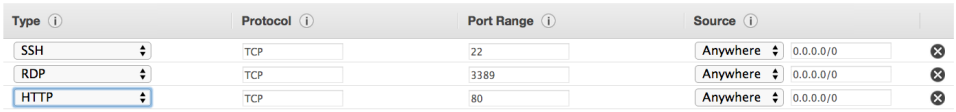
Key (127 characters maximum)	Value (255 characters maximum)
Name	parrt-linux

9. You want to create a new security group, so that you learn how to deal with firewalls. We want to allow SSH access, Windows RDP, and HTTP ports. Name it something like your userid-default. You should be able to reuse this the next time you create an instance just by selecting the name from the existing security groups pulldown. It initially shows just SSH port open so we have to add two more.

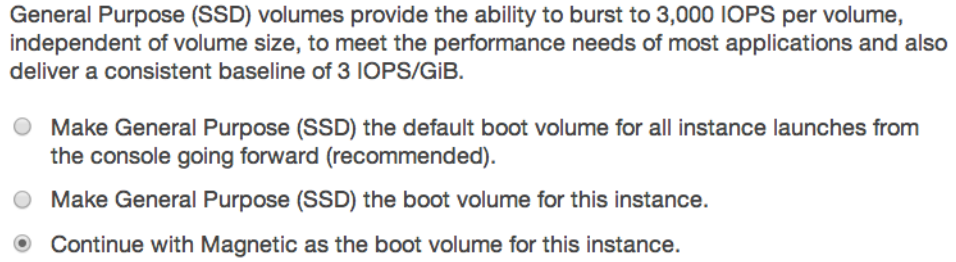


10. Click on the "Add rule" button and select RDP under the type and Anywhere under the source. That

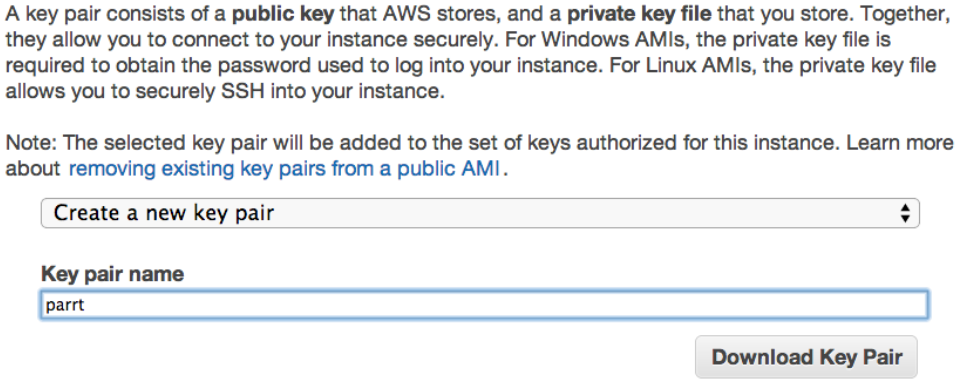
means we want anyone to be able to connect to this machine using the Windows Remote Desktop protocol and from any machine on the Internet. This is not a Windows machine but you will reuse the security group later. As we might want to start a Web server on our cloud computer, add a rule for HTTP.



11. Click "Review and launch," which will pull up a dialog box asking you to select whether you want SSD or old-school spinning magnetic disk. As we are just testing things and don't care about I/O speed, choose the magnetic disk and click "Next."

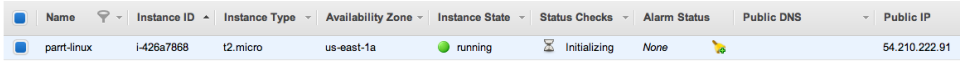


12. Click "Launch," which will bring a dialog box up to select a key pair. A key pair is what allows you to securely access the server and prevent unauthorized access. The first time, you will need to create a new key pair. Name it as your user ID then click on "Download key pair." It will download a *userid*.pem file, which are your security credentials for getting into the machine. Save that file in a safe spot. If you lose it you will not be able to get into the machine that you create.

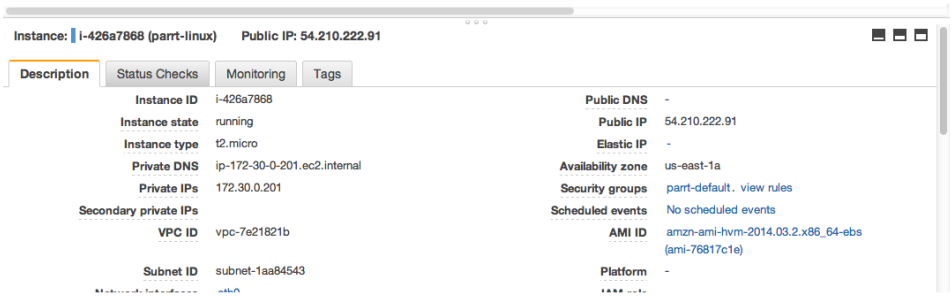


13. Click on the "I acknowledge that I have ..." checkbox then "Launch instances." You should see something like:

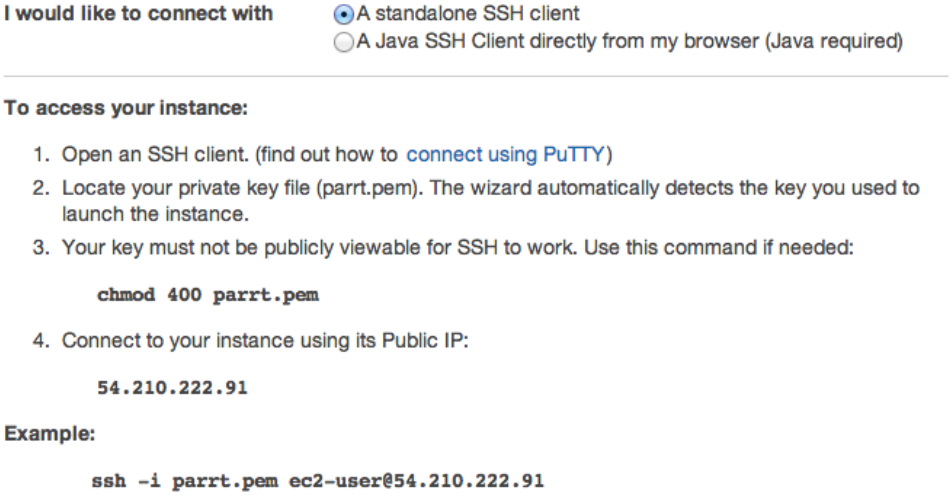
14. Click on the "i-..." link to go to the EC2 console showing your instance.



15. Click on your instance and you should see a description box at the bottom. Look for the "Public IP" address, which is 54.210.222.91 in this case:



16. Click on the "Connect" button at the top of the page and it will bring up a dialog box that tells you how to connect to the server. You want to connect with "A standalone SSH client" link (Java is now a security risk in the browser so we can't use that choice.) Inside you will see the `ssh` command necessary to connect to your machine. If you have Windows, there is a link to show you how to use an SSH client called PuTTY.



For mac and linux users, we will use the direct `ssh` command from the command line. It will be something like:

```
ssh -i ~/Dropbox/licenses/parrt.pem ec2-user@54.210.222.91
```

Naturally, you will have to provide the full pathname to your user.pem file.

17. Before we can connect, we have to make sure that the security file is not visible to everyone on the computer (other users). Otherwise ssh will not let us connect because the security file is not secure:

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@         WARNING: UNPROTECTED PRIVATE KEY FILE!          @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
Permissions 0644 for '/Users/parrt/Dropbox/licenses/parrt.pem' are too open.
It is required that your private key files are NOT accessible by others.
This private key will be ignored.
bad permissions: ignore key: /Users/parrt/Dropbox/licenses/parrt.pem
Permission denied (publickey).
```

Whoa! Do this:

```
$ cd ~/Dropbox/licenses
$ ls -l parrt.pem
-rw-r--r--@ 1 parrt  parrt  1696 Aug  4 15:15 /Users/parrt/Dropbox/licences/parrt.pem
```

To fix the permissions, we can use whatever "show information about file" GUI your operating system has or, from the command line, do this:

```
cd ~/Dropbox/licenses
chmod 600 parrt.pem
```

which changes the permissions like this:

```
$ ls -l parrt.pem
-rw-------@ 1 parrt  501  1696 Aug  1 12:12 /Users/parrt/Dropbox/licenses/parrt.pem
```

Don't worry if you don't understand exactly what's going on there. It's basically saying that the file is only read-write for me, the current user, with no permissions to anybody else.

18. Try to connect again and it will now warn you that you have never connected to that machine before. Again, this is a security measure. You can simply say "yes" here.

```
ssh -i ~/Dropbox/licenses/parrt.pem ec2-user@54.210.222.91
The authenticity of host '54.210.222.91 (54.210.222.91)' can't be established.
RSA key fingerprint is 49:1d:f6:ff:1a:19:5d:00:bb:cd:43:c1:84:ee:8e:a6.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '54.210.222.91' (RSA) to the list of known hosts.
```

Once you connect, you should see the following output from the terminal:

```
       __|  __|_  )
       _|  (     /   Amazon Linux AMI
      ___|___|___|

https://aws.amazon.com/amazon-linux-ami/2014.03-release-notes/
8 package(s) needed for security, out of 19 available
Run "sudo yum update" to apply all updates.
[ec2-user@ip-172-30-0-201 ~]$
```

The $ is your prompt just like you have on your local machine using the terminal / shell.

19. To get data up to the server, you can cut-and-paste if the file is small. For example, cut-and-paste the following data into a file called `coffee` in your home directory. First copy this data from the PDF:

```
3 parrt
2 jcoker
8 tombu
```

then type these commands and paste the data in the sequence:

```
$ cd ~ # get to my home directory
$ cat > coffee
3 parrt
2 jcoker
8 tombu
^D
$ cat coffee # print it back out
3 parrt
2 jcoker
8 tombu
$
```

The `^D` means control-D, which means end of file. `cat` is reading from standard input and writing to the file. The way it knows we are done is when we signal in the file with control-D.

20. For larger files, we need to use the secure copy `scp` command that has the same argument structure as

secure shell `ssh`. Get another shell up on your laptop. From the directory where you have the `coffee` file on your laptop, use the following similar command:

```
$ scp -i ~/Dropbox/licenses/parrt.pem access.log ec2-user@54.210.222.91:~ec2-user
access.log                                    100% 1363KB   1.3MB/s   00:00
$
```

Do not forget the `:~ec2-user` *on the end of that line.* The `access.log` file is at github under labs/data. From the shell that is connected to the remote server, ask for the directory listing and you will see the new file:

```
$ ls
access.log  coffee
$
```

21. Play around with your instance and then *TERMINATE YOUR INSTANCE WHEN YOU ARE DONE*, otherwise you will continue to get charged for the use of that machine. If you right-click on the instance and say "Stop", it will stop the machine and you still get charged but you can restart it without having to go through this whole procedure. If you say "Terminate", it will toss the machine out and you will have to go through this procedure again.

Deliverables

None. Please follow along in class.

Using the AWS Hadoop Streaming Interface

Goal

Your goal in this lab is to learn how to launch a simple map-reduce job at Amazon using their elastic map reduce mechanism and the shell/commandline. Our application is the trite "word counting," which we will use to find the most common words in a set of Google ads scarfed from the net in `ads1000.txt` at github/parrt/msan501. You'll use Python as in the other labs.

Discussion

Hadoop introduction

Hadoop is a distributed computing framework that supports a map-reduce computing paradigm. The *map* operation executes on multiple machines and gets partial results, which are then combined with the *reduce* operation.

Hadoop is written in Java and so, to use another language such as Python, we have to use the so-called *streaming interface*. That just means that we will write programs that read from standard input and write to standard output.

The *Hadoop file system* (HDFS) is a distributed file system that can handle massive amounts of data by distributing it across multiple machines and hard drives. Hadoop tries to keep map operations on the machines that store the associated data the mappers should run on. That is what typically is done, but we will be using Amazons S3 storage instead since it is the easiest thing to do.

Hadoop splits the input into chunks and splits the chunks into lines before feeding the lines to standard input of the mappers. It gives each chunk of lines to a separate mapper task, which generates partial results. The mappers generate partial results as a set of key-value pairs of the form:

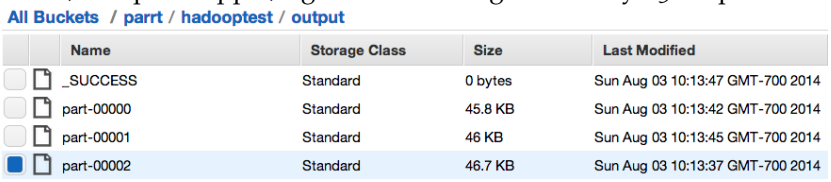
key `\t` *value* `\n`

Because partial results are created on a variety of machines where the map tasks run, hadoop has to collect this data from the machines of the cluster before giving it to the reducers. Hadoop sorts these partial results according to key (via merge sort) and distributes regions of the key space across one or more reducers. A specific key is only seen by a single reducer. A reducer reads these key-value pairs line by line from standard input and is responsible for generating a final result. That output can be whatever we want, but in our case we will use the same key-value output format. We don't have to have any reducers at all, if we just want to run mappers across all the data.

We will be using Amazon Elastic MapReduce (EMR) that will take care of all the details of launching a cluster, running our job, and creating the output files. A hadoop *job* is a chunk of work, which can have one or more tasks. If one of these tasks fails, hadoop tries to rerun them. One of Hadoop's big benefits is that it is fault-tolerant. In a cluster of 1000 computers, it's very possible a machine will go down or the system operator will kick a power cord out by mistake. AWS introduces the notion of a *step*, which is one of more jobs. We will not be using the step mechanism of the EMR GUI to launch jobs because, oddly

enough, it's much easier and quicker to do it from the command line of the master node of the cluster.

Hadoop streaming generates an output file per reducer, which can be handy if we are interested in partitioning, say, sales results per country. In that case, we would have one reducer per country. With three reducers, one per mapper, I got the following files in my S3 output folder:



To get a single output file, we need to specify a single reducer, which we will do below from the command line.

Testing map-reduce on single machine

Before spending money at Amazon to run your job, make sure that it works properly by simulating it from the command line on your laptop. To simulate hadoop collecting data and sending it as standard input to your mapper, we will use `cat`:

```
$ cat /tmp/ads1000.txt
"title" "blurb" "url" "target" "retrievetime"
"Exclusive Music For DJs" "DJ One Stop For Edits, Mash-Ups, Remixes. Browse, Listen, ...
...
```

Then, pipe that input into the mapper (which will appear like standard input to the mapper: `wcmap.py`):

```
$ cat /tmp/ads1000.txt | python wcmap.py
"title" 1
"blurb" 1
"url" 1
"target" 1
"retrievetime" 1
"Exclusive 1
Music 1
For 1
DJs" 1
"DJ 1
...
```

Hadoop always sorts the partial results coming out of each mapper before passing it to the reducer(s), which we can simulate bypassing the output of the mapper through `sort`:

```
$ cat /tmp/ads1000.txt | python wcmap.py | sort
! 1
! 1
! 1
! 1
! 1
!" 1
!" 1
!" 1
!" 1
...
```

Obviously the data is not very interesting because we have not stripped out punctuation, which you can do as an exercise. The point is that the data is sorted by key. Then, we can run the reduce job:

```
$ python wcmap.py < /tmp/ads1000.txt | sort | python wcreduce.py
...
```

The issue is that our Python program does not sort by keys when emitting key-value pairs, but we can use the command line to handle that. Here is our final command line that streams data using pipes between processes:

```
$ python wcmap.py < /tmp/ads1000.txt | sort | python wcreduce.py | sort
! 5
!" 6
"#3 3
"$189 3
"$200 1
...
```

We can also write that to file using:

```
$ python wcmap.py < /tmp/ads1000.txt | sort | python wcreduce.py | sort > output.txt
```

On a multicore machine, this process is virtually identical to what hadoop is doing for us, except of course on a smaller scale and without the network traffic.

S3 storage

AWS's elastic map reduce mechanism likes to process data out of its S3 storage. (It's tempting to try to process a local file with Hadoop from the cluster master, but that doesn't work as a local file is not available to slave nodes.) You will need to create a bucket in S3 that is unique across AWS so maybe use your user ID. My bucket is parrt. And I can access that with web address: parrt.s3.amazonaws.com. As long as I've made the folders underneath public, then I can add elements to the URL to get access to those files. For example, here is the data that I have updated for our lab in the msan501 bucket:

http://msan501.s3.amazonaws.com/data/ads1000.txt

You can load this data into your S3 bucket folder by downloading from msan501/data and uploading it into your own bucket.

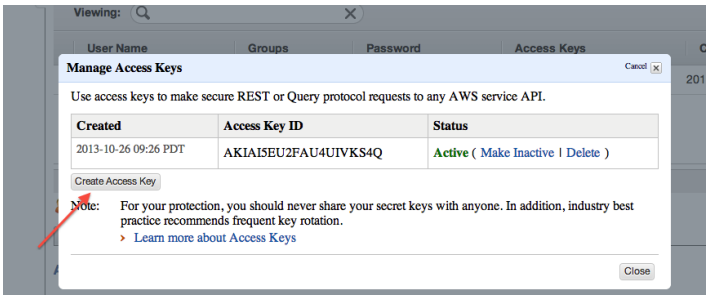
In order to access S3 from the command line of the master server of your cluster, we first need to configure the `aws` command with your access key and secret access key.

```
aws configure
aws s3 ls s3://parrt/hadooptest/
aws s3 cp s3://parrt/hadooptest/output localdir --recursive
```

Actually, I can't get this to work yet as I can't figure out how to give it my .pem file, or whatever it means to resolve the following issue: "`certificate routines:X509_load_cert_crl_file:system lib`"

Launching a cluster

Before creating a cluster, you have to create a root id. For good measure, create an access key as well:



Make sure both of them say they are active.

From the EMR console at Amazon, click on the "Create cluster" button. Choose all the defaults on the resulting page, except:

- Turn OFF "Termination protection" near the top.
- Set a log dir like `s3://parrt/hadooptest/logs` or something appropriate.
- You can delete Hive and Pig "Applications to be installed" as it slows down the launch and we aren't using them for this lab.
- Under "Security and access", select the EC2 key pair you created previously (and saved in userid.pem or whatever) or create a new one.

The default is to use three machines, one master and two core. That's fine.

At the bottom, click "Create cluster." It will take you to the status page showing you the cluster starting up. It will say Starting for a while and then it will eventually say "`Provisioning Amazon EC2 capacity`" and eventually Waiting, which means we can go to the server and launch jobs.

> If you see the error message "`No active keys found for user account`" after a few minutes, that means that you have not created the key correctly. See the stackoverflow Q/A.

After about 10 or 15 minutes, we will have a cluster up with everything installed properly to run Hadoop jobs!!!

Running a Hadoop job

On a cluster that has Hadoop installed, the easiest way to run a job is from the commandline shell. To demonstrate, we can use shell commands themselves as mappers and reducers:

```
$ hadoop jar /home/hadoop/contrib/streaming/hadoop-streaming.jar \
    -input s3://msan501/data/ads1000.txt \
    -output s3://parrt/hadooptest/output \
    -mapper /bin/cat \
    -reducer "/usr/bin/wc -l"
```

Once the job finishes, you can go to the S3 console at AWS and look in your directory for the output. Your output files, part-0000*, will have one line with just a number like 330 in it. That is the result of running wc. After downloading from S3's web interface, I can look at the results with a simple command on my local machine:

```
$ cat ~/Downloads/part-0000[0-2]
330
```

```
302
368
```

> **Warning!** If you launch a cluster and tell it to write output to an existing directory, it will fail with a permissions issue and the cluster will terminate. Consequently, use output directories with different names for each run.

Now let's run our Python code:

```
$ hadoop jar /home/hadoop/contrib/streaming/hadoop-streaming.jar \
      -files s3://parrt/hadooptest/wcmap.py,s3://parrt/hadooptest/wcreduce.py \
      -mapper wcmap.py \
      -reducer wcreduce.py \
      -input s3://msan501/data/ads1000.txt \
      -output s3://parrt/hadooptest/output2
```

We can also run with the Python files locally if we copy them up. On your local machine, do this with the appropriate pem file:

```
$ scp -i ~/Dropbox/licenses/parrt.pem wc*.py \
   hadoop@ec2-54-87-142-212.compute-1.amazonaws.com:~hadoop
```

You will also need that certificate file to communicate with the slave machines so copy that pem file to the master as well:

```
$ scp -i ~/Dropbox/licenses/parrt.pem ~/Dropbox/licenses/parrt.pem \
   hadoop@ec2-54-87-142-212.compute-1.amazonaws.com:~hadoop/.ssh
```

On the master, set the permissions of that file as we did in a previous lab:

```
$ chmod 600 .ssh/parrt.pem
```

Now that we have the Python code to the master, we need to copy the code to each of the slaves as the slaves will need to run the same code. To do that we need to know what the IP addresses of the slaves are:

```
$ hadoop dfsadmin -report | grep ^Name | cut -f2 -d:
DEPRECATED: Use of this script to execute hdfs command is deprecated.
Instead use the hdfs command for it.
10.110.197.20
10.206.58.80
$ scp -i .ssh/parrt.pem wc*.py hadoop@10.110.197.20:~hadoop
wcmap.py 100% 384 0.4KB/s 00:00
wcreduce.py 100% 677 0.7KB/s 00:00
$ scp -i .ssh/parrt.pem wc*.py hadoop@10.206.58.80:~hadoop
wcmap.py 100% 384 0.4KB/s 00:00
wcreduce.py 100% 677 0.7KB/s 00:00
```

Now, still on the master server, run the job without the `-files` option:

```
$ hadoop jar /home/hadoop/contrib/streaming/hadoop-streaming.jar \
      -mapper wcmap.py \
      -reducer wcreduce.py \
      -input s3://msan501/data/ads1000.txt \
      -output s3://parrt/hadooptest/output2
```

To run with a single reducer instead of 3, use:

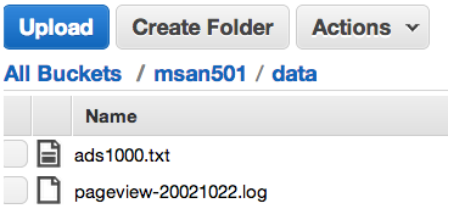
```
$ hadoop jar /home/hadoop/contrib/streaming/hadoop-streaming.jar \
   -D mapred.reduce.tasks=1 \
   -mapper wcmap.py \
   -reducer wcreduce.py \
   -input s3://msan501/data/ads1000.txt \
   -output s3://parrt/hadooptest/output3
```

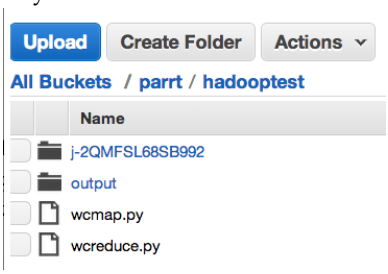
The output file will have a set of unsorted key-value pairs.

Running a job via AWS web interface

You can also run a job at Amazon without knowing the command line interface, but it's a lot of clicking. Here is the process:

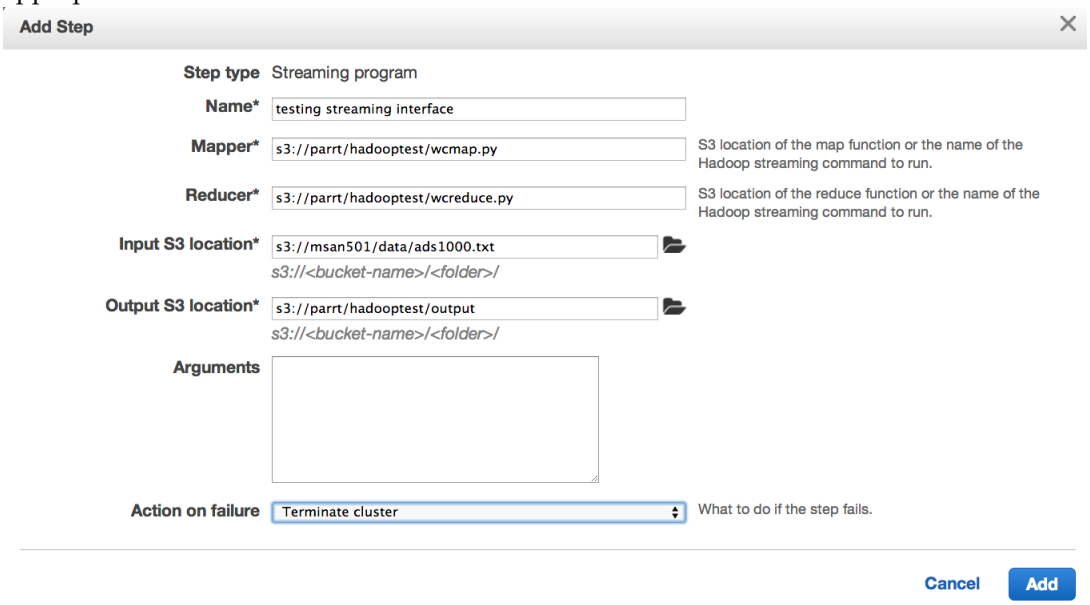
1. Load your data into an S3 bucket folder by downloading from msan501/data and uploading it into your own bucket. Load your code into S3, presumably in a different folder.





2. Create a cluster as shown previously and under "Steps" near the bottom, select "Streaming program" from the "Add step" drop-down. If you want to keep the cluster alive so that you can rerun jobs more quickly, set auto terminate to no. Otherwise set it to yes so that the cluster disappears after your job and you will not be charged further for it.

3. Enter the fields of the step dialogue as shown, substituting your user ID or your bucket/folder names as appropriate:



Warning! If you launch a cluster and tell it to write output to an existing directory, it will fail with a permissions issue and the cluster will terminate. Consequently, use output directories with different names for each run.

For convenience, here is the text so that you can cut-and-paste:

```
s3://parrt/hadooptest/wcmap.py
s3://parrt/hadooptest/wcreduce.py
s3://msan501/data/ads1000.txt
s3://parrt/hadooptest/output4
```

4. Wait about 15 minutes while Amazon creates the cluster and then wait a minute or so for your actual job to run.

5. Download or examine your data with the S3 interface.

Once our cluster is up, you can run another job "quickly" by adding another step. Go into EMR and select your cluster and then click "add step". That is a tiny link hidden down in the Steps area. Or, run a command-line job.

Extra credit

Now that you know how to run a job, it's a good idea to spend the time improving the mapper so that it strips punctuation. That way we'll get a much better set of keys. You can also strip characters not in the printable ASCII code which will automatically strip non-English characters.

Resources

- You will find `wcmap.py` and `wcreduce.py` at msan501 github. There is also the necessary data file, `ads1000.txt`.
- A helpful tutorial, from which we get our sample programs.

Deliverables

None. Please follow along in class.