

**Computational Freakonomics:
Computational Tools for Social Studies Analysis**

Mark Guzdial and Richard Catrambone
College of Computing and School of Psychology
Georgia Institute of Technology

June 18, 2006

Copyright held by Mark Guzdial and Richard Catrambone, 2006.

Contents

Contents	ii
List of Program Examples	iv
List of Figures	v
1 Starting Python and Reading Files	1
1.1 Starting with Python and SciPy	1
1.2 Reading Files	5
1.3 How CSVfile Works	9
2 Plotting	13
2.1 Your Basic Plot: Slicing Up The World's Population	13
2.2 Options on the Plot	17
2.3 The Plot Thickens: Combining Plots to Determine US and UK Growth Rates	19
3 Descriptive Statistics	25
3.1 Average or mean: Petroleum Tax Prices	25
3.2 Standard Deviation	27
3.3 Histogram	29
4 Correlation	31
4.1 Computing correlation: Is it the company, or war in the Mid- dle East?	31
4.2 But do we believe it?	35
5 Text Analysis	41
5.1 Visualizing textual differences: Bacon v. Shakespeare	41
5.2 Counting Text Patterns	49
6 Hypothesis Testing	55
6.1 The Context: Elections and Unemployment Rates	55
6.2 T-Test	56
6.3 ANOVA: Analysis of Variance	59

A Program Listings	65
A.1 CVSfile	65
A.2 fancierplot.py – a run-able plot	66
A.3 US-UK Population Plot for years 1999–2000	66
A.4 Exploring British and American Petroleum Company Stock Prices	68
A.5 Text Analysis: Shakespeare or Bacon?	71
A.6 Hypothesis Testing: Does the unemployment rate make the President?	73
Bibliography	77
Index	79

List of Program Examples

List of Figures

1.1	Starting iPython on Windows	3
1.2	Running the first plot on Mac OS X	4
1.3	Running the first plot on Windows	5
1.4	Example CSV data from World Economics Dataset	6
2.1	Our first graph—countries’ populations, unsorted	15
2.2	Sorted countries’ populations	16
2.3	A graph generated with X and Y values	18
2.4	Making a fancier plot	20
2.5	US and UK Populations, as two subplots	24
3.1	BP and Exxon-Mobil stock prices in 1990, as histograms	30
4.1	Uniform distribution	37
4.2	Normal distribution	38
5.1	Francis Bacon’s essays with white background, ‘the’ highlighted	43
5.2	Francis Bacon’s essays with black background, ‘the’ highlighted	45
5.3	Comparing ‘the’ patterns in Bacon’s <i>Essays</i> and Shakespeare’s <i>Macbeth</i>	46
5.4	Visualization of all capitalized letters in the <i>Essays of Francis Bacon</i>	50

1 Starting Python and Reading Files

These are the course notes for the class *Computational Freakonomics* which we're teaching at the Georgia Tech Study Abroad program at Oxford University in the Summer 2006. The book *Freakonomics: A rogue economist explores the hidden side of everything* [Levitt and Dubner, 2005] (<http://www.freakonomics.com/>) by Steven D. Levitt and Stephen J. Dubner is a NY Times Bestseller that uses economic methods for studying social questions. In the six weeks of this class, we'll:

- Read and discuss each of the six chapters
- Learn social science methods used in that chapter (led by psychologist Richard Catrambone)
- Learn computer science tools for implementing that method on live data (led by computer scientist Mark Guzdial)

These course notes provide the details on the computational side of the course so that students have examples of syntax and semantics to work from. Why *computational* Freakonomics? Because it's the computational side that makes it interesting. Using computation, we can access real and authentic data—like the data used in *Freakonomics*. Using computation, we can work with many more data points than we could manipulate by hand—just like in *Freakonomics*.

1.1 Starting with Python and SciPy

You'll find the details on getting Python started up at the class website <http://swiki.cc.gatech.edu/CompFreak>. We are going to be using a particular branch of Python, *SciPy*¹ or Scientific Python. SciPy includes *NumPy* for numeric processing and *Matplotlib*² for graphical visualization. There is documentation for all these components available at the course

¹<http://www.scipy.org>

²<http://matplotlib.sourceforge.net/>

website. The combination allows us to work with Freakonomics-sized data and Freakonomics-style analyses.

These course notes are not meant to be a tutorial on Python. It's assumed that:

- Either you had CS1315 or CS1321, and thus met Python already, or
- You know enough about programming from CS1371 that you can pick up Python from these coursenotes and using other documentation linked to <http://www.python.org>. Matplotlib, in particular, was designed to make it easy to pick up if you know Matlab.

Windows

For Windows, your best bet is to use the *Enthought Python* which has all the pieces already in it for scientific processing (such as data analysis and graphing). I³ store Enthought at `C:\Enthought`.

Once you install Enthought Python, you can start it from a *Command Prompt* in Windows by simply typing `python` and hitting return. An option that you have in Enthought, though, is using *iPython*. iPython is especially designed to make it easy to use Matplotlib.

iPython is located in the `Scripts` folder in the Enthought directly, like this:

```
C:\Enthought>cd Scripts
C:\Enthought\Scripts>dir/w
Volume in drive C is IBM_PRELOAD
Volume Serial Number is 20FD-0389

Directory of C:\Enthought\Scripts

[.]
easy_install-2.3-script.py  [..]
easy_install-script.py    easy_install-2.3.exe
endo.py                   easy_install.exe
fsdump.py                 f2py.py
fstail.py                 fsrefs.py
gfplus.bat                fstest.py
ipython                   gfserver.bat
mayavi2                   MayaUI.pyw
nctoh5                    mkzeoinst.py
pildriver.py              pilconvert.py
pilfont.py                pilfile.py
ptdump                    pilprint.py
pypcolor                  ptrepack
repozo.py                 pywin32_postinstall.py
rst2latex.py              rst2html.py
sqlite.exe                runzeo.py
xmlproc_parse             vtkpython.exe
zconfig                   xmlproc_val
zdctl.py                  zconfig_schema2html
zeoctl.py                 zdrun.py
zeopasswd.py              zeopack.py

41 File(s)          760,500 bytes
2 Dir(s)            15,937,982,464 bytes free

C:\Enthought\Scripts>_
```

³When you read “I” in the computational parts, you can presume that it’s Mark Guzdial speaking.

You start iPython, then, by:

- If you choose, you can first `cd` to whatever directory you want to work in. (This isn't absolutely necessary, as we'll see in the next chapter.)
- Starting iPython as an input to python with the argument `--pylab` to get it ready to work with Matplotlib (Figure 1.1).
- Alternatively, if you have the Enthought installation, simply choose iPython from the START-ALL PROGRAMS menu. Then you can `cd` from there to where you want to be. (Be sure to just type `cd` first, to get to your home directory, then `cd` down into your desired directory.)

```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\Mark Guzdial>cd "My Documents\Work\CompFreak"

C:\Documents and Settings\Mark Guzdial\My Documents\Work\CompFreak>python \Entho
ught\Scripts\ipython --pylab
Python 2.3.5 - Enthought Edition 0.9.6 (#62, May 11 2005, 20:02:58) [MSC v.1200
32 bit (Intel)]
Type "copyright", "credits" or "license" for more information.

IPython 0.6.15 -- An enhanced Interactive Python.
? -> Introduction to IPython's features.
%magic -> Information about IPython's 'magic' % functions.
help -> Python's own help system.
object? -> Details about 'object'. ?object also works, ?? prints more.

Welcome to pylab, a matplotlib-based Python environment.
For more information, type 'help(pylab)'.

In [1]: _
```

Figure 1.1: Starting iPython on Windows

Macintosh

Getting a Macintosh Python setup is a very similar process. The closest thing to an Enthought Python on Macintosh is the *ActivePython* available at <http://www.activestate.com/>. Download that and install it.

If you just type `python` into a *Terminal* window (the Macintosh equivalent of a Command Prompt), it will work, but it won't be *ActivePython*. *ActivePython* installs itself slightly differently, so that you run it from `/usr/local/bin/python` like this:

```
Mark-Guzdial-PB-G4-12in:~ guzdial$ whereis python
/usr/bin/python
Mark-Guzdial-PB-G4-12in:~ guzdial$ /usr/local/bin/python
ActivePython 2.4.3 Build 11 (ActiveState Software Inc.) based on
Python 2.4.3 (#1, Apr 3 2006, 18:07:18)
[GCC 3.3 20030304 (Apple Computer, Inc. build 1666)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy
```

Assuming that you're running Mac OS X 10.4 (the latest version), you can download and install the SciPy-Superpack from <http://homepage>.

mac.com/fonnesbeck/mac/index.html. This will include NumPy and Matplotlib.

Testing the Installation

Here's how you test your installation. Start up Python. (On Macs, remember that you'll use `/usr/local/bin/python`.) Load in Matplotlib by typing `from pylab import *` and hitting return. (On Macs, the first time you do all these things, you'll get errors. Ignore them, and keep going. After you restart your computer, you won't see them anymore.) Then do `plot([1,2,3,4])` (or some other numbers. If you're using iPython, the graph will now appear. For everyone else, you'll also have to `show()`, and this should work on both Macs (Figure 1.2) and Windows (Figure 1.3)

It looks something like the below:

```
Python 2.3.5 - Enthought Edition 0.9.6 (#62, May 11 2005, 20:02:58)
[MSC v.1200 32 bit (Intel)] on win32 Type "help", "copyright",
"credits" or "license" for more information.
>>> from pylab import *
>>> plot([1,2.5,3.5,6])
[<matplotlib.lines.Line2D instance at 0x01B26418>]
>>> show()
```

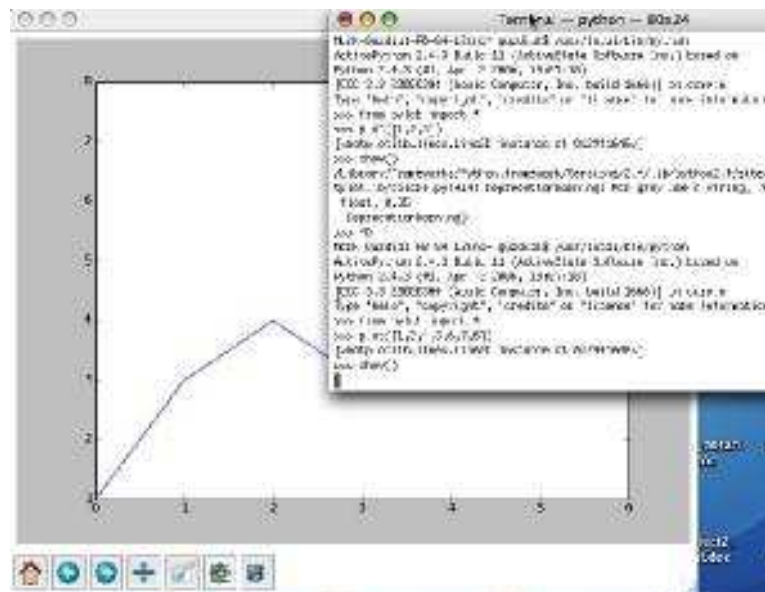


Figure 1.2: Running the first plot on Mac OS X

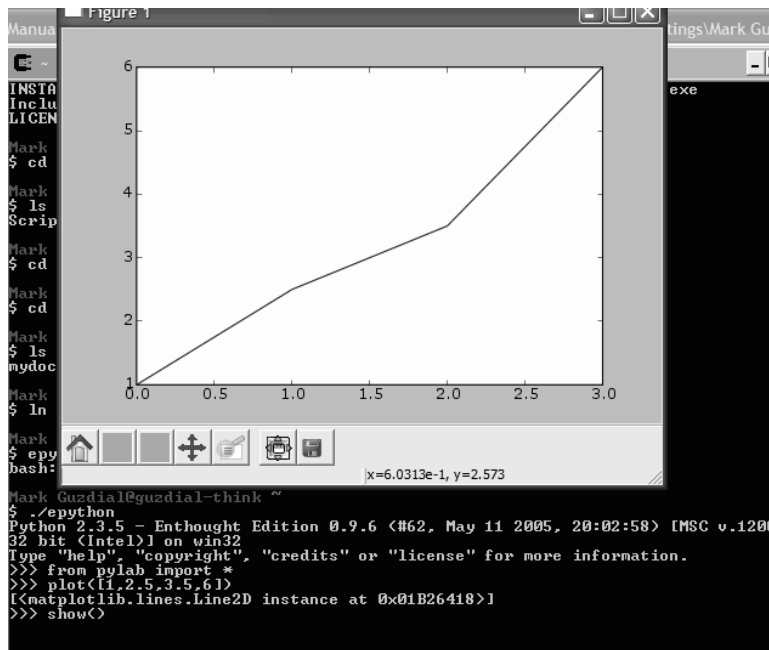


Figure 1.3: Running the first plot on Windows

1.2 Reading Files

It's pretty easy to read and write files in Python.

```

Python 2.3.5 - Enthought Edition 0.9.6 (#62, May 11 2005, 20:02:58)
[MSC v.1200 32 bit (Intel)] on win32 Type "help", "copyright",
"credits" or "license" for more information.
>>> #Writing a file
>>> file = open("SampleFile.txt","wt")
>>> file.write("Here is some text!\n")
>>> #Reading a File
>>> file.close()
>>> newfile = open("SampleFile.txt","rt")
>>> newfile.read()
'Here is some text!\n'
>>> newfile.close()

```

Typically, I keep all my data files and Python files in one directory. Before I start Python, I *cd* (*change directory*) into that directory, and then start python. Then, all my files can be accessed without using long paths to special places on the disk.

We're mostly going to deal with *CSV* (Comma Separated Values) files in this class. Many of the data sources that one might find on the Internet can produce files of this sort. In this example, I generated a data set from the World Economic Dataset at http://pwt.econ.upenn.edu/php_site/pwt_index.php (Figure 1.4). I then copied the text, opened Notepad, pasted the text into a file, then saved it as `somefile.csv`.

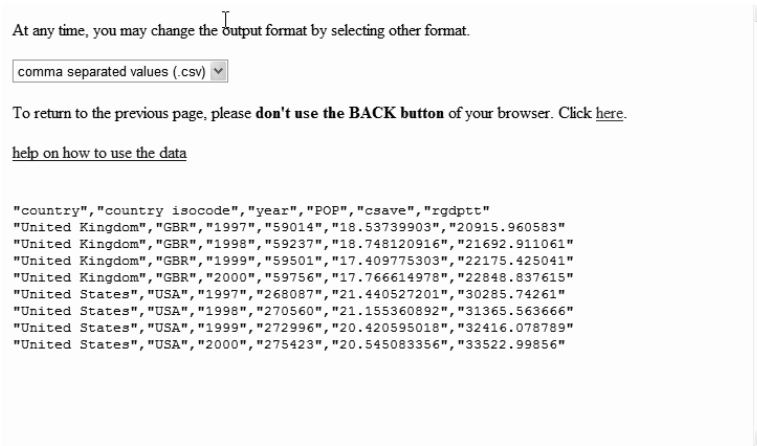


Figure 1.4: Example CSV data from World Economics Dataset

There are tools built in to Python for handling CSV data. You simply type `import csv` and the package is available. You might be wondering “`import csv`? Didn’t we do `from pylab import *` a few minutes ago? What’s the difference?”. When you use `import csv`, you then have to access all the parts of the *module* `csv` with a *dot operator*, e.g., `csv.reader`. When you do `from pylab import *`, you can access the parts of module `pylab` just as if they were global (accessible from anywhere, any object) functions, e.g., `plot([1,2,3])`. Sounds like `from...import` is the best way of doing it, right? That depends on whether you’re *fixing* the code yet. If you have to fix the code, you can update the module in Python by executing `reload(csv)`. If you use `from...import`, you can’t. When you’re developing your own code, it’s often better to `import` so that you can later reload.

Here is the start of a dataset I created of 168 nations’ population in the year 2000.

```
"country","country isocode","year","POP" "Angola","AGO","2000","na"
"Albania","ALB","2000","3411" "Argentina","ARG","2000","37032"
"Armenia","ARM","2000","3803" "Antigua","ATG","2000","68"
"Australia","AUS","2000","19157" "Austria","AUT","2000","8110.2"
"Azerbaijan","AZE","2000","8049"
```

The CSV package knows about how to read individual lines of a CSV file. You open the file, but use that file to create a reader that knows about how to figure out the lines in the file and return the individual pieces in a way that's easily indexable.

```
>>> headerFile = csv.reader(open("pops-2000.csv", "rb"))
>>> headerFile.next()
['country', 'country isocode', 'year', 'POP']
>>> headerFile.next()
['Angola', 'AGO', '2000', 'na']
>>> headerFile.next()
['Albania', 'ALB', '2000', '3411']
>>> line = headerFile.next()
>>> line[0]
'Argentina'
>>> line[1]
'ARG'
>>> line[2]
'2000'
>>> float(line[2]) #converts it to be a number
2000.0
```

But even that's not the easiest way to deal with a CSV file. If you assume that the top line is a list of fieldnames (as is common in well-formed CSV files), then you can use a special `csv.DictReader` to return lines that know what's inside them.

```
>>> headerFile = csv.reader(open("pops-2000.csv", "rb"))
>>> headers = headerFile.next()
>>> headers
['country', 'country isocode', 'year', 'POP']
>>> data = csv.DictReader(open("pops-2000.csv", "rb"), fieldnames=headers)
>>> data.next()
{'country': 'country', 'country isocode': 'country isocode', 'POP':
'POP', 'year ': 'year'}
>>> data.next()
{'country': 'Angola', 'country isocode': 'AGO', 'POP': 'na', 'year':
'2000'}
>>> nextline=data.next()
>>> nextline['country']
'Albania'
>>> nextline['POP']
'3411'
>>> nextline['year']
'2000'
>>> nextline.get('country')
'Albania'
```

What `next()` is returning is a *dictionary*. You can access the dictionary by fieldname, as you can see. You can treat the filenames as indices with square brackets, or using the method `get`.

Using CSVfile

That's actually enough for you to be able to start downloading and playing with data, but I've tried to make it a little easier. I've created a package called `csvfile` (Program Program Example #1) that knows about headers and such and provides arrays of data for analysis.

```
>>> import csvfile
>>> popdata = csvfile.CSVfile("pops-2000.csv")
>>> popdata.headers
['country', 'country isocode', 'year', 'POP']
>>> popdata = csvfile.CSVfile("pops-2000.csv")
>>> usa = popdata.getRows('country isocode', 'USA')
>>> usa
[{'country': 'United States', 'country isocode': 'USA', 'POP':
'275423', 'year':
'2000'}]
>>> usa[0] #just returns the dictionary
{'country': 'United States', 'country isocode': 'USA', 'POP':
'275423', 'year': '2000'}
>>> usa[0]["POP"]
'275423'
```

How do we re-execute lines like `popdata = csvfile.CSVfile("pops-2000.csv")` so easily? Just press up-arrow. That will allow you to see all the lines you've entered. Hit return on the one you want to execute again. You can also use left and right arrow keys to edit the line before re-executing it.

Where `getRows` returns all rows (dictionaries) where the field has that value, there is also a method to return a column of all values of a given field.

```
>>> popdata = csvfile.CSVfile("pops-2000.csv")
>>> pops = popdata.getColumn("POP")
>>> pops[0]
-1
>>> pops[1]
3411.0
>>> pops[2]
37032.0
```

`getColumn` always returns a bunch of numbers. If there is something that isn't a number in the list (say, the field name "POP"), then a default

value of -1 is provided. `getColumn` is a great tool for getting lists of numbers that we might want to plot—see next chapter.

After doing any of these analyses, the CSVfile needs to be *rewound*. To do the analysis, the file gets read. If you want to do a new search through the data file, you need to rewind to the beginning of the file to do a new search⁴.

```
>>> popdata = csvfile.CSVfile("pops-2000.csv")
>>> usaPop = popdata.getRows('country isocode','USA')[0]["POP"]
File "<stdin>", line 1
    usaPop = popdata.getRows('country isocode','USA')[0]["POP"]
                                     ^
SyntaxError: invalid syntax
>>> #Forgot the ending quote!
>>> usaPop = popdata.getRows('country isocode','USA')[0]["POP"]
>>> usaPop
'275423'
>>> csvfile.number(usaPop) #We can use the number converter here
275423.0
>>> popdata.rewind() #Here's the rewind
>>> ausPop = popdata.getRows('country','Australia')[0]["POP"]
>>> ausPop
'19157'
```

1.3 How CSVfile Works

CSVfile will get you started, but at some point, you will have to deal with more complex analyses and data manipulation than it will allow. At that point, you will be writing code *like* CSVfile. It's worthwhile understanding how it works.

CSVfile is written as a *class* from which you create *objects* that understand various *methods* and have various *fields* or *instance variables* associated iwth them.

```
## CSVfile — a front end to CVS
```

```
import csv
```

```
def number(input, default=-1):
    try:
        return float(input)
    except:
        return default
```

⁴You're probably realizing from these examples that # is the commenting character in Python. Everything from the # on is ignored on the same line.

The file starts out importing `csv` since that's necessary for `CSVfile` to work. A general function is defined number that knows how to convert strings to numbers. Since some values are "na" (not applicable or not available) and others are field names, a default value is created that gets returned whenever a non-number is found.

```
class CSVfile:
    def __init__(self, filename):
        self.filename = filename
        self.rewind()

    def rewind(self):
        self.fp = open(self.filename, "rb")
        headerReader = csv.reader(self.fp)
        self.headers = headerReader.next()
        self.dataReader = csv.DictReader(self.fp,
            fieldnames=self.headers)
```

The next part of `CSVfile` is definition of the class `CSVfile` and the initialization method, `__init__`. This is the method that gets called when we first create a `CSVfile`, like `popdata = csvfile.CSVfile("pops-2000.csv")`. You'll notice that all methods in Python start out with the argument of `self`. That's how Python methods get access to the instance of the class that is being accessed with this method call. Even if your method takes no arguments when you use it, you must still include `self` as an argument when you define it.

The `__init__` method simply saves the input filename as a field (*instance variable*) within the instance, `self.filename`. Then the `rewind` method is called. In that method, we open the file (as "rb" which means that it's readable and binary—the `csv` module likes to be able to get at the binary representation, not just the text), read out the field/header names, then create the `DictReader`. Notice that we save the headers in an instance variable so that we can access them later.

```
def next(self):
    return self.dataReader.next()
```

This method allows us to get at individual dictionary rows, if we want, through the `next` method.

```
def getRows(self, fieldname, value):
    ret = []
    for row in self.dataReader:
        if row[fieldname]==value:
            ret.append(row)
    return ret
```

Here's how the `getRows` method works. It takes a `fieldname` (like 'country') and a `value` (like 'Australia') as inputs (and `self`, as always), then returns a *list* of all the dictionaries where that `fieldname` matches that value. In the simple population dataset we're using now, that's simple, but one could

also (for example) have more data and pull out all rows for a given year with this method. We create the list that we will be returning with the line `ret = []`. The square brackets (`[]`) define a list, and here, an empty list (one with nothing in it to start).

The **for** loop in Python is very powerful. You can iterate through all the rows in the dataset with **for** `row` **in** `self.dataReader`—the variable `row` will take on the value of each row in the data. You can use a **for** loop to iterate through just about anything in Python. Here's an example that iterates through a list to **print** each value in the list.

```
>>> for letter in ['a', 'b', 'c']:
...     print letter
...
a
b
c
```

In `getRows`, we iterate through the list, and everywhere that the row dictionary has the fieldname hold the specified value, we append that row to our return value list, `ret`. After iterating through everything, we return the return list.

How might you use this elsewhere? You can use **for** loops to iterate all kinds of data, including data that you gathered from different analyses. You could do a search for all the rows with the year 1990, then all the rows with the year 2000, and then iterate through *each* returned list to get the difference in populations.

```
def getColumn(self, fieldname):
    ret = []
    for row in self.dataReader:
        ret.append(row.get(fieldname))
    return map(number, ret)
```

The `getColumn` method is very similar to `getRows`. Here, we gather *every* value of the specified filename (like 'POP') and put it in the list. But before we return the list, we map the function `number` (from the top of `csvfile`) on to all the values. That's what turns the list of strings (which is what is stored in the CSVfile) to a list of numbers.

2 Plotting

An important part of data analysis is visualizing your data. This chapter describes how to do that.

2.1 Your Basic Plot: Slicing Up The World's Population

To do any plotting, we need to access *Matplotlib* with `from pylab import *`.

The basic command to plot is, not surprisingly `plot`. The function `plot` can take a variety of different kinds of inputs. The most basic is just a sequence—an array or list of all numeric values.

We saw in the previous chapter how to create an array of populations in the year 2000 for 168 countries. We knew that the first element in the list was from the field name "POP", so we can skip that. It turns out that Python has some powerful tools for grabbing *parts* of a sequence. It's called *slicing*.

For any sequence, you can provide indices for the sequence as square brackets with a colon within them. The first value indicates where to start *from* and the second value indicates the index to stop *before*. That's important—the second value is *not* included in the result. The first index in Python is zero—the first value in any Python sequence is numbered zero. If the first value is missing, it's considered to be 0. If the second value is missing, it's considered to be the length of the list.

```
>>> a=[1,2,3,4,5]
>>> a[0]
1
>>> a[1:] # From index 1 (second element) to the end
[2, 3, 4, 5]
>>> a[:3] # From 0 to 2 (not including index 3)
[1, 2, 3]
>>> a[1:3]
[2, 3]
>>> len(a)
5
```

Slicing will work for any sequence, including strings.

```
>>> alpha="abcdefghijklmnopqrstuvwxy"
>>> alpha[0]
'a'
>>> alpha[2:]
'cdefghijklmnopqrstuvwxy'
>>> alpha[14:18]
'opqr'
>>> len(alpha)
26
```

All of this is to explain that the list of populations *skipping* the first value is `pops[1:]`. So, simply put, plotting the populations is `plot(pops[1:])`.

Once you make a plot, you don't see it. You have a choice what to do.

- You can `show()` the plot (Figure ??). The plot window is really nice and allows you to save it, pan around it, zoom into it.
- You can `savefig`.

The function `savefig` is really pretty amazing. You simply give it a filename as input, and it tries to save the file in the format specified by the filename. If your filename ends in `.eps`, it will try to save the plot as Encapsulated Postscript (EPS) (Figure ??). If your filename ends in `.png`, it will try to save the plot in the portable graphics format *PNG*. If your filename ends in `.jpg`, it will try to save the plot in *JPEG* format. (Both PNG and JPEG can be inserted into Microsoft Word documents.) Whether or not it can depends on details of your specific computer (e.g., operating system, whether you have the newest version of all the software, etc.). I can't save in JPEG on my Windows computer with Enthought Python, so I got the `IOError` below. (A *traceback* shows us all the methods or functions currently executing, and at what line, when the error occurs. This error is on purpose it was **raised**. In other cases, a *traceback* can help you debug.)

```
>>> plot(pops[1:])
[<matplotlib.lines.Line2D instance at 0x01ABFE90>]
>>> savefig("populations-unsorted.eps")
>>> savefig("populations-unsorted.jpg")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "C:\Enthought\lib\site-packages\matplotlib\pylab.py", line 839, in s
g
    return fig.savefig(*args, **kwargs)
  File "C:\Enthought\lib\site-packages\matplotlib\figure.py", line 658, in
ig
    self.canvas.print_figure(*args, **kwargs)
  File "C:\Enthought\lib\site-packages\matplotlib\backends\backend_wxagg.py
```

2.1. YOUR BASIC PLOT: SLICING UP THE WORLD'S POPULATION⁵

```
ne 104, in print_figure
    agg.print_figure(filename, dpi, facecolor, edgecolor,
orientation)
File "C:\Enthought\lib\site-packages\matplotlib\backends\backend_agg.py", line
495, in print_figure
    raise IOError('Do not know how to handle extension %s' % ext)
IOError: Do not know how to handle extension *.jpg
>>> savefig("populations-unsorted.png")
>>> show()
```

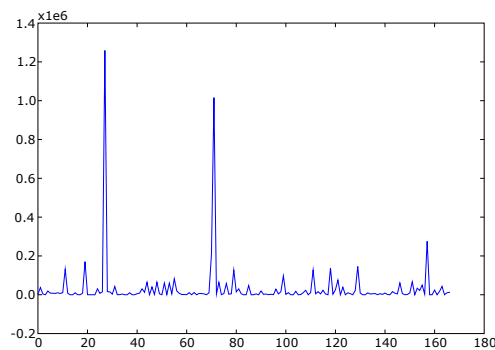


Figure 2.1: Our first graph—countries' populations, unsorted

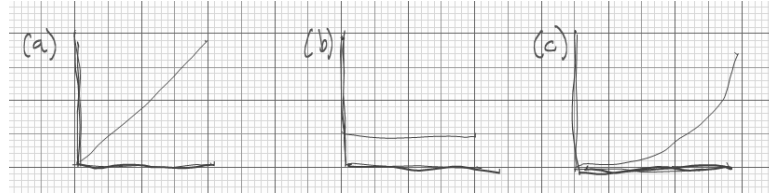
This is a particularly unimpressive graph. The y-axis is obvious—that's populations in thousands. But what's the x-axis? It's countries, in alphabetical order by first name. If you don't provide an x-axis (which you can do, and we'll do it in just a few minutes), the x-axis is assumed to be just the index values of the sequence: 0, 1, 2, and so on.

So let's make the chart a little more interesting. Python knows how to sort any sequence. Let's put the population into sorted order. Our new sequence will start at `-1` (see below) — we know that unavailable or label data maps to `-1`, so we can expect to see at least one of those. The max value is pretty big, 1258821.0314. We then generate the plot, save it, and show it.

```
>>> spops = sort(pops)
>>> spops[0]
-1.0
>>> spops[len(spops)-1]
1258821.0314
>>> plot(spops)
[matplotlib.lines.Line2D instance at 0x01AE5788>]
```

```
>>> savefig("populations-sorted.eps")
>>> show()
```

Before you look at the plot, think about it. What do you expect to see? How do you think that populations *distribute* around the world? Consider these three possibilities.



Option (a) says that all populations are equally likely in the world, so if you plot them from smallest to largest, it's a gradual slope from left to right. Option (b) says that all populations are roughly the same, so the slope of the line is essentially flat. Option (c) says that all population levels occur, but they grow faster than the gradual slope in (a) would suggest—the biggest countries are much bigger than the smaller countries.

Now take a look at the graph, Figure 2.2. None of the three, is it? What this graph seems to be saying is that most of the populations are roughly *the same* and the curve doesn't really start going up until the very end, where it *shoots* up really fast. A few countries are just enormous, while most are (comparatively speaking) about the same size.

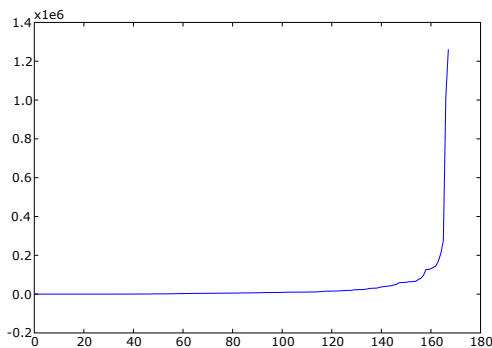


Figure 2.2: Sorted countries' populations

Is that really true? Is that *really* what's going on in this data? One of the nice things about Python's slicing is that you can literally take *slices* through the data with it. What do the first ten values look like? All -1. "Aha!" you may think. "That 'flatness' is just an artifact of populations that we didn't have - NA or Not Available!" Let's take another slice about

half way through (there are 168 values here) – by index 50, the values are *not* -1 . And by the end, they are huge.

```
>>> spops[0:10]
[-1., -1., -1., -1., -1., -1., -1., -1., -1., -1.]
>>> spops[50:80]
[ 1199.    , 1230.    , 1301.    , 1303.    , 1369.    , 1988.    , 2031.    ,
 2035.    ,
   2372.    , 2633.    , 2856.    , 3018.    , 3337.    , 3411.    , 3695.    , 3786.9 ,
   3803.    , 3811.    , 3831.    , 4018.    , 4282.    , 4328.    , 4380.    , 4491.    ,
   4527.    , 4886.81, 4915.    , 5024.    , 5031.    , 5071.    ,]
>>> len(spops)
168
>>> spops[160:168]
[ 131050.    , 138080.    , 145555.008 , 170406.    ,
 210420.992 ,
   275423.    , 1015923.008 , 1258821.0314,]
```

Graphs are obviously darn useful here, but plots alone don't tell us everything. We need to look at some of the numbers. Do we need to look at all the numbers we did above? And did we look at the *right* values above – what if the values *all the way* from 0 to 49 are -1 ? Would that change your opinion about the graph? In this class, we're going to learn about a variety of techniques for describing values, to get a sense of what the data are doing in a set and how they relate to other data. Graphing is really useful, but it's only one way to look at the data.

2.2 Options on the Plot

The plot method has lots of ways that it can be used. One is that you can pass in two sequences as arguments—one containing the Y values, and the other X axis values (Figure 2.3). Here, we use `arange` to generate a bunch of *floating point numbers* (not *integers* but numbers with a decimal place) between 0 and 3, spaced out 0.05 apart. We then generate another array, `s`, by using a special version of `sin` that iterates over the array `t` and generates a new array element for `s`. There's a loop there, but it's hidden inside of `sin`. It's called a *universal function* (or *ufunc*), and they're documented in the *Numeric Python* documentation.

```
>>> t = arange(0.0, 3.0, 0.05)
>>> t[0:5]
[ 0.    , 0.05, 0.1 , 0.15, 0.2 ,]
>>> s = sin(2*pi*t)
>>> s[0:5]
[ 0.          , 0.30901699, 0.58778525, 0.80901699, 0.95105652,]
>>> plot(t,s)
```

```
[<matplotlib.lines.Line2D instance at 0x00C1BE18>]
>>> savefig("C:/temp/sinplot.eps")
>>> show()
```

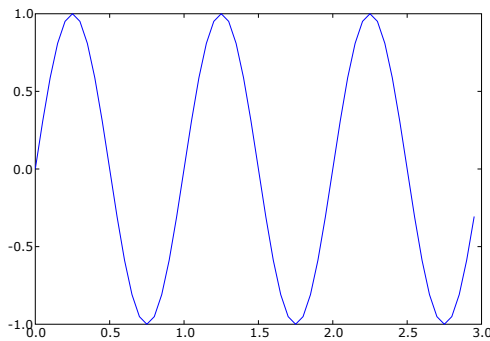


Figure 2.3: A graph generated with X and Y values

Generating plots from files

You don't really want to type in all those commands at the command prompt each time you want a plot. Instead, it's easier to put these commands in a file like this (see also Program Program Example #2):

```
from pylab import *
import csvfile
popdata = csvfile.CSVfile("pops-2000.csv")
pops = popdata.getColumn("POP")
spops=sort(pops)

plot(spops[1:],marker="o",color="r")
title('Populations of countries in the year 2000')
xlabel('Countries in increasing order of population')
ylabel('Population in millions')
grid(True)
show()
```

This “file” (“program”?) is doing the necessary **import** commands, setting up the data, then generating the plot (Figure 2.4). You can use just about any text editor for creating this file—Word might work, but Notepad would be better. I like to use *WinEdt* which is a really nice editor for text on Windows. There are also editors like emacs and vi that you can use. Just make sure that the filename always ends in ‘.py’ to stand for Python.

You'll notice that we're also doing a lot more tweaking to this graph.

- We can label the X-axis and Y-axis with `xlabel` and `ylabel`.
- We can title the whole graph with `title`.
- We can define a marker for the line. Markers can be `'+'', 'o'', 's'', 'v'', 'x'', 'i'',` or `'^'`.
- We can define a color for the line. Here we're using `'r'` for *red*. Colors can be:

<code>b</code>	blue
<code>g</code>	green
<code>r</code>	red
<code>c</code>	cyan
<code>m</code>	magenta
<code>y</code>	yellow
<code>k</code>	black (go figure)
<code>w</code>	white
<code>(0.25,0.35,0.5)</code>	An RGB triplet (<i>tuple</i>) where the scale is 0..1
<code>red</code>	Any HTML color name

- Not used here, we can also specify a linestyle: `'-''', '-.',` or `'--'`.
- We can use the same color parameters to specify a `markeredgecolor` and `markerfacecolor` and even `markersize` (in points) if we wanted.

Now, how to run this code. In *IPython*, it's easy. There are commands to `cd` to the right directory (where you put the file) and then run the file.

```
In [2]: cd C:/Documents\ and\ Settings/Mark\ Guzdial/My\
Documents/Work/CompFreak
```

```
In [3]: run fancierplot.py
```

In other forms of Python, you need to *import* the file. If you change the file (to fix a bug, to generate a slightly different plot), you can *reload* the file to re-execute it.

```
Python 2.3.5 - Enthought Edition 0.9.6 (#62, May 11 2005, 20:02:58)
[MSC v.1200 32 bit (Intel)] on win32 Type "help", "copyright",
"credits" or "license" for more information.
>>> import fancierplot
```

2.3 The Plot Thickens: Combining Plots to Determine US and UK Growth Rates

In work in *Freakonomics*, you will often want to compare multiple plots at once. The easiest way to do this is by putting both lines that you care about on the same plot.

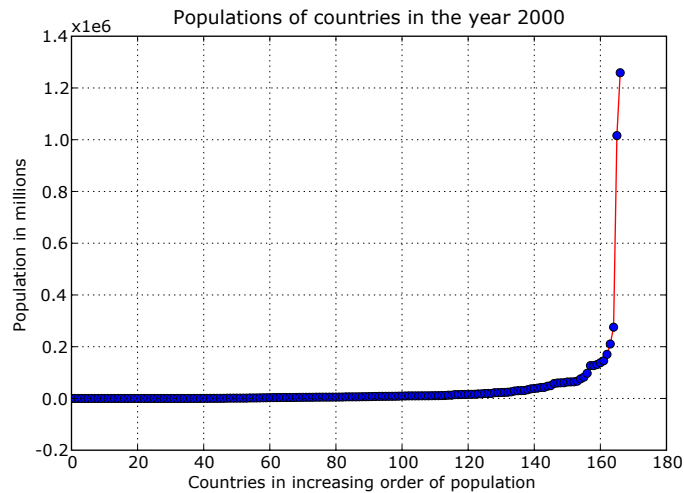


Figure 2.4: Making a fancier plot

I generated a dataset with all the populations (and savings rates, and other fields) for the US and the UK from 1990 to 2000. Here's a segment of what the file looks like:

```
"country","country isocode","year","POP","XRAT","csave","rgdptt"
"United Kingdom","GBR","1999","59501","0.6181","17.409775303","22175.425041"
"United Kingdom","GBR","2000","59756","0.6609","17.766614978","22848.837615"
"United States","USA","1990","249981","1","18.785790541","26365.46609"
"United States","USA","1991","252677","1","18.4248996","25893.640342"
```

We can write code to withdraw the relevant data from this file, and then plot it.

```
from pylab import *
import csvfile
natdata = csvfile.CSVfile("us-uk-1990-2000.csv")
usdata = natdata.getRows('country','United States')
natdata.rewind()
ukdata = natdata.getRows('country','United Kingdom')

#Get the populations
uspops = []
for row in usdata:
    uspops.append(csvfile.number(row['POP']))
ukpops = []
for row in ukdata:
    ukpops.append(csvfile.number(row['POP']))
years=range(1990,2001)
print "US",uspops,len(uspops)
print "UK",ukpops,len(ukpops)
```

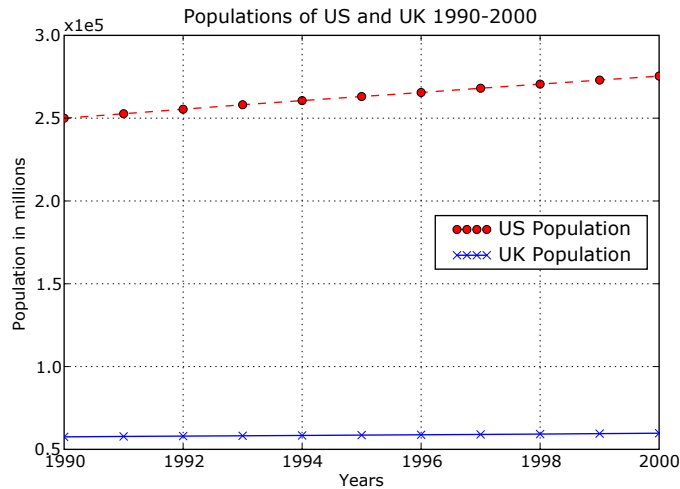
```
print "Years", years, len(years)

plot(years, uspops, 'r—o', years, ukpops, 'b—x')
legend(('US Population', 'UK Population'), loc='center right')
title('Populations of US and UK 1990–2000')
xlabel('Years')
ylabel('Population in millions')
grid(True)
savefig("us_uk_pop_plot.eps")
show()
```

How it works: We open the datafile, grab the US data, rewind it, then grab the UK data. We make an assumption that the data is still in year order – that could have been a bad assumption, and there are other ways to grab the data so that we don't have to assume that. The data in usdata and ukdata are now in row/dictionary form. To get just the population ('POP') data, we create lists with those values, converted to numbers. (Notice that I check my results with **print** statements—it's okay to have them in your file, and they'll work.)

We then plot with X (year), and Y (population) data, in the same plot command (Figure ??). You'll note that we can specify the line color, line style, and marker in a string next to the relevant plot.

From this plot, it looks like the US population has been rising steeply, while the UK population has been essentially flat. That's what it *looks like*, but we'll see later that there are other ways to look at it.



Obviously, the legend function generates a legend. The legend function doesn't *have* to have a loc (location) parameter. I found that the default (so-called 'best') location is the upper-right, which covered over the US

population curve. So I changed the location. How did I figure out *how* to change the legend location? It's not in the Matplotlib documentation (that I could find). Turns out that there's even more documentation within Python using the help function.

```
>>> help(legend)
```

```
Help on function legend in module matplotlib.pyplot:
```

```
legend(*args, **kwargs)
```

```
LEGEND(*args, **kwargs)
```

```
Place a legend on the current axes at location loc. Labels are a
sequence of strings and loc can be a string or an integer specifying
the legend location
```

```
USAGE:
```

```
Make a legend with existing lines
```

```
>>> legend()
```

```
legend by itself will try and build a legend using the label
property of the lines/patches/collections. You can set the label of
a line by doing plot(x, y, label='my data') or line.set_label('my
data'). If label is set to '_nolegend_', the item will not be shown
in legend.
```

```
# automatically generate the legend from labels
```

```
legend( ('label1', 'label2', 'label3') )
```

```
# Make a legend for a list of lines and labels
```

```
legend( (line1, line2, line3), ('label1', 'label2', 'label3') )
```

```
# Make a legend at a given location, using a location argument
```

```
# legend( LABELS, LOC ) or
```

```
# legend( LINES, LABELS, LOC )
```

```
legend( ('label1', 'label2', 'label3'), loc='upper left')
```

```
legend( (line1, line2, line3), ('label1', 'label2', 'label3'), loc
```

```
The location codes are
```

```
'best' : 0,
```

```
'upper right' : 1, (default)
```

```
'upper left' : 2,
```

```
'lower left' : 3,
```

```
'lower right' : 4,
```

```
'right' : 5,
```

```
'center left' : 6,
```

```
'center right' : 7,
```

```
'lower center' : 8,
```

```
'upper center' : 9,
```

```
'center' : 10,
```

```
If none of these are suitable, loc can be a 2-tuple giving x,y
in axes coords, ie,
```

```
loc = 0, 1 is left top
```

```
loc = 0.5, 0.5 is center, center
```

As Two Separate Plots

The problem with the legend points out that, sometimes, it doesn't work out well to have the two (or more) lines in the same graph. If the X axes are the same, you can do vertical plots for the same effect. The graphs can be compared, but without having to stick to the same Y axis.

The below program does that using the subplot function (Figure ??). The subplot specifies how many rows and columns of plots you want (first two arguments) and then which one you're specifying now¹ You'll also see in this program that we add an additional loop to make sure that we're calling up the right year in the right order. This will be important when we try to join data later.

```

from pylab import *
import csvfile
natdata = csvfile.CSVfile("us-uk-1990-2000.csv")
usdata = natdata.getRows('country', 'United States')
natdata.rewind()
ukdata = natdata.getRows('country', 'United Kingdom')

#Get the populations
# This time, making SURE that they're in year-order
years=range(1990,2001)
uspops = []
for y in years:
    for row in usdata:
        if row['year']==str(y): #Items in rows are strings
            uspops.append(csvfile.number(row['POP']))
            break #Leave the row loop
ukpops = []
for y in years:
    for row in ukdata:
        if row['year']==str(y):
            ukpops.append(csvfile.number(row['POP']))
            break

# Top subplot: 2 rows, 1 column, subplot #1
subplot(2,1,1)
plot(years,uspops,'r-o')
title('Population of US 1990-2000')
xlabel('Years')
ylabel('Population in millions')
grid(True)

subplot(2,1,2)
plot(years,ukpops,'b-x')
title('Population UK 1990-2000')

```

¹What would happen if you changed the rows and columns between subplot calls? Dunno.

```

xlabel('Years')
ylabel('Population in millions')
grid(True)

savefig("us_uk_pop_plot2.eps")
show()

```

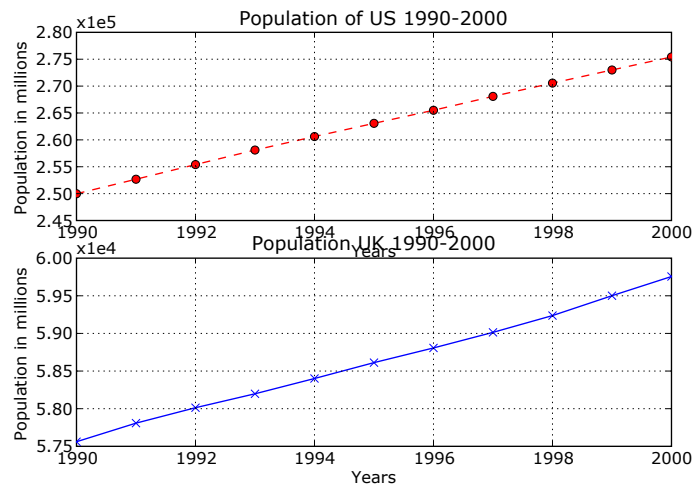


Figure 2.5: US and UK Populations, as two subplots

Notice something important about this double plot. Sure looks like the *slope* of each line *is about the same!* There are ways to check that assumption later, but it looks like both the US and UK plot have been increasing at very similar rates, but we only see that if we shift the scales appropriately to see how each is increasing.

3 Descriptive Statistics

The way in which we usually describe sets of numbers is with *descriptive statistics*—numbers that reflect the overall picture, range, or distribution of the set.

3.1 Average or mean: Petroleum Tax Prices

The average or mean is simply the sum of the values divided by the number of values.

Let's compute the value of a stock over a given year. From Yahoo Stocks, we can get the monthly value of a stock over some period of time. The below is some of the values for British Petroleum (*BP*).

```
Date,Open,High,Low,Close,Volume,Adj. Close*
1-Jun-06,69.61,72.38,66.20,67.48,4938385,67.48
1-May-06,74.25,76.67,68.50,70.70,3859318,70.70
3-Apr-06,69.50,76.85,69.49,73.72,3520315,73.18
1-Mar-06,66.92,70.68,65.35,68.94,2938130,68.43
1-Feb-06,71.99,72.58,66.01,66.42,3647978,65.93
3-Jan-06,65.50,72.88,65.47,72.31,4301770,71.19
1-Dec-05,67.06,69.25,63.26,64.22,2983219,63.23
```

We can access this data just as we have any other CSV data.

```
In [11]: import csvfile

In [12]: bpdata=csvfile.CSVfile("BritishPetroleum-BP-table.csv")

In [13]: bpdata.next() Out[13]: {'Adj. Close*': '67.48',
'Close': '67.48',
'Date': '1-Jun-06',
'High': '72.38',
'Low': '66.20',
'Open': '69.61',
'Volume': '4938385'}

In [14]: bpdata.next()['Date'] Out[14]: '1-May-06'
```

There are builtin functions to sum across a sequence, and to get the length of a sequence (the number of values there). We can use these to

define an average function. Notice that we multiply by 1.0 to force Python to do floating point arithmetic, rather than simple integer arithmetic.

```
In [15]: a=[1,2,3,4]
```

```
In [16]: sum(a)
Out[16]: 10
```

```
In [17]: len(a)
Out[17]: 4
```

```
In [18]: sum(a)/len(a)
Out[18]: 2
```

```
In [19]: (sum(a)*1.0)/len(a)
Out[19]: 2.5
```

```
from pylab import *
import csvfile

def average(sequence):
    return (1.0*sum(sequence))/len(sequence)

bpdata = csvfile.CSVfile("BritishPetroleum-BP-table.csv")

#Let's get the 1990 year.
closes = []
for row in bpdata.dataReader:
    if row['Date'].endswith('90'):
        closes.append(csvfile.number(row['Close']))

#Return the average
print "Closing values",closes
print "Average:",average(closes)
```

How it works: We import pylab and csvfile, as we have in the past. We define a function average which is fine to do in-line. We then open up the file and do a search for all those dates that end in '90' (in order to get the average of the 1990 monthly closing dates). Notice that our loop executes over the dataReader. That's how we did it in csvfile.py.

```
In [26]: run bpAvg1990.py
Closing values [76.870000000000005, 80.25, 77.5, 77.25, 82.120000000000005, 74.5
, 66.5, 66.620000000000005, 60.130000000000003, 64.75, 68.5, 68.75]
Average: 71.9783333333
```

3.2 Standard Deviation

Standard deviation is the amount of *spread* in the values in the data set. If all the values in the data set are exactly the same, then they're all equal to the mean value, and the standard deviation is zero. The higher the standard deviation, the further values differ from the mean.

The standard deviation is the square root of the *variance*. It's the average of the squared differences from the mean. Here's the basic process for figuring out the standard deviation.

1. First, compute the mean.
2. Figure out the difference between each value and the mean, e.g., $x_i - \text{mean}$ for all positions i in the data sequence. Then *square* that distance. The square removes the possibility of a negative value, since you don't know which is bigger, x_i or the mean.
3. Sum up all these squared differences, then divide by the number of numbers in the sequence. This is called the *average of the squared differences*, or the *variance*.
4. Finally, take the square root of the whole thing. The idea is to get close to the average of the differences, with the squared-differences and square root in there to deal with positive and negative values.

Let's see a program that implements the standard deviation algorithm. In this program, we gather data from both British Petroleum (BP), but also Exxon-Mobil (*XOM* on Yahoo).

```

from pylab import *
import csvfile

def average(sequence):
    return (1.0*sum(sequence))/len(sequence)

def std_dev(sequence):
    ave = average(sequence)
    # Compute the mean squared difference
    diffs = 1.0
    for num in sequence:
        diffs = diffs + pow((ave-num),2)
    # Compute the variance
    variance = diffs/len(sequence)
    # Return the square root of the variance
    return pow(variance,0.5)

bpdata = csvfile.CSVfile("BritishPetroleum-BP-table.csv")

```

```

#Let's get the 1990 year.
closes = []
for row in bpdata.dataReader:
    if row['Date'].endswith('90'):
        closes.append(csvfile.number(row['Close']))

#Return the average
print "*** BP ***"
print "Closing values", closes
print "Average:", average(closes)
print "Standard Deviation:", std_dev(closes)

amdata = csvfile.CSVfile("Exxon-Mobil-XOM-table.csv")

#Let's get the 1990 year.
closes = []
for row in amdata.dataReader:
    if row['Date'].endswith('90'):
        closes.append(csvfile.number(row['Close']))

#Return the average
print "*** Exxon/Mobil ***"
print "Closing values", closes
print "Average:", average(closes)
print "Standard Deviation:", std_dev(closes)

```

The function `pow` takes the *power* of one number to another number. It works for squaring (`pow(something,2)`) and for getting square roots (`pow(something,0.5)`). And here's the run of it.

```

In [30]: run bpStdDev1990.py
*** BP ***
Closing values [76.870000000000005, 80.25, 77.5, 77.25, 82.120000000000005,
, 66.5, 66.620000000000005, 60.130000000000003, 64.75, 68.5, 68.75]
Average: 71.9783333333
Standard Deviation: 6.67530877355
*** Exxon/Mobil ***
Closing values [51.75, 50.630000000000003, 49.0, 49.0, 50.0, 51.8800000000
47.880000000000003, 48.0, 45.25, 46.25, 47.0, 47.0]
Average: 48.6366666667
Standard Deviation: 2.05769018292

```

Okay, so on average, BP stock had a higher close in 1990 than Exxon-Mobil, but BP also had a higher standard deviation. It varied more over that year than Exxon did. Does that matter? Is it *really* different? And what does “*really different*” mean, anyway?

3.3 Histogram

Another way of getting a picture of what's happening with a data set is to use a *histogram*. A histogram tells you the number of occurrences of values in a certain range. "Hold on!" you say. "I don't see that in the Matplotlib documentation!" Yet again, you need to use help.

```
In [9]: from pylab import *
```

```
In [10]: help(hist)
```

```
Help on function hist in module matplotlib.pylab:
```

```
hist(*args, **kwargs)
    HIST(x, bins=10, normed=0, bottom=0, orientation='vertical', **kwargs)
    Compute the histogram of x. bins is either an integer number of
    bins or a sequence giving the bins. x are the data to be binned.
    The return values is (n, bins, patches)
    If normed is true, the first element of the return tuple will
    be the counts normalized to form a probability density, ie,
    n/(len(x)*dbin)
    orientation = 'horizontal' | 'vertical'. If horizontal, barh
    will be used and the "bottom" kwarg will be the left.
    width: the width of the bars. If None, automatically compute
    the width.
    kwargs are used to update the properties of the
    hist bars
```

```
    Addition kwargs: hold = [True|False] overrides default hold state
```

Here's an example that generates a histogram for each of the BP and Amoco-Mobil stock.

```
from pylab import * import csvfile
```

```
def average(sequence):
    return (1.0*sum(sequence))/len(sequence)
```

```
def std_dev(sequence):
    ave = average(sequence)
    # Compute the mean squared difference
    diffs = 1.0
    for num in sequence:
        diffs = diffs + pow((ave-num),2)
    # Compute the variance
    variance = diffs/len(sequence)
    # Return the square root of the variance
    return pow(variance,0.5)
```

```

bpdata = csvfile.CSVfile("BritishPetroleum-BP-table.csv")

#Let's get the 1990 year. closes = [] for row in bpdata.dataReader:
    if row['Date'].endswith('90'):
        closes.append(csvfile.number(row['Close']))

subplot(2,1,1) title("BP stock in 1990--Histogram") hist(closes)

amdata = csvfile.CSVfile("Exxon-Mobile-XOM-table.csv")

#Let's get the 1990 year. closes = [] for row in amdata.dataReader:
    if row['Date'].endswith('90'):
        closes.append(csvfile.number(row['Close']))

subplot(2,1,2) title("Amoco/Mobil stock in 1990--Histogram")
hist(closes)

savefig("BP_AM.hist.eps") show()

```

The result is Figure 3.1. This helps some, like showing that BP basically had two common prices during this time, while Exxon-Mobil is more disperse. We call that the *shape* of the distribution.

So, do you think BP and Exxon-Mobil roughly track one another? That is, do they move up or down in the same ways? If they do, one would presume that the impacts on their prices have more to do with external factors (e.g., peace in the Middle East) than with anything in the companies themselves or in the UK or US, respectively. How would we find out? See next chapter. . .

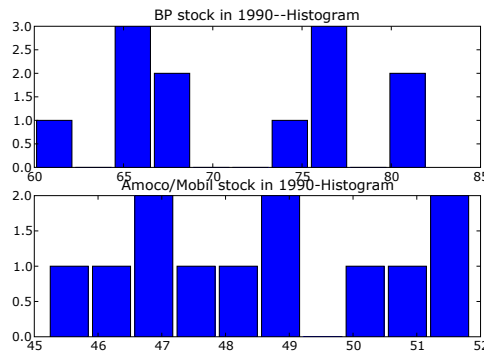


Figure 3.1: BP and Exxon-Mobil stock prices in 1990, as histograms

4 Correlation

How do we *compare* two sequences of values? How do we figure out if BP and Exxon-Mobile change in roughly the same ways at roughly the same times (suggesting that factors external to either company are acting upon both at the same time)? How do we figure out if the UK and US populations grow and shrink at about the same rate (suggesting that whatever factors influence the size of populations are impacting both countries at the same time in the same way)? One way of doing that is with a *correlation*. A correlation is a number that describes how related two data sets are.

4.1 Computing correlation: Is it the company, or war in the Middle East?

Let's call one data set x and the other data set y . Each data set should have the same number of elements, call it n . The correlation number is called r . Here's the formula for r .

$$r = \frac{(n \sum_{i=0}^n x_i y_i) - (\sum_{i=0}^n x_i)(\sum_{i=0}^n y_i)}{\sqrt{(n \sum_{i=0}^n x_i^2 - (\sum_{i=0}^n x_i)^2)(n \sum_{i=0}^n y_i^2 - (\sum_{i=0}^n y_i)^2)}} \quad (4.1)$$

Let's talk our way through that mess.

- In the numerator, $(n \sum_{i=0}^n x_i y_i)$ is the sum of each pair of x_i and y_i multiplied together. We subtract from that the product of the sum of all the x values $(\sum_{i=0}^n x_i)$ and the sum of all the y values $(\sum_{i=0}^n y_i)$. So, the numerator is this sing-song term: the sum of the products of the scores less the product of the sums of the scores.
- In the denominator, it's the square root of a product. The first term in the product is n times the sum of all x values squared *minus* the square of the sum of all x values $(n \sum_{i=0}^n x_i^2 - (\sum_{i=0}^n x_i)^2)$. The second term in the product is the y version of that $(n \sum_{i=0}^n y_i^2 - (\sum_{i=0}^n y_i)^2)$.

While that's messy as one might imagine, it turns out to be pretty straightforward to convert that into Python code.

```
def correlation(x,y):
    n = len(x)
```

```

if n != len(y):
    print "Uh-oh! x and y must be paired values!"
    return 0.0
# Compute the numerator
prod_pairs = 0
for i in range(0,n):
    prod_pairs = prod_pairs + (x[i]*y[i])
numerator = n*prod_pairs - (sum(x)*sum(y))
# Compute the denominator
x_square = 0
for i in range(0,n):
    x_square = x_square + pow(x[i],2)
y_square = 0
for i in range(0,n):
    y_square = y_square + pow(y[i],2)
denom_term1 = ((n*x_square)-pow(sum(x),2))
denom_term2 = ((n*y_square)-pow(sum(y),2))
denominator = pow((denom_term1*denom_term2),0.5)
return numerator/denominator

```

How it works: Computing n is easy—it's the length. We want to make sure that the two lengths are equal, else the values couldn't possibly be *paired*. The sum of the product of the pairs is just what it sounds like: for all index values i , $\text{prod_pairs} = \text{prod_pairs} + (x[i]*y[i])$. The numerator is then $\text{numerator} = n*\text{prod_pairs} - (\text{sum}(x)*\text{sum}(y))$. The sum of the x 's squared is for all index values i , $\text{x_square} = \text{x_square} + \text{pow}(x[i],2)$. The y _square is computed in the same way. The denominator

$$\sqrt{(n \sum_{i=0}^n x_i^2 - (\sum_{i=0}^n x_i)^2)(n \sum_{i=0}^n y_i^2 - (\sum_{i=0}^n y_i)^2)}$$

is then computed with the code:

```

denom_term1 = ((n*x_square)-pow(sum(x),2))
denom_term2 = ((n*y_square)-pow(sum(y),2))
denominator = pow((denom_term1*denom_term2),0.5)

```

The correlation is then $\text{numerator}/\text{denominator}$.

Example: Correlating British vs. American Petroleum Stock Prices

Let's look again at our BP vs. Exxon-Mobil petroleum stock prices in 1990. Are these two data sets highly correlated?

```

from pylab import *
import csvfile

def average(sequence):
    return (1.0*sum(sequence))/len(sequence)

def std_dev(sequence):
    ave = average(sequence)

```

4.1. COMPUTING CORRELATION: IS IT THE COMPANY, OR WAR IN THE MIDDLE EAST?

33

```
# Compute the mean squared difference
diffs = 1.0
for num in sequence:
    diffs = diffs + pow((ave-num),2)
# Compute the variance
variance = diffs/len(sequence)
# Return the square root of the variance
return pow(variance,0.5)

def correlation(x,y):
    n = len(x)
    if n != len(y):
        print "Uh-oh! x and y must be paired values!"
        return 0.0
    # Compute the numerator
    prod_pairs = 0
    for i in range(0,n):
        prod_pairs = prod_pairs + (x[i]*y[i])
    numerator = n*prod_pairs - (sum(x)*sum(y))
    # Compute the denominator
    x_square = 0
    for i in range(0,n):
        x_square = x_square + pow(x[i],2)
    y_square = 0
    for i in range(0,n):
        y_square = y_square + pow(y[i],2)
    denom_term1 = ((n*x_square)-pow(sum(x),2))
    denom_term2 = ((n*y_square)-pow(sum(y),2))
    denominator = pow((denom_term1*denom_term2),0.5)
    return numerator/denominator

bpdata = csvfile.CSVfile("BritishPetroleum-BP-table.csv")

#Let's get the 1990 year.
bpcloses = []
for row in bpdata.dataReader:
    if row['Date'].endswith('90'):
        bpcloses.append(csvfile.number(row['Close']))

amdata = csvfile.CSVfile("Exxon-Mobile-XOM-table.csv")

#Let's get the 1990 year.
amcloses = []
for row in amdata.dataReader:
    if row['Date'].endswith('90'):
        amcloses.append(csvfile.number(row['Close']))
```

```

print "BP closing values:", bpcloses
print "average", average(bpcloses)
print "number", len(bpcloses)
print "standard deviation", std_dev(bpcloses)

print "Exxon-Mobil (American) closing values:", amcloses
print "average", average(amcloses)
print "number", len(amcloses)
print "standard deviation", std_dev(amcloses)

print "Correlation is ", correlation(bpcloses, amcloses)

```

And here's what the run looks like:

```

In [9]: run bpAmCorrel1990.py
BP closing values: [76.870000000000005, 80.25, 77.5, 77.25, 82.120000000000005,
74.5, 66.5, 66.620000000000005, 60.130000000000003, 64.75, 68.5, 68.75]
average 71.9783333333
number 12
standard deviation 6.67530877355
Exxon-Mobil (American) closing values: [51.75, 50.630000000000003, 49.0, 49.0, 5
0.0, 51.880000000000003, 47.880000000000003, 48.0, 45.25, 46.25, 47.0, 47.0]
average 48.6366666667
number 12
standard deviation 2.05769018292
Correlation is 0.819128769403

```

This correlation r is called the *Pearson Product Moment Correlation*. It has values between -1.0 and 1.0 . A negative value suggests a negative correlation—when x goes up, y goes down. A positive value suggests a positive correlation—both values go up at about the same time. A value of 0.82 is pretty darn close to 1.0 , so that suggests a strong positive relation.

But is it significant?

But maybe 1990 was a bad year. Maybe if we had looked at 1991 or 1989, we wouldn't have seen any correlation at all, or at least, a much weaker one. What we want to do is consider the probability that what we observed was just luck. We call this a *significance chance*. Basically, the more values that you have and the stronger the correlation, the more confidence that you can have that the result is *significant*.

The correlation r and the number of pairs are only two of the three variables that you need to determine significance. The last value is the *significance level*, also called the *alpha value*. How sure do you want to be? A common alpha value is 0.05 . That means that only 5 of 100 experiments with data samples from these variables would be wrong. If this were medical research, we might want an alpha value of 0.01 or even 0.10 . (In Education research, we can sometimes get away with 0.10 .)

There is another factor that we're not going to talk about much here, and that's whether you're doing a *one-tailed* or *two-tailed* test. The first means "Are we only testing for one value being *greater* than the other?"

where the second one means “Are we testing for *any difference* between the two?” We’ll go with two-tailed for now.

The number of pairs is called the *degrees of freedom*¹. The degrees of freedom for a correlation is $n - 2$.

So, for our data set, we have 10 degrees of freedom (because we have 12 pairs and $12 - 2$ is 10), and we’ll use an alpha value of 0.05. Now we need an ever-present correlation table. There are a bunch linked to the class Swiki. The one I’m using is at <http://www.gifted.uconn.edu/siegle/research/Correlation/corrchrt.htm>. The value at *df* 10 and alpha value 0.05 is 0.576. That means that if our r is less than -0.576 it’s a significant negative correlation, and if it’s greater than 0.576, it’s a significant positive correlation.

Since our r value is 0.82, we have a significant relationship. Our inference, then, is that whatever factors influenced the stock price of BP and Exxon-Mobil in 1990, they were mostly the *same* factors, since the stock prices are highly correlated. This doesn’t mean that there is a *causal* relationship—just because BP went down, Exxon went down, nor vice-versa. This doesn’t say anything about causation at all. But it does say that there is a *relationship*. How would you find out what factors *are* leading to that relationship? Hmm, good question—one for a future chapter.

4.2 But do we believe it?

I don’t know about you, but I find the correlation computation to be a bit of mumbo-jumbo – statistical voodoo. How do we know that it’s really telling us anything? Let’s do some experiments to find out.

I want to get access to our correlation function, so I put it in a file `correlation.py`.

```
from pylab import *

def correlation(x,y):
    n = len(x)
    if n != len(y):
        print "Uh-oh! x and y must be paired values!"
        return 0.0
    # Compute the numerator
    prod_pairs = 0
    for i in range(0,n):
        prod_pairs = prod_pairs + (x[i]*y[i])
    numerator = n*prod_pairs - (sum(x)*sum(y))
    # Compute the denominator
    x_square = 0
    for i in range(0,n):
        x_square = x_square + pow(x[i],2)
    y_square = 0
```

¹These statisticians have names for *everything*!

```

for i in range(0,n):
    y_square = y_square + pow(y[i],2)
    denom_term1 = ((n*x_square)-pow(sum(x),2))
    denom_term2 = ((n*y_square)-pow(sum(y),2))
    denominator = pow((denom_term1*denom_term2),0.5)
return numerator/denominator

```

That way, I can just import it and use it. First test: Are two identical sets of numbers “correlated”?

```
In [10]: from correlation import *
```

```
In [11]: correlation([1,2,3,4,5,6,7,8,9,10],[1,2,3,4,5,6,7,8,9,10])
Out[11]: 1.0
```

That’s good. I would expect two identical sets of numbers to be correlated as strongly as they possibly could be.

But maybe it’s easier with small sets of integers. Let’s do something more complicated. There is a library in NumPy that can create random arrays. Here’s how we create an array of 20 random numbers.

```
In [13]: from RandomArray import *
```

```
In [14]: x = random((20,))
```

```
In [15]: x[0]
Out[15]: 0.0037765550990622415
```

```
In [16]: x[1]
Out[16]: 0.55916408054947242
```

```
In [17]: x[19]
Out[17]: 0.36556442411289364
```

Let’s make it larger – 1000. With 1000 values, we can start to see the shape of the distribution from the histogram (Figure 4.1). All values between 0.0 and 1.0 are equally likely. The distribution has a few bumps, but overall, it’s flat.

```
In [12]: x=random((1000,))
```

```
In [13]: hist(x)
Out[13]:
([ 90, 96,126,107, 94, 95, 77,114,111, 90,],
 [ 2.08577149e-004, 1.00040727e-001, 1.99872877e-001, 2.99705027e-001,
   3.99537176e-001, 4.99369326e-001, 5.99201476e-001, 6.99033626e-001,
   7.98865776e-001, 8.98697926e-001,],
 <a list of 10 Patch objects>)
```

```
In [14]: savefig("uniform_hist.eps")
```

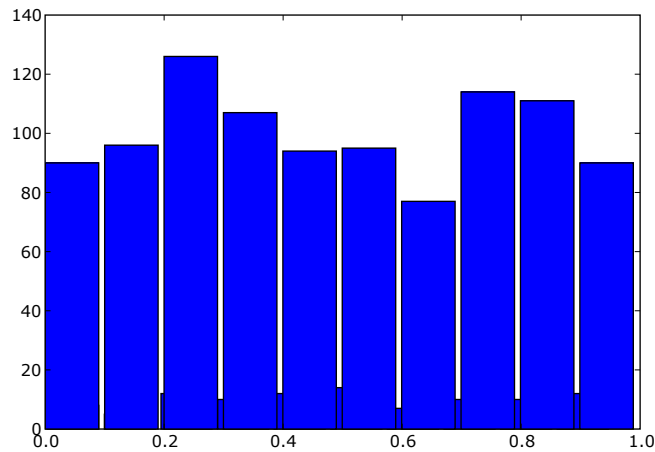


Figure 4.1: Uniform distribution

Okay – now x is 1000 random values. Let's make y two times each value of x . Can correlation still tell that these are similar values?

```
In [16]: y = x * 2
```

```
In [17]: correlation(x,y)
```

```
Out[17]: 1.0
```

Yup. Still can't fool it.

But those are uniform values. Most values in the real world are *normal*—they have a mean value that is really common, and other values are much less common. We can generate that from `RandomArray`, too, using the `standard_normal` function which assumes a mean of 0.0 and a standard deviation of 1.0.

```
In [19]: x = standard_normal((1000,))
```

```
In [20]: hist(x)
```

```
Out[20]:
```

```
([ 7, 24, 56, 143, 210, 255, 180, 98, 22, 5, ],
 [-3.13742447, -2.52991383, -1.92240319, -1.31489255, -0.70738192, -0.09987128,
  0.50763936, 1.11515, 1.72266064, 2.33017128, ],
 <a list of 10 Patch objects>)
```

```
In [21]: savefig("normal_hist.eps")
```

Now, if we look at the distribution of these 1000 values, we see a very different shape. There's clearly a bump in the middle at the average, and other values are less common (Figure 4.2).

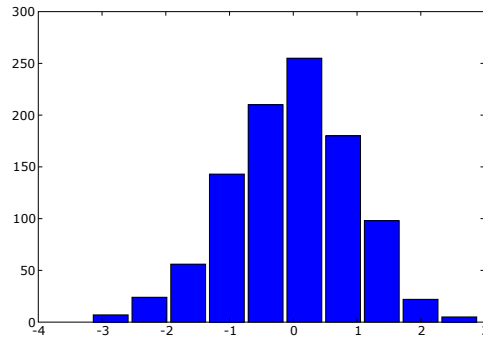


Figure 4.2: Normal distribution

Cool – so now let's see if correlation can figure out a relationship with normal values.

```
In [23]: y = 2 * x
```

```
In [24]: correlation(x,y)
Out[24]: 1.0
```

Yup – it still saw the strong positive correlation.

Let's try one last trick. Basically, what we have been exploring is where the y values are a simple factor (multiplication by two) from x . What if there is some other factor at play? What if the y values *remembered* the last value of y ? y_0 is just $2x_0$. But y_1 is $2x_1 + y_0$, and y_2 is $2x_2 + y_1$. If there's any other factor at play, even a very simple model of *memory*, can correlation detect that?

```
In [25]: memory = 0
```

```
In [26]: for i in range(0,1000):
....:     y[i] = (2 * x[i]) + memory
....:     memory = y[i]
....:
```

```
In [27]: correlation(x,y)
Out[27]: 0.035369277425458617
```

The answer seems to be *no*. Correlation no longer returns a strong r value, even when the y values are completely defined by the x values. This points out the weakness of correlation—which is also a strength. Correlation can't pick out complex relationships. It has to be a simple linear relationship or correlation won't see the relationship. On the other hand, if you *do* find a significant r , you can be pretty darn sure that you do have two related data sets. If you don't find a significant r , it doesn't mean that the values aren't somehow related—it just means that any relationship that's there is more complex than simply linear.

5 Text Analysis

One of the techniques used in *Freakonomics* [Levitt and Dubner, 2005] that is unusual for economists (and other social scientists, for that matter) is *textual analysis*. Levitt and Dubner analyze text strings of answers from students' standardized exams to find cheating teachers in one chapter, and they analyze baby names much later in the book. Computers are absolutely fantastic at textual analysis. We'll only use a couple of techniques in this chapter (enough to do some of what's in *Freakonomics*), but as you'll see, it's amazingly easy to do some really interesting textual analysis.

5.1 Visualizing textual differences: Bacon v. Shakespeare

As you may know, scholars for years have questioned whether Shakespeare really did write the works that he is said to have authored. (See <http://www.urbana.k12.oh.us/699/oh/authorship\%20controversy.html> for a nice summary of the controversy.) Here he was, the son of two illiterate parents with little formal education. Who would believe him to be the greatest English playwright?

One of the earliest authors thought to have penned Shakespeare's works was Francis Bacon. Bacon was a much more likely candidate—well-educated, well-spoken, a well-known writer.

We're not going to come up with anything novel that others haven't tried, but it's a fun context for trying out some interesting techniques. Our strategy will be to find something that correlates highly between two Bacon texts and between two Shakespeare texts, and *then* correlates highly between the Bacon and Shakespeare texts but not other authors' texts. Fortunately, Project Gutenberg¹ has tons of free books on-line. I grabbed *The Advancement of Learning* by Francis Bacon and *The Essays of Francis Bacon*, and then I grabbed *Macbeth* and *Romeo and Juliet*.

¹<http://www.gutenberg.org>

Visualizing the ‘the’

Here’s a stupid but fun hypothesis: Authors use a similar number of the instances of the word ‘the’ in a similar way. It’s silly, but easy to check. We’re going to start checking by simply visualizing the ‘the’s.

Reading the file is easy—we use `open`, `read`, `close`. The `find` method will find a given string. Even more powerful is `replace` that will replace all instances of one string with another one.

```
In [31]: file=open("essays-bacon.txt","rt")
```

```
In [32]: text=file.read()
```

```
In [33]: file.close()
```

```
In [34]: pat=" the "
```

```
In [35]: text.find(pat)
```

```
Out[35]: 137
```

```
In [36]: text[125:145]
```

```
Out[36]: 'ing all over the wor'
```

```
In [37]: text.replace(pat,"*"+pat+"*")
```

```
Out[37]:
```

```
In [38]: text[125:150] #NO CHANGE!
```

```
Out[38]: 'ing all over the world, b'
```

```
In [42]: "ababab".replace("a","z") #RETURNS the change
```

```
Out[42]: 'zbzbzb'
```

```
In [43]: newtext=text.replace(pat,"*"+pat+"*")
```

```
In [44]: newtext[125:150] #There’s the change!
```

```
Out[44]: 'ing all over* the *world,'
```

So here’s what we’re going to do. We’re going to create a file `viztext.py` and give it a `highlight` method that copies the text file to HTML. The HTML will reduce the font to its lowest possible size, and make the background black. Then we’ll make the pattern (the word ‘the’) red to make it stand out. Visually, we’ll be able to scan to see patterns of the word pattern we care about.

```
def highlight(basename, pattern):
    file = open(basename+".txt","rt")
    text=file.read()
    file.close()
```

```
# Now make the new one
newpat = '<font color="red">' + pattern + '</font>'
html = open(basename+".html", "wt")
html.write("<html><title>"+basename+"</title>\n")
html.write('<body bgcolor="white">')
html.write("<font size=1 color=black>")
newtext=text.replace(pattern, newpat)
html.write(newtext)
html.write("</body>")
html.close()
```

At first I tried it against a white background (Figure 5.1), but found that it wasn't very powerful. The red text pattern didn't stand out as much as against a black background (Figure 5.2).



*****The Project Gutenberg Etext of Essays of Francis Bacon***** #1 in our series by Francis Bacon Copyright laws are changing all over the world, be sure to check the copyright laws for your country before posting these files!! Please take a look at the important information in this header. We encourage you to keep this file on your own disk, keeping an electronic path open for the next readers. Do not remove this. **Welcome To The World of Free Plain Vanilla Electronic Texts** **Etexts Readable By Both Humans and By Computers, Since 1971** *These Etexts Prepared By Hundreds of Volunteers and Donations* Information on contacting Project Gutenberg to get Etexts, and further information is included below. We need your donations. Essays by Francis Bacon June, 1996 [Etext #575] [Most recently updated September 1, 2003] *****The Project Gutenberg Etext of Essays of Francis Bacon***** *****This file should be named ebacn10.txt or ebacn10.zip***** Corrected EDITIONS of our etexts get a new NUMBER, ebacn11.txt. VERSIONS based on separate sources get new LETTER, ebacn10a.txt. This etext was created by Judith Boss, Omaha, Nebraska. The equipment: an IBM-compatible 486/50, a Hewlett-Packard ScanJet IIC flatbed scanner, and Calera Recognition Systems' M/600 Series Professional OCR software and RISC accelerator board donated by Calera Recognition Systems. We are now trying to release all our books one month in advance of the official release dates, for time for better editing. Please note: neither this list nor its contents are final till midnight of the last day of the month of any such announcement. The official release date of all Project Gutenberg Etexts is at Midnight, Central Time, of the last day of the stated month. A preliminary version may often be posted for suggestion, comment and editing by those who wish to do so. To be sure you have an up to date first edition [xxxxx10x.xxx] please check file sizes in the first week of the next month. Since our ftp program has a bug in it that scrambles the data [tried to fix and failed] a look at the file size will have to do, but we will try to see a new copy has at least one byte more or less. Information about Project Gutenberg (one page) We produce about two million dollars for each hour we work. The fifty hours is one conservative estimate for how long it we take to get any etext selected, entered, proofread, edited, copyright searched and analyzed, the copyright letters written, etc. This projected audience is one hundred million readers. If our value per text is nominally estimated at one dollar then we produce \$2 million dollars per hour this year as we release thirty-two text files per month: or 400 more Etexts in 1996 for a total of 800. If these reach just 10% of the computerized population, then the total should reach 80 billion Etexts. The Goal of Project Gutenberg is to Give Away One Trillion Etext Files by the December 31, 2001. [10,000 x 100,000,000=Trillion] This is ten thousand titles each to one hundred million readers, which is only 10% of the present number of computer users. 2001 should have at least twice as many computer users as that, so it will require us reaching less than 5% of the users in 2001. We need your donations more than ever! All donations should be made to "Project Gutenberg/IBC", and are tax deductible to the extent allowable by law ("IBC" is Illinois Benedictine College). (Subscriptions to our paper newsletter go to IBC, too) For these and other matters, please mail to: Project Gutenberg P. O. Box 2782 Champaign, IL 61825 When all other email fails try our Executive Director: Michael S. Hart We would prefer to send you this information by email (Internet, Bitnet, Compuserve, ATTMAIL or MCImail). ***** If you have an FTP program (or emulator), please FTP directly to the Project Gutenberg archives: [Mac users, do NOT point and click. . .type] ftp.us.archive.cso.uiuc.edu login: anonymous password: your@login cd etext/etext90 through /etext96 or cd etext/articles [get suggest gut for more information] dir [to see files] get or mget [to get files. . . set bin for zip files] GET INDEX700.GUT for a list of books and GET NEW GUT for general information and MGET GUT* for newsletters. **Information prepared by the Project Gutenberg legal advisor** (Three Pages) ***START**THE SMALL PRINT!**FOR PUBLIC DOMAIN ETEXTS**START*** Why is this "Small Print!" statement here? You know: lawyers. They tell us you might sue us if there is something wrong with your copy of this etext, even if you got it for free from someone other than us, and even if what's wrong is not our fault. So, among other things, this "Small Print!" statement disclaims most of our liability to you. It also tells you how you can distribute copies of this etext if you want to. *BEFORE!* YOU USE OR READ THIS ETEXT By using or reading any part of this PROJECT GUTENBERG-tm etext, you indicate that you understand, agree to and accept this "Small Print!" statement. If you do not, you can receive a refund of the money (if any) you paid for this etext by sending a request within 30 days of receiving it to the person you got it from. If you received this etext on a physical medium (such as a

Figure 5.1: Francis Bacon's essays with white background, 'the' highlighted

```
In [53]: reload(viztext)
```

```
Out[53]: <module 'viztext' from 'viztext.py'>
```

```
In [54]: viztext.highlight("essays-bacon", " the ")
```

```
def highlight(basename, pattern):
    file = open(basename+".txt", "rt")
    text=file.read()
    file.close()
    # Now make the new one
    newpat = '<font color="red">'+pattern+'</font>'
    html = open(basename+".html", "wt")
    html.write("<html><title>"+basename+"</title>\n")
    html.write('<body bgcolor="black">')
    html.write("<font size=1 color=black>")
    newtext=text.replace(pattern, newpat)
    html.write(newtext)
    html.write("</body>")
    html.close()
```

The obvious next thing to do is to process both the *Essays* and *Macbeth* to compare the visualizations (Figure 5.3). As one might anticipate for a fairly simple and stupid hypothesis – I don't see anything there, do you?

```
In [54]: viztext.highlight("essays-bacon", " the ")
```

```
In [55]: viztext.highlight("macbeth-shakespeare", " the ")
```

Visualizing the Proper Nouns: Using Regular Expressions

So let's consider a different hypothesis: That a unique characteristic of an author is the number of capitalized words that they use. That indicates the number of sentences, but also indicates the number of proper nouns (e.g., names of things) that are used. Perhaps that's a determining factor?

To locate capitalized words, we need something a bit stronger than a text pattern to search for. Computer scientists use *regular expressions* to describe patterns of words, not just specific words. This allows for significant flexibility in exploring text. Think of regular expressions as being like mathematical expressions, in that there are constants and operators, but we're describing patterns of letters, not patterns of numbers.

The name of the package that knows about regular expressions in Python is `re`. You **import** it to use it.

```
In [3]: import re
```

```
In [4]: string = "This is my name: Mark Guzdial. I live in Decatur."
```

The `match` method takes a regular expression and a string as input, then returns a *match object* that describes the match, or literally `None` if

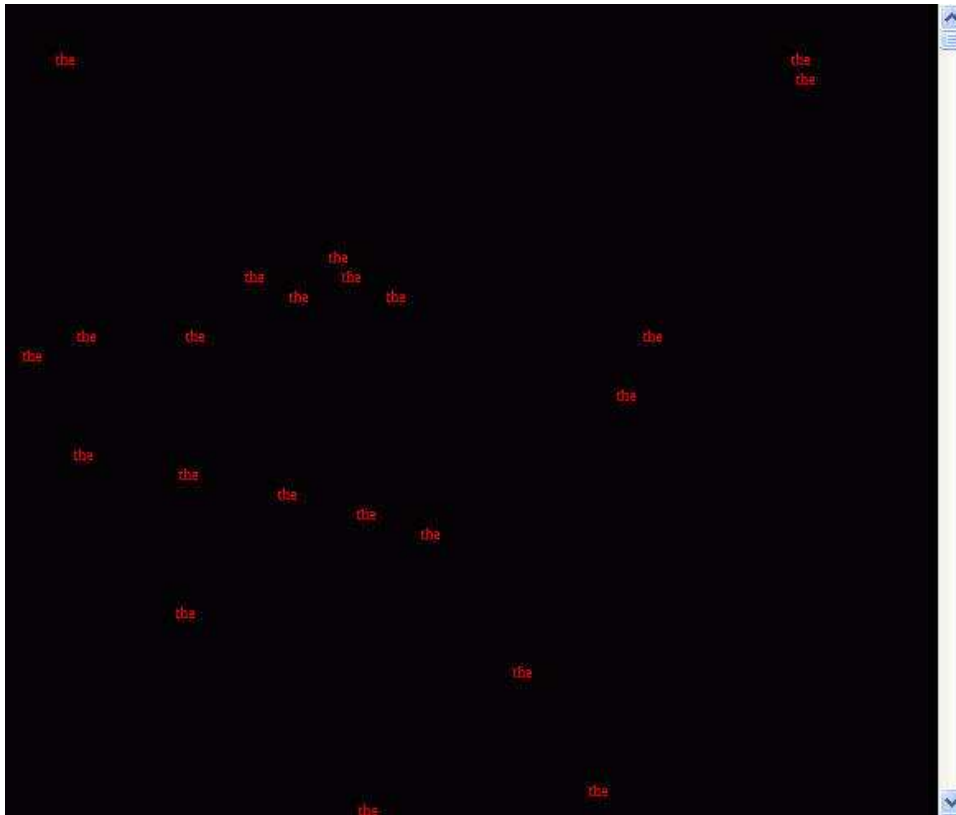


Figure 5.2: Francis Bacon's essays with black background, 'the' highlighted

no match is found. The most common uses of the match object is to get the start position and the end position of the match.

Rules for matches

Here are how regular expressions are constructed.

- Any regular character matches only the same regular character. 'M' matches only to 'M', and 'a' matches only to the letter 'a'.

```
In [5]: matchobject=re.search("Mark",string)
```

```
In [6]: print matchobject.start()
```

```
17
```

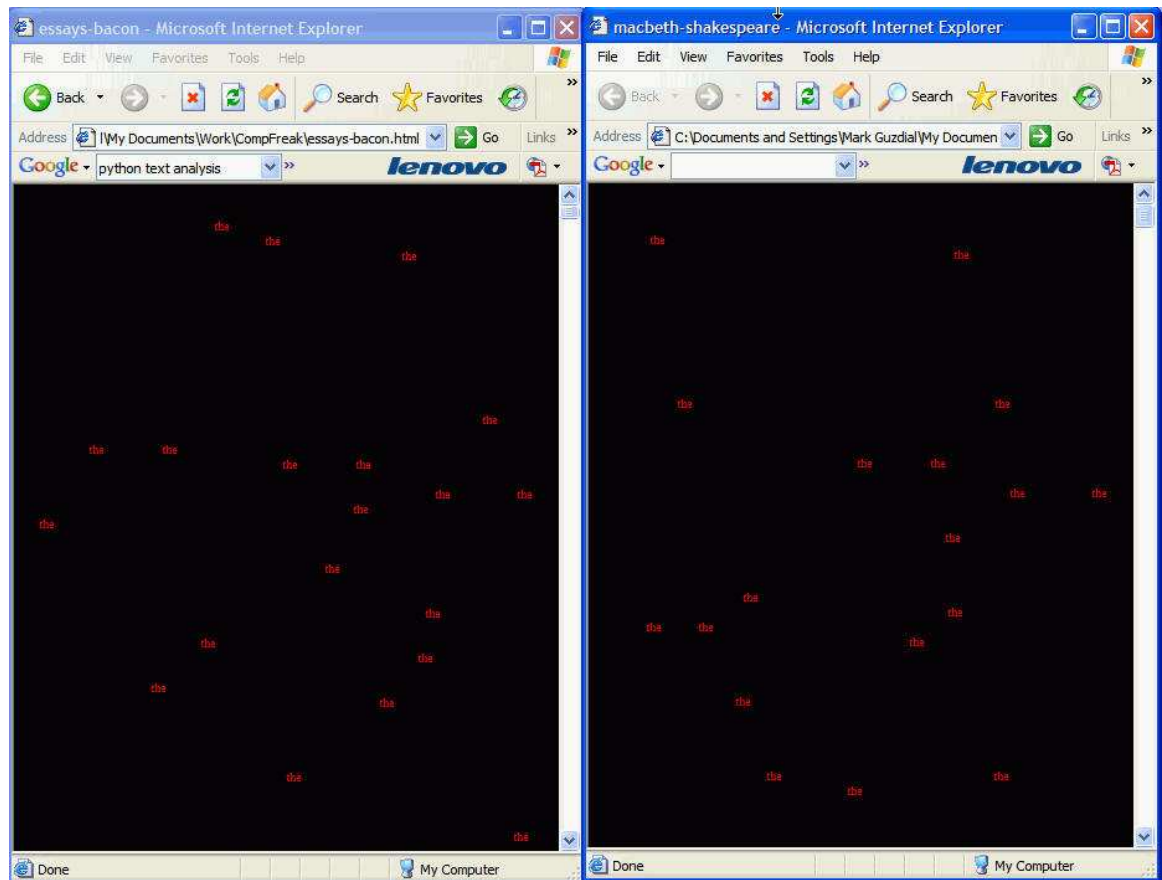


Figure 5.3: Comparing 'the' patterns in Bacon's *Essays* and Shakespeare's *Macbeth*

```
In [7]: print matchobject.end()
21
```

```
In [8]: string[17:21]
Out[8]: 'Mark'
```

- A period matches anything.
- A * says "repeat zero or more times whatever came before me." The below example looks for a match that starts with a lowercase 'm' and then is followed by any number of characters—which, of course, matches everything else in the string.

```
In [9]: matchobject=re.search("m.*",string)
```

```
In [10]: print string[matchobject.start():matchobject.end()]
my name: Mark Guzdial. I live in Decatur.
```

Again, if the match doesn't work, you get a None that doesn't understand start nor end

```
In [11]: matchobject=re.search("m.*\b",string)
```

```
In [12]: print string[matchobject.start():matchobject.end()]
```

```
-----
exceptions.AttributeError
last)
```

```
Traceback (most recent call
```

```
C:\Documents and Settings\Mark Guzdial\My Documents\Work\CompFreak\<console>
```

```
AttributeError: 'NoneType' object has no attribute 'start'
```

- Putting an "r" before a string treats it as raw mode and backslashes don't get interpreted by Python. A b is supposed to find word boundaries.

```
In [13]: matchobject=re.search(r"m.*\b",string)
```

```
In [14]: print matchobject
<_sre.SRE_Match object at 0x00CC7218>
```

```
In [15]: print string[matchobject.start():matchobject.end()]
my name: Mark Guzdial. I live in Decatur
```

- The code S means "anything that isn't *whitespace* (tab, return, space." The s means whitespace. So the below code looks for a word that starts with "m" and has any number of characters before a whitespace character.

```
In [18]: matchobject=re.search(r"m\S*\s",string)
```

```
In [19]: print string[matchobject.start():matchobject.end()]
my
```

- Character classes are in square brackets. [A-Z] only matches a single uppercase character. [a-zA-Z] matches any character of any case. The below test looks for an uppercase character at the beginning of a word, followed by any number of any kinds of letters, but only letters.

```
In [3]: match = re.search(r"\b[A-Z][a-zA-Z]*",string)
```

```
In [4]: print match.start()
0
```

```
In [5]: print match.end()
4
```

```
In [6]: print string[match.start():match.end()]
This
```

- The character `+` is like `*`, but `+` insists that there must be at least one of what it matches.

There are many other parts of regular expressions, but that's enough to get started.

Finding Capitalized Words

There are a couple more methods besides `search` that can be useful in regular expression processing. The first is called `split`. Given a regular expression and a string, it returns a sequence of substrings of everything that does *not* match the pattern.

```
In [13]: chopped = re.split(r"\b[A-Z][a-zA-Z]*",string)
```

```
In [14]: chopped[0]
Out[14]: ''
```

```
In [15]: chopped[1]
Out[15]: ' is my name: '
```

```
In [16]: chopped[2]
Out[16]: ''
```

```
In [17]: chopped[3]
Out[17]: '. '
```

Any regular expression in parentheses is *grouped*. We can later refer to those groupings as 1, 2, and so on here. We can use that with `sub` which substitutes one pattern with another in a given string.

Here, we use our capitalization pattern to wrap red font coloring around capitalized words.

```
In [24]: newtext = re.sub(r"(\b[A-Z][a-zA-Z]*)",r'<font color=red>\1</font>>
ing)
```

```
In [25]: newtext
```

```
Out[25]: '<font color=red>This</font> is my name: <font color=red>Mark</font> <font color=red>Guzdial</font>. <font color=red>I</font> live in <font color=red>Deatur</font>.'
```

Using our newfound capability to replace patterns, we can expand our `viztext.py` tool to highlight capitals. (Note the **import re** at the top.)

```
import re
```

```
def highlight(basename, pattern):
    file = open(basename+".txt", "rt")
    text=file.read()
    file.close()
    # Now make the new one
    newpat = '<font color="red">'+pattern+'</font>'
    html = open(basename+".html", "wt")
    html.write("<html><title>"+basename+"</title>\n")
    html.write('<body bgcolor="black">')
    html.write("<font size=1 color=black>")
    newtext=text.replace(pattern, newpat)
    html.write(newtext)
    html.write("</body>")
    html.close()

def highlightCapitals(basename):
    file = open(basename+".txt", "rt")
    text=file.read()
    file.close()
    # Now make the new one
    html = open(basename+".html", "wt")
    html.write("<html><title>"+basename+"</title>\n")
    html.write('<body bgcolor="black">')
    html.write("<font size=1 color=black>")
    newtext=re.sub(r"(\b[A-Z][a-zA-Z]*)", r'<font color="red">\1</font>', text)
    html.write(newtext)
    html.write("</body>")
    html.close()
```

It's pretty easy to use, and the result is more interesting than all the 'the's (Figure 5.4).

```
In [26]: import viztext
```

```
In [27]: viztext.highlightCapitals("essays-bacon")
```

5.2 Counting Text Patterns

While the visualizations are fun, I doubt that we're going to see any particularly interesting patterns that way. We're going to need to leverage some numeric capability.


```

file = open(basename+".txt","rt")
text=file.read()
file.close()
# Break it up by paragraphs
newtext=text.split('\n\n') #Two returns = paragraph
# Now, count the number of 'the's in the paragraph
ret = []
for s in newtext:
    ret.append(s.count(pattern))
return ret

```

Now, let's try it.

```
In [44]: import counttext
```

```
In [45]: essaysThe = counttext.countText("essays-bacon"," the ")
```

```
In [46]: len(essaysThe)
```

```
Out[46]: 508
```

```
In [47]: essaysThe[0:10] #What do the answers look like?
```

```
Out[47]: [0, 1, 2, 0, 0, 0, 0, 0, 0, 0]
```

```
In [48]: advThe = counttext.countText
         ("advancement-learning-bacon"," the ")
```

```
In [50]: romeoThe = counttext.countText
         ("romeo-juliet-shakespeare"," the ")
```

```
In [51]: macbethThe = counttext.countText
         ("macbeth-shakespeare"," the ")
```

```
In [52]: len(romeoThe)
```

```
Out[52]: 287
```

```
In [53]: len(macbethThe)
```

```
Out[53]: 271
```

```
In [54]: len(advThe)
```

```
Out[54]: 597
```

```
In [55]: len(essaysThe)
```

```
Out[55]: 508
```

Unfortunately, there are different number of paragraphs in each sample text. So, we'll do a correlation just the first 200 paragraphs. The answer is pretty abysmal. Counting the "the"s doesn't seem to be a useful metric.

```
In [56]: from correlation import *
```

```
In [57]: correlation(romeoThe[0:200],macbethThe[0:200])
```

```
Out[57]: 0.039670370779755854
```

```
In [58]: correlation(advThe[0:200],essaysThe[0:200])
```

```
Out[58]: -0.0085796837192241779
```

Counting Capitals

Let's try our second hypothesis, counting the number of capitalized letters. To get the number of capital words in each paragraph, we'll generate the `re.split` of each paragraph, then count the number of match objects returned. One less than that will be the number of capitals.

```
def countCapitals(basename):
    file = open(basename+".txt", "rt")
    text=file.read()
    file.close()
    # Break it up by paragraphs
    newtext=text.split('\n\n')
    # Count the capitals
    ret = []
    for para in newtext:
        match = re.split(r"\b[A-Z][a-zA-z]*", para)
        ret.append(len(match)-1)
    return ret
```

Now let's try it.

```
In [61]: advCap = counttext.countCapitals("essays-bacon")
```

```
In [62]: len(advCap)
```

```
Out[62]: 508
```

```
In [63]: advCap[0:10]
```

```
Out[63]: [9, 1, 3, 9, 8, 7, 5, 1, 2, 4]
```

```
In [64]: essaysCap = counttext.countCapitals("advancement-learning-bacon")
```

```
In [65]: romeoCap = counttext.countCapitals("romeo-juliet-shakespeare")
```

```
In [66]: macbethCap = counttext.countCapitals("macbeth-shakespeare")
```

```
In [67]: correlation(advCap[0:200],essaysCap[0:200])
```

```
Out[67]: -0.10076023917690945
```

```
In [68]: correlation(romeoCap[0:200],macbethCap[0:200])
```

```
Out[68]: 0.016919364776092155
```

Eww – that correlation isn't very good either. Good thing we're not Shakespearean scholars...

6 Hypothesis Testing

In our text analysis, we ran against the problem of having to compare sequences of numbers that weren't paired. Correlations were really the wrong things to use there. We had no reason to believe that paragraph-by-paragraph, our metrics (counting "the"s and capitalized words) would change in-step.

What we really wanted to ask was if the sets were *different*, not in lock-step. What do we mean by different? Well, are there *means* different? Are the averages of each set significantly different? That's what we're going to test in this chapter. We're going to use sets that have the same number of elements, but we'll still be asking about means.

6.1 The Context: Elections and Unemployment Rates

One of the claims of political pundits is that what the US people are voting on in a presidential election is whether they're doing better or worse than they were four years previously. Let's test that.

- In 1996, Clinton was re-elected over Dole – the American people chose to stick with their party.
- In 2000, Bush won over Gore (even though Gore won the popular vote). The American people switched parties. Were they better off than they were 4 years previously?
- In 2004, Bush won over Kerry. The American people stuck with the Republican party. Were they about the same as they were four years previously?

I downloaded a data set from the US Bureau of Labor Statistics of US Unemployment Data (one measure of "doing better") over many years.

```
In [70]: import csvfile
```

```
In [71]: file = csvfile.CSVfile("USUnemploymentRate.csv")
```

```
In [72]: file.next()
```

```
Out[72]:
```

```
{'Annual': '',
  'Apr': '3.9',
  'Aug': '3.9',
  'Dec': '4',
  'Feb': '3.8',
  'Jan': '3.4',
  'Jul': '3.6',
  'Jun': '3.6',
  'Mar': '4',
  'May': '3.5',
  'Nov': '3.8',
  'Oct': '3.7',
  'Sep': '3.8',
  'Year': '1948'}
```

```
In [73]: file.next()
```

```
Out[73]:
```

```
{'Annual': '',
  'Apr': '5.3',
  'Aug': '6.8',
  'Dec': '6.6',
  'Feb': '4.7',
  'Jan': '4.3',
  'Jul': '6.7',
  'Jun': '6.2',
  'Mar': '5',
  'May': '6.1',
  'Nov': '6.4',
  'Oct': '7.9',
  'Sep': '6.6',
  'Year': '1949'}
```

Let's use these data to compare 1996, 2000, and 2004. Were they really different? Does the direction of difference match the election results?

6.2 T-Test

A *T-Test* tells us whether the averages of two sets are significantly different or not. The hypothesis we're testing is whether they're the same (H_0), or different (H_1).

Here's the process for a t-test computation.

- We need the averages (\bar{x}_1 and \bar{x}_2) of each group. We know how to do that already.
- We need the variance (s_1^2 and s_2^2) of each group. We can chop that out of our standard deviation function that we created earlier.
- We need the *pooled sample variance*. This is:

$$s_p^2 = \frac{(N_{group1} - 1)s_1^2 + (N_{group2} - 1)s_2^2}{(N_1 + N_2) - 2} \quad (6.1)$$

The overall t statistic is then:

$$\frac{\bar{x}_1 - \bar{x}_2}{\sqrt{s_p^2 \left(\frac{1}{N_1} + \frac{1}{N_2} \right)}} \quad (6.2)$$

The degrees of freedom are $N_1 + N_2 - 1$.

How to do a T-Test in Python

Here's an implementation of all of that in Python.

```
def average(sequence):
    return (1.0*sum(sequence))/len(sequence)

def variance(sequence):
    ave = average(sequence)
    # Compute the mean squared difference
    diffs = 1.0
    for num in sequence:
        diffs = diffs + pow((ave-num),2)
    # Compute the variance
    variance = diffs/(len(sequence)-1)
    return variance

def ttest(seq1,seq2):
    x1 = average(seq1)
    x2 = average(seq2)
    s1 = variance(seq1)
    s2 = variance(seq2)
    n1 = len(seq1)
    n2 = len(seq2)
    pooleds = (((n1-1)*s1)+((n2-1)*s2))/((n1+n2)-2)
    t=(x1-x2)/pow(pooleds*((1/n1)+(1/n2)),0.5)
    return t
```

And putting it all together, with reading our unemployment rates, we get:

```
from pylab import *
import csvfile

def average(sequence):
    return (1.0*sum(sequence))/len(sequence)

def variance(sequence):
    ave = average(sequence)
    # Compute the mean squared difference
```

```

    diffs = 1.0
    for num in sequence:
        diffs = diffs + pow((ave-num),2)
    # Compute the variance
    variance = diffs / (len(sequence) - 1)
    return variance

def ttest(seq1, seq2):
    x1 = average(seq1)
    x2 = average(seq2)
    s1 = variance(seq1)
    s2 = variance(seq2)
    n1 = len(seq1)
    n2 = len(seq2)
    pooleds = (((n1-1)*s1) + ((n2-1)*s2)) / ((n1+n2)-2)
    t = (x1-x2) / pow(pooleds * ((1.0/n1) + (1.0/n2)), 0.5)
    return t

unemdata = csvfile.CSVfile("USUnemploymentRate.csv")
months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec']

#Let's get 1996 year.
rates1996 = []
# getRows returns a set. We want just one, the 0th
row1996 = unemdata.getRows('Year', '1996')[0]
for item in months:
    rates1996.append(csvfile.number(row1996[item]))

unemdata.rewind()
#Let's get 2000 year.
rates2000 = []
row2000 = unemdata.getRows('Year', '2000')[0]
for item in months:
    rates2000.append(csvfile.number(row2000[item]))

unemdata.rewind()
#Let's get 2004 year.
rates2004 = []
row2004 = unemdata.getRows('Year', '2004')[0]
for item in months:
    rates2004.append(csvfile.number(row2004[item]))

print "1996 results"
print "average", average(rates1996)
print "variance", variance(rates1996)
print "number", len(rates1996)

print "2000 results"
print "average", average(rates2000)
print "variance", variance(rates2000)

```

```

print "number", len(rates2000)

print "2004 results"
print "average", average(rates2004)
print "variance", variance(rates2004)
print "number", len(rates2004)

print "1996–2000 ttest", ttest(rates1996, rates2000)
print "2000–2004 ttest", ttest(rates2000, rates2004)

```

And here's the run:

```

In [107]: run ttest-electoral.py
1996 results
average 5.40833333333
variance 0.120833333333
number 12
2000 results
average 3.96666666667
variance 0.0987878787879
number 12
2004 results
average 5.51666666667
variance 0.105151515152
number 12
1996-2000 ttest 10.6565919854
2000-2004 ttest -11.8897236086

```

This suggests that 2000 was much worse (on average) than 1996, and 2004 was better than 2000. But was it significant? We can check a t-test table¹, assuming an *alpha* value (willingness to be wrong) of 0.05 and $(12 + 12 - 1 = 23)$ 23 degrees of freedom, we get a $t_{critical}$ value of 1.714. We reject H_0 if our t-value is greater than $t_{critical}$. Since our t-value is way larger, we say that the difference is significant at the 0.05 level.

That means that the 2000 election matched the pundit's prediction – people *were* worse off, so they changed parties (ignoring the popular vs. electoral vote complexity). In 2004, they were markedly *better* than they were in 2000, so they stuck with the same party.

6.3 ANOVA: Analysis of Variance

Another way of testing the difference between groups is with an *ANOVA* or *Analysis of Variance*. Here, we look at variance more, and we use an *f statistic* rather than the *t statistic* (that is, the lookup table). One thing that's cool about ANOVA is that we can use it for more than two groups, to

¹We're using <http://www.socr.ucla.edu/Applets.dir/T-table.html>.

see if there is any difference anywhere. But we'll use it just for two groups here.

In ANOVA, we need the totals of the groups, the sample sizes, and the means, but also the *grand total* (of all the groups), the *total sample size* (of all groups), and the *grand mean* which is the grand total divided by the total sample size. Strangely enough, we don't actually use the variance in the Analysis of Variance (ANOVA) process.

Here's the process:

- We compute the *SSB*, the sum of squares between groups.

$$SSB = (N_1(x_1 - GrandMean))^2 + (N_2(x_2 - GrandMean))^2 \quad (6.3)$$

(You can easily see how this would extend to more groups.)

- We compute the *SSW*, the sum of squares *within* groups. For all items in *all* groups,

$$SSW = (item1 - (meanofitem1'sgroup))^2 + (item2 - (meanofitem2'sgroup)) \dots \quad (6.4)$$

- We then compute the *MSB*, mean square between groups. That's simpler to compute:

$$MSB = \frac{SSB}{numberofgroups - 1} \quad (6.5)$$

- We then compute the *MSW*, mean square within groups.

$$MSW = \frac{SSW}{(totalsamplesize) - (numberofgroups)} \quad (6.6)$$

- The F-statistic is:

$$F = \frac{MSB}{MSW} \quad (6.7)$$

- The degrees of freedom between groups is the number of groups - 1. The degrees of freedom for the total is the total sample size - 1. The degrees of freedom within groups is the degrees of freedom for the total minus the degrees of freedom between groups. I find on some F-statistic tables², the between groups is called *df1* and the within groups is called *df2*.

²As at <http://www.statsoft.com/textbook/sttable.html#f05>

How to do an ANOVA in Python

Here's how we map that process to Python.

```

from pylab import *
import csvfile

def average(sequence):
    return (1.0*sum(sequence))/len(sequence)

def variance(sequence):
    ave = average(sequence)
    # Compute the mean squared difference
    diffs = 1.0
    for num in sequence:
        diffs = diffs + pow((ave-num),2)
    # Compute the variance
    variance = diffs/(len(sequence)-1)
    return variance

def ttest(seq1,seq2):
    x1 = average(seq1)
    x2 = average(seq2)
    s1 = variance(seq1)
    s2 = variance(seq2)
    n1 = len(seq1)
    n2 = len(seq2)
    pooleds = (((n1-1)*s1)+((n2-1)*s2))/((n1+n2)-2)
    t=(x1-x2)/pow(pooleds*((1.0/n1)+(1.0/n2)) ,0.5)
    return t

def anova(seq1,seq2):
    sum1 = sum(seq1)
    sum2 = sum(seq2)
    grandtotal = sum1 + sum2
    n1 = len(seq1)
    n2 = len(seq2)
    totalsize = n1+n2
    x1 = average(seq1)
    x2 = average(seq2)
    grandmean = float(grandtotal)/totalsize
    SSB = (n1*pow(x1-grandmean,2))+(n2*pow(x2-grandmean,2))
    SSW = 0
    for i in seq1:
        SSW=SSW+pow(i-x1,2)
    for i in seq2:
        SSW=SSW+pow(i-x2,2)
    MSB=SSB/1
    MSW=float(SSW)/(totalsize-2)
    return MSB/MSW

```

Then, here's the whole thing, including the reading of the data again.

```

from pylab import *
import csvfile

def average(sequence):
    return (1.0*sum(sequence))/len(sequence)

def variance(sequence):
    ave = average(sequence)
    # Compute the mean squared difference
    diffs = 1.0
    for num in sequence:
        diffs = diffs + pow((ave-num),2)
    # Compute the variance
    variance = diffs/(len(sequence)-1)
    return variance

def ttest(seq1,seq2):
    x1 = average(seq1)
    x2 = average(seq2)
    s1 = variance(seq1)
    s2 = variance(seq2)
    n1 = len(seq1)
    n2 = len(seq2)
    pooleds = (((n1-1)*s1)+((n2-1)*s2))/((n1+n2)-2)
    t=(x1-x2)/pow(pooleds*((1.0/n1)+(1.0/n2)) ,0.5)
    return t

def anova(seq1,seq2):
    sum1 = sum(seq1)
    sum2 = sum(seq2)
    grandtotal = sum1 + sum2
    n1 = len(seq1)
    n2 = len(seq2)
    totalsize = n1+n2
    x1 = average(seq1)
    x2 = average(seq2)
    grandmean = float(grandtotal)/totalsize
    SSB = (n1*pow(x1-grandmean,2))+(n2*pow(x2-grandmean,2))
    SSW = 0
    for i in seq1:
        SSW=SSW+pow(i-x1,2)
    for i in seq2:
        SSW=SSW+pow(i-x2,2)
    MSB=SSB/1
    MSW=float(SSW)/(totalsize-2)
    return MSB/MSW

unemdata = csvfile.CSVfile("USUnemploymentRate.csv")

```

```

months=['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec']

#Let's get 1996 year.
rates1996 = []
# getRows returns a set. We want just one, the 0th
row1996 = unemdata.getRows('Year', '1996')[0]
for item in months:
    rates1996.append(csvfile.number(row1996[item]))

unemdata.rewind()
#Let's get 2000 year.
rates2000 = []
row2000 = unemdata.getRows('Year', '2000')[0]
for item in months:
    rates2000.append(csvfile.number(row2000[item]))

unemdata.rewind()
#Let's get 2004 year.
rates2004 = []
row2004 = unemdata.getRows('Year', '2004')[0]
for item in months:
    rates2004.append(csvfile.number(row2004[item]))

print "1996 results"
print "average", average(rates1996)
print "variance", variance(rates1996)
print "number", len(rates1996)

print "2000 results"
print "average", average(rates2000)
print "variance", variance(rates2000)
print "number", len(rates2000)

print "2004 results"
print "average", average(rates2004)
print "variance", variance(rates2004)
print "number", len(rates2004)

print "1996-2000 anova", anova(rates1996, rates2000)
print "2000-2004 anova", anova(rates2000, rates2004)

```

And here are the results:

```

1996 results
average 5.40833333333
variance 0.120833333333
number 12
2000 results
average 3.96666666667
variance 0.0987878787879

```

```
number 12
2004 results
average 5.51666666667
variance 0.105151515152
number 12
1996-2000 anova 659.75751503
2000-2004 anova 1303.2739726
```

The degrees of freedom between groups is 1 ($2groups - 1$). The degrees of freedom total is 23 ($24 - 1$). The degrees of freedom within groups is 22 ($23 - 1$). The F-statistic there is 4.3009. These values (659 and 1303) are, ahem, a tad bit larger. So, we come up with the same result as with the t-test.

So, for the 2000 and 2004 elections, the pundit's prediction held true. As goes the unemployment rate, so goes the presidential vote.

A Program Listings

A.1 CVSfile

Program Example #0

CVSfile

```
## CSVfile — a front end to CVS

import csv

def number(input, default=-1):
    try:
        return float(input)
    except:
        return default

class CSVfile:
    def __init__(self, filename):
        self.filename = filename
        self.rewind()

    def rewind(self):
        self.fp = open(self.filename, "rb")
        headerReader = csv.reader(self.fp)
        self.headers = headerReader.next()
        self.dataReader = csv.DictReader(self.fp, fieldnames=self.headers)

    def next(self):
        return self.dataReader.next()

    def getRows(self, fieldname, value):
        ret = []
        for row in self.dataReader:
            if row[fieldname]==value:
                ret.append(row)
        return ret
```

```

def getColumn(self,fieldname):
    ret = []
    for row in self.dataReader:
        ret.append(row.get(fieldname))
    return map(number,ret)

```

A.2 fancierplot.py – a run-able plot

Program Example #1

fancierplot.py

```

from pylab import *
import csvfile
popdata = csvfile.CSVfile("pops-2000.csv")
pops = popdata.getColumn("POP")
spops=sort(pops)

plot(spops[1:],marker="o",color="r")
title('Populations of countries in the year 2000')
xlabel('Countries in increasing order of population')
ylabel('Population in millions')
grid(True)
show()

```

A.3 US-UK Population Plot for years 1999–2000

Program Example #2

us_uk_pop_plot.py

```

from pylab import *
import csvfile
natdata = csvfile.CSVfile("us-uk-1990-2000.csv")
usdata = natdata.getRows('country','United States')
natdata.rewind()
ukdata = natdata.getRows('country','United Kingdom')

#Get the populations
uspops = []
for row in usdata:
    uspops.append(csvfile.number(row['POP']))

```

```

ukpops = []
for row in ukdata:
    ukpops.append(csvfile.number(row['POP']))
years=range(1990,2001)
print "US",uspops,len(uspops)
print "UK",ukpops,len(ukpops)
print "Years",years,len(years)

plot(years,uspops,'r—o',years,ukpops,'b-x')
legend(('US Population','UK Population'),loc='center right')
title('Populations of US and UK 1990–2000')
xlabel('Years')
ylabel('Population in millions')
grid(True)
savefig("us_uk_pop_plot.eps")
show()

```

us_uk_pop_plot2.py*Program Example #3*

```

from pylab import *
import csvfile
natdata = csvfile.CSVfile("us-uk-1990-2000.csv")
usdata = natdata.getRows('country','United States')
natdata.rewind()
ukdata = natdata.getRows('country','United Kingdom')

#Get the populations
# This time, making SURE that they're in year-order
years=range(1990,2001)
uspops = []
for y in years:
    for row in usdata:
        if row['year']==str(y): #Items in rows are strings
            uspops.append(csvfile.number(row['POP']))
            break #Leave the row loop
ukpops = []
for y in years:
    for row in ukdata:
        if row['year']==str(y):
            ukpops.append(csvfile.number(row['POP']))
            break

# Top subplot: 2 rows, 1 column, subplot #1
subplot(2,1,1)
plot(years,uspops,'r—o')

```

```

title('Population of US 1990–2000')
xlabel('Years')
ylabel('Population in millions')
grid(True)

subplot(2,1,2)
plot(years, ukpops, 'b-x')
title('Population UK 1990–2000')
xlabel('Years')
ylabel('Population in millions')
grid(True)

savefig("us_uk_pop_plot2.eps")
show()

```

A.4 Exploring British and American Petroleum Company Stock Prices

Program Example #4

bpStdDev1990.py—computing descriptive statistics of BP and XOM

```

from pylab import *
import csvfile

def average(sequence):
    return (1.0*sum(sequence))/len(sequence)

def std_dev(sequence):
    ave = average(sequence)
    # Compute the mean squared difference
    diffs = 1.0
    for num in sequence:
        diffs = diffs + pow((ave-num),2)
    # Compute the variance
    variance = diffs/len(sequence)
    # Return the square root of the variance
    return pow(variance,0.5)

bpdata = csvfile.CSVfile("BritishPetroleum-BP-table.csv")

#Let's get the 1990 year.
closes = []
for row in bpdata.dataReader:

```

```
    if row[ 'Date' ].endswith( '90' ):
        closes.append( csvfile.number(row[ 'Close' ]))

#Return the average
print "*** BP ***"
print "Closing values", closes
print "Average:", average( closes )
print "Standard Deviation:", std_dev( closes )

amdata = csvfile.CSVfile( "Exxon-Mobile-XOM-table.csv" )

#Let's get the 1990 year.
closes = []
for row in amdata.dataReader:
    if row[ 'Date' ].endswith( '90' ):
        closes.append( csvfile.number(row[ 'Close' ]))

#Return the average
print "*** Exxon/Mobil ***"
print "Closing values", closes
print "Average:", average( closes )
print "Standard Deviation:", std_dev( closes )
```

bpHist1990.py—computing a histogram of each

Program Example #5

```
from pylab import *
import csvfile

bpdata = csvfile.CSVfile( "BritishPetroleum-BP-table.csv" )

#Let's get the 1990 year.
closes = []
for row in bpdata.dataReader:
    if row[ 'Date' ].endswith( '90' ):
        closes.append( csvfile.number(row[ 'Close' ]))

subplot(2,1,1)
title( "BP stock in 1990--Histogram" )
hist( closes )

amdata = csvfile.CSVfile( "Exxon-Mobile-XOM-table.csv" )

#Let's get the 1990 year.
closes = []
```

```

for row in amdata.dataReader:
    if row[ 'Date' ].endswith( '90' ):
        closes.append( csvfile.number( row[ 'Close' ] ))

subplot(2,1,2)
title( "Amoco/Mobil stock in 1990–Histogram" )
hist( closes )

savefig( "BP_AM_hist.eps" )
show()

```

*Program Example #6***bpAmCorrel1990.py—computing a correlation between them**

```

from pylab import *
import csvfile

def average( sequence ):
    return ( 1.0*sum( sequence ) ) / len( sequence )

def std_dev( sequence ):
    ave = average( sequence )
    # Compute the mean squared difference
    diffs = 1.0
    for num in sequence:
        diffs = diffs + pow( ( ave-num ), 2 )
    # Compute the variance
    variance = diffs / len( sequence )
    # Return the square root of the variance
    return pow( variance , 0.5 )

def correlation( x, y ):
    n = len( x )
    if n != len( y ):
        print "Uh-oh! x and y must be paired values!"
        return 0.0
    # Compute the numerator
    prod_pairs = 0
    for i in range( 0, n ):
        prod_pairs = prod_pairs + ( x[ i ] * y[ i ] )
    numerator = n * prod_pairs - ( sum( x ) * sum( y ) )
    # Compute the denominator
    x_square = 0
    for i in range( 0, n ):
        x_square = x_square + pow( x[ i ], 2 )
    y_square = 0

```

```

    for i in range(0,n):
        y_square = y_square + pow(y[i],2)
    denom_term1 = ((n*x_square)-pow(sum(x),2))
    denom_term2 = ((n*y_square)-pow(sum(y),2))
    denominator = pow((denom_term1*denom_term2),0.5)
    return numerator/denominator

bpdata = csvfile.CSVfile("BritishPetroleum-BP-table.csv")

#Let's get the 1990 year.
bpcloses = []
for row in bpdata.dataReader:
    if row['Date'].endswith('90'):
        bpcloses.append(csvfile.number(row['Close']))

amdata = csvfile.CSVfile("Exxon-Mobile-XOM-table.csv")

#Let's get the 1990 year.
amcloses = []
for row in amdata.dataReader:
    if row['Date'].endswith('90'):
        amcloses.append(csvfile.number(row['Close']))

print "BP closing values:",bpcloses
print "average",average(bpcloses)
print "number",len(bpcloses)
print "standard deviation",std_dev(bpcloses)

print "Exxon-Mobil (American) closing values:",amcloses
print "average",average(amcloses)
print "number",len(amcloses)
print "standard deviation",std_dev(amcloses)

print "Correlation is ",correlation(bpcloses,amcloses)

```

A.5 Text Analysis: Shakespeare or Bacon?

viztext.py

Program Example #7

```
import re
```

```

def highlight(basename, pattern):
    file = open(basename+".txt", "rt")
    text=file.read()
    file.close()
    # Now make the new one
    newpat = '<font color="red">'+pattern+'</font>'
    html = open(basename+".html", "wt")
    html.write("<html><title>"+basename+"</title>\n")
    html.write('<body bgcolor="black">')
    html.write('<font size=1 color=black>')
    newtext=text.replace(pattern, newpat)
    html.write(newtext)
    html.write("</body>")
    html.close()

def highlightCapitals(basename):
    file = open(basename+".txt", "rt")
    text=file.read()
    file.close()
    # Now make the new one
    html = open(basename+".html", "wt")
    html.write("<html><title>"+basename+"</title>\n")
    html.write('<body bgcolor="black">')
    html.write('<font size=1 color=black>')
    newtext=re.sub(r"(\b[A-Z][a-zA-Z]*)", r'<font color="red">\1</font>', text)
    html.write(newtext)
    html.write("</body>")
    html.close()

```

*Program Example #8***counttext.py**

```

import re

def countText(basename, pattern):
    file = open(basename+".txt", "rt")
    text=file.read()
    file.close()
    # Break it up by paragraphs
    newtext=text.split('\n\n') #Two returns = paragraph
    # Now, count the number of 'the's in the paragraph
    ret = []
    for s in newtext:
        ret.append(s.count(pattern))
    return ret

```

```
def countCapitals(basename):
    file = open(basename+".txt","rt")
    text=file.read()
    file.close()
    # Break it up by paragraphs
    newtext=text.split('\n\n')
    # Count the capitals
    ret = []
    for para in newtext:
        match = re.split(r"\b[A-Z][a-zA-z]*",para)
        ret.append(len(match)-1)
    return ret
```

A.6 Hypothesis Testing: Does the unemployment rate make the President?

Program Example #9

electoral.py – t-test and ANOVA predicting electoral results

```
from pylab import *
import csvfile

def average(sequence):
    return (1.0*sum(sequence))/len(sequence)

def variance(sequence):
    ave = average(sequence)
    # Compute the mean squared difference
    diffs = 1.0
    for num in sequence:
        diffs = diffs + pow((ave-num),2)
    # Compute the variance
    variance = diffs/(len(sequence)-1)
    return variance

def ttest(seq1,seq2):
    x1 = average(seq1)
    x2 = average(seq2)
    s1 = variance(seq1)
    s2 = variance(seq2)
    n1 = len(seq1)
    n2 = len(seq2)
    pooleds = (((n1-1)*s1)+((n2-1)*s2))/((n1+n2)-2)
    t=(x1-x2)/pow(pooleds*((1.0/n1)+(1.0/n2)),0.5)
```

```

    return t

def anova(seq1, seq2):
    sum1 = sum(seq1)
    sum2 = sum(seq2)
    grandtotal = sum1 + sum2
    n1 = len(seq1)
    n2 = len(seq2)
    totalsize = n1+n2
    x1 = average(seq1)
    x2 = average(seq2)
    grandmean = float(grandtotal)/totalsize
    SSB = (n1*pow(x1-grandmean,2))+(n2*pow(x2-grandmean,2))
    SSW = 0
    for i in seq1:
        SSW=SSW+pow(i-x1,2)
    for i in seq2:
        SSW=SSW+pow(i-x2,2)
    MSB=SSB/1
    MSW=float(SSW)/(totalsize-2)
    return MSB/MSW

unemdata = csvfile.CSVfile("USUnemploymentRate.csv")
months=['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec']

#Let's get 1996 year.
rates1996 = []
# getRows returns a set. We want just one, the 0th
row1996 = unemdata.getRows('Year', '1996')[0]
for item in months:
    rates1996.append(csvfile.number(row1996[item]))

unemdata.rewind()
#Let's get 2000 year.
rates2000 = []
row2000 = unemdata.getRows('Year', '2000')[0]
for item in months:
    rates2000.append(csvfile.number(row2000[item]))

unemdata.rewind()
#Let's get 2004 year.
rates2004 = []
row2004 = unemdata.getRows('Year', '2004')[0]
for item in months:
    rates2004.append(csvfile.number(row2004[item]))

print "1996 results"
print "average", average(rates1996)
print "variance", variance(rates1996)
print "number", len(rates1996)

```

**A.6. HYPOTHESIS TESTING: DOES THE UNEMPLOYMENT RATE
MAKE THE PRESIDENT?**

75

```
print "2000 results"  
print "average", average(rates2000)  
print "variance", variance(rates2000)  
print "number", len(rates2000)  
  
print "2004 results"  
print "average", average(rates2004)  
print "variance", variance(rates2004)  
print "number", len(rates2004)  
  
print "1996–2000 anova", anova(rates1996, rates2000)  
print "2000–2004 anova", anova(rates2000, rates2004)
```


Bibliography

[Levitt and Dubner, 2005] Levitt, S. D. and Dubner, S. J. (2005). *Freakonomics: A rogue economist explores the hidden side of everything*. HarperCollins, New York, NY.

Index

- alpha, 59
- alpha value, 34
- Analysis of Variance, 59
- ANOVA, 59
- append, 11
- arange, 17
- average, 25, 26
- average of the squared differences, 27

- background, 42
- Bacon, Francis, 41
 - The Advancement of Learning, 41
 - The Essays Of, 41
- BP, 25
- British Petroleum, 27

- causal, 35
- cd, 5, 19
- change directory, 5
- class, 9
- close, 42
- color, 19
- Comma Separated Values, 6
- Command Prompt, 2
- correlation, 31, 35
- count, 50
- CSV, 6
- csv.DictReader, 7
- csv.reader, 6

- dataReader, 26
- degrees of freedom, 35
 - t-test, 57
- descriptive statistics, 25

- df, 35
- df1, 60
- df2, 60
- dictionary, 8
- directory, 5
 - changing, 5
- distribute, 16
- dot operator, 6

- Encapsulated Postscript, 14
- end, 45
- EPS, 14
- Exxon, 27

- f statistic, 59
- fields, 9
- files
 - reading, 5, 42
 - writing, 5
- find, 42
- floating point numbers, 17
- folder, 5
- font size, 42
- for, 11
- from-import, 6

- get, 8
- getColumn, 8, 11
 - defined, 11
- getRows, 8, 10
 - defined, 10
 - usage, 8
- global, 6
- grand mean, 60
- grand total, 60
- grouped, 48

- help, 22, 29
- highlight, 42
- histogram, 29
- HTML background, 42
- import, 6, 19
 - for running files, 19
 - from, 6
- init method, 10
- instance variable, 10
- instance variables, 9
- integers, 17
- IPython, 19
- iPython, 2
- JPEG, 14
- legend, 21
 - location, 21
- linestyle, 19
- list, 10
- loc, 21
- Macbeth, 41
- map, 11
- marker, 19
- markeredgecolor, 19
- markerfacecolor, 19
- markersize, 19
- match, 44
- match object, 44
- Matplotlib, 13
 - usage, 13
- mean, 25
 - differences, 55
- means, 55
- memory, 38
- methods, 9
- Mobil, 27
- module, 6
- MSB, 60
- MSW, 60
- next, 10
- None, 44
- normal, 37
- number, 10
- NumPy, 1, 36
- objects, 9
- one-tailed, 34
- open, 42
- open(), 5
- paired, 32
- Pearson Product Moment Correlation, 34
- plot, 13, 17
 - saving, 14
 - showing, 14
 - usage, 13
- PNG, 14
- pooled sample variance, 56
- pow, 28
- print, 11
 - for debugging, 21
- python, 5
- raise, 14
- raw mode, 47
- re, 44
- read, 42
- read(), 5
- reader, 7
- reading text files, 42
- regular expressions, 44
 - grouping, 48
 - match, 44
- reload, 6, 19
- replace, 42
- rewind, 9
- Romeo and Juliet, 41
- run, 19
 - via import, 19
- savefig, 14
- SciPy, 1
- self, 10
- Shakespeare, William, 41
- shape, 30
- show(), 14
- significance chance, 34
- significance level, 34

- significant, 34
- sin, 17
- slicing, 13
 - usage, 16
- slope, 24
- sort, 15
- split, 48
- spread, 27
- SSB, 60
- SSW, 60
- standard deviation, 27
- standard_normal, 37
- start, 45
- Students T-Test, 56
- sub, 48
- subplot, 23
- sum, 25

- t statistic, 59
- T-Test, 56
- t-test
 - degrees of freedom, 57
- Terminal, 3
- textual analysis, 41
- The Advancement of Learning, 41
- The Essays of Francis Bacon, 41
- title, 19
- total sample size, 60
- traceback, 14
- tuple, 19
- two-tailed, 34

- ufunc, 17
- universal function, 17

- variance, 27

- WinEdt, 18

- xlabel, 19
- XOM, 27

- ylabel, 19