Python for Data Analysis, Statistical Modelling and Optimization

v.0.31

Jesus Perez Colino

January, 2014

CONTENTS

0.1 Something about the Author...

Documentation prepared by **Jesus Perez Colino**. Version 0.31,Released 15/01/2014, Alpha

- Comments and suggestions are always welcome. You can contact me by email: `jpcolino@gmail.com`

- **Acknowledge**: To create this document I have inserted many examples from many people and many places. Specially thanks to **Edward Tufte** from Stanford, **Hanspeter Pfister**, **Joe Blitzstein** and **Chris Beaumont** from Harvard University for some of the examples and, of course, big thanks to the people who develop **Numpy**, **Scipy**, **IPython**, **pandas**, **Scikit-learn**, **Statsmodels** and **CVXOpt** to leave it for everybody for free.

- This work is offered for free, with the hope that it will be useful. Enjoy!

CHAPTER

ONE

SOME BASICS BEFORE TO START

1.1 Why Python for Data Analysis?

- Python is great for scripting and applications.
- The `pandas` library offers improved Time Series and Data Framework support.
- Easy Web Scraping, web APIs
- Strong High Performance Computation support
  - Load balanceing tasks
  - MPI, GPU
  - MapReduce
- Strong support for abstraction
  - Intel MKL
  - HDF5
- IPhyton (Notebook) for presentation and communication of 'reproducible' results

1.2 Why not Python?

- R, MATLAB are great... and C++ is still the King. Julia is promising.
- Slow compared with C++ or Fortran
- Visualisation
  - Matplotlib is 10 years old now
  - Rob Story's vicent
  - Bokeh
  - R's ggplot2 is amazing

1.3 Introductory References

There is a huge amount of info and examples in Internet. I didn't come up with all the examples!

- Python Tutorial
- Learn Python the Hard Way
- IPython notebook documentation or the IPython tutorial
- Python for Data Analysis
- J.R. Johansson's Lectures in Scientific Computing, excellent notebooks, including Scientific Computing with Python and Computational Quantum Physics with QuTiP lectures;
- XKCD style graphs in matplotlib;

- A collection of Notebooks for using IPython effectively
- A gallery of interesting IPython Notebooks

1.4 What You Need to Install

To do this course, you need a Python package installed that should includes:

- Python version 2.7;
- Numpy, the core numerical extensions for linear algebra and multidimensional arrays;
- Scipy, additional libraries for scientific programming;
- Matplotlib, excellent plotting and graphing libraries;
- Pandas, easy-to-use data structures and data analysis tools;
- Scikit-Learn, for statistical modelling and machine learning;
- IPython, with the additional libraries required for the notebook interface.

An easy to install for Mac, Windows, and Linux, with all these packages is the Anaconda Python Distribution, from Continuum Analytics

Alternatives:

- **Linux:** Most distributions have an installation manager. Redhat has yum, Ubuntu has apt-get.
- **Mac:** Macports
- **Windows:** Anaconda, PythonXY or Canopy
- **Cloud:** Check out Wakari, from Continuum Analytics, which is a cloud-based IPython notebook.

Let us check, first, if every package is correctly installed in your computer, let us install some functions in advance (we will use it later). Go to the **next cell**, click on with the mouse and press **Shift-Intro** to execute the next cell.

In [92]:
```
%matplotlib inline
from collections import defaultdict
import json

import numpy as np
import scipy as sp
from scipy import stats
import matplotlib.pyplot as plt
import pandas as pd
import IPython
from matplotlib import rcParams
import matplotlib.cm as cm
import matplotlib as mpl

#colorbrewer2 Dark2 qualitative color table
dark2_colors = \
[(0.10588235294117647, 0.6196078431372549, 0.4666666666666667),
(0.8509803921568627, 0.37254901960784315, 0.00784313725490196),
(0.4588235294117647, 0.4392156862745098, 0.7019607843137254),
(0.9058823529411765, 0.1607843137254902, 0.5411764705882353),
(0.4, 0.6509803921568628, 0.11764705882352941),
(0.9019607843137255, 0.6705882352941176, 0.00784313725490196),
(0.6509803921568628, 0.4627450980392157, 0.11372549019607843)]

rcParams['figure.figsize'] = (10, 6)
rcParams['figure.dpi'] = 150
rcParams['axes.color_cycle'] = dark2_colors
rcParams['lines.linewidth'] = 2
rcParams['axes.facecolor'] = 'white'
rcParams['font.size'] = 14
```

```
rcParams['patch.edgecolor'] = 'white'
rcParams['patch.facecolor'] = dark2_colors[0]
rcParams['font.family'] = 'StixGeneral'


def remove_border(axes=None, top=False, right=False,
                  left=True, bottom=True):
    """
    Minimize chartjunk by stripping out unnecessary
    plot borders and axis ticks

    The top/right/left/bottom keywords toggle
    whether the corresponding plot border is drawn
    """
    ax = axes or plt.gca()
    ax.spines['top'].set_visible(top)
    ax.spines['right'].set_visible(right)
    ax.spines['left'].set_visible(left)
    ax.spines['bottom'].set_visible(bottom)

    #turn off all ticks
    ax.yaxis.set_ticks_position('none')
    ax.xaxis.set_ticks_position('none')

    #now re-enable visibles
    if top:
        ax.xaxis.tick_top()
    if bottom:
        ax.xaxis.tick_bottom()
    if left:
        ax.yaxis.tick_left()
    if right:
        ax.yaxis.tick_right()

pd.set_option('display.width', 500)
pd.set_option('display.max_columns', 100)

from sys import version
import time as t
print '='*70
print 'Python version:     ' + version
print 'Numpy version:      ' + np.__version__
print 'Pandas version:     ' + pd.__version__
print 'Matplotlib ver:     ' + mpl.__version__
print 'IPython version:    ' + IPython.__version__
direct = %pwd
print 'Working directory:  ' + direct
print '='*70
now = t.asctime()
print 'Today is ' + now + ' ... AND WE ARE READY TO GO!! '
print '='*70
```

```
======================================================================
Python version:     2.7.5 |Anaconda 1.8.0 (x86_64)| (default, Oct 24 2013, 07:02:2
[GCC 4.0.1 (Apple Inc. build 5493)]
Numpy version:      1.7.1
Pandas version:     0.12.0
Matplotlib ver:     1.3.1
IPython version:    1.1.0
Working directory:  /Users/JPC
======================================================================
Today is Thu Dec 12 23:10:39 2013 ... AND WE ARE READY TO GO!!
======================================================================
```

CHAPTER

TWO

GETTING STARTED WITH PYTHON

"Let us change our traditional attitude to the construction of programs:
Instead of imagining that our main task is to instruct a computer what to do,
let us concentrate rather on explaining to humans what we want the computer to do."
– Donald E. Knuth Literate Programming, 1984

2.1 What is Python?

Python is a general-purpose, high-level and multi-paradigm programming language.

Its design philosophy emphasizes code readability, and its syntax allows programmers to express concepts in fewer lines of code than would be possible in languages such as C++.

- Simple, clean syntax
- Easy to learn
- Interpreted or Compiled
- Dynamically typed
- Multi-platform: Linux, Mac, and Windows
- Free, as in beer and speach
- Expressive: do more with fewer lines of code
- Lean: **modules**

2.2 History

- Created by Dutch programmer **Guido van Rossum**, 1989, declared *Benevolent Dictator for Life (BDFL)* by Python community.
- Conceived as a successor to ABC language and interfacing with Amoeba distributed OS.
- Initially the syntax came from `C`, but with less exceptions and special cases
- **Multi-paradigm**: accept object-oriented, structured and functional programming
- It brings from `LISP` (functional prog.) the `lambda`, `reduce`, `filter`, and `map`
- `Modula-3` inspired keyword arguments and `imports`
- **`Python 2.0`** was released on Oct-2000, with many major new features including a full garbage collector and support for Unicode.
- **`Python 3.0`** was a backwards-incompatible version released on Dec-2008. Many of its major features have been backported to the backwards-compatible `Python 2.6` and `Python 2.7`

2.3 What can you do with Python?

- Scripting language
  - os support
  - glue existing applications
- Scientific abstraction
  - Use highly optimized C and Fortran libraries
  - Intel MKL, HDF5, Blas, Lapack, MPI, Cuda
- Data Analysis
- Visualisation
- Scrape websites
- Build websites

2.4 Performance

- Minimal time required to develop code
- Rapid prototyping
- Interpreted and dynamically typed means it's slower
  - Use optimized libraries: e.g. numpy , scipy
  - Find the bottleneck and write it in C/C++/Fortran: e.g. f2py, cython
  - Just-in-time: e.g. numba
- Implementing the built-in Python modules would require some advanced programming skills in other languages

2.5 Advantages

- Interpretive
- Named function arguments
- Heterogeneous data structures
  - dictionary
  - lists
- No memory management
- Print structures
- Packages: large community of support
- Modular: great for larger projects (modules)

2.6 Modules for Data Analysis

Base:

- IPython": interactive shell
- Numpy: array and matrix library
- Scipy: scientific libraries
- Matplotlib: visualization
- Pandas: data frames
- Scikit-Learn: statistical modelling and machine learning

Additional:

- Requests: data from the web
- BeautifulSoup: parse web pages
- Pytables: fast, binary files
- Sqlite3: binding for sqlite3
- Spark: in-memory MapReduce
- Disco: MapReduce

CHAPTER

THREE

PYTHON OVERVIEW

3.1 Using Python as Calculator

If you want, you can start holding a conversation with the Python interpreter, where you can speak in expressions and it replies with evaluations. There is clearly a **read-eval-print** loop going on. Let us start with the simplest possible mathematical expressions: as an online calculator (If you're typing this into an IPython notebook, press **Shift-Enter** to evaluate a cell.)

```
In [2]: 2+2*3
```

```
Out [2]:
        8
```

```
In [3]: 7/3
```

```
Out [3]:
        2
```

In Python, data takes the form of *objects* - either built-in objects that Python provides, or objects we create using Python tools. But in Python, numbers are not really a single object type, but a category of similar types. Python has all the well-know data types that you can see in any other language (`int, long, float, double, booleans, char, ...`) but does not require you to state why type of data need to be stored in any particular variable. Let us start with the simplest different numeric data types: **integers**, or *whole numbers* to the non-programming world, and **floating-point numbers**, also known (incorrectly) as *decimal numbers* with the optional signed exponent (implemented in C as `doubles`) and therefore get as much precision as the C compiler used to build the Python interpreter gives to doubles.

Python integer division, like C or Fortran, truncates the remainder and returns an integer. In Python 3, returns a floating point number.

```
In [4]: 7/3.
```

```
Out [4]:
        2.3333333333333335
```

```
In [5]: 7/float(3)
```

Out [5]:
```
2.3333333333333335
```

Python has a huge number of libraries included with the distribution. To keep things simple, most of these variables and functions are not accessible in a Python session. Instead, you have to **`import`** the name.

For example, there is a `math` module containing many useful functions. To access, say, the square root function, you can either first

```
from math import sqrt
```

and then

In [6]:
```
from math import sqrt
sqrt(10000)
```

Out [6]:
```
100.0
```

or you can simply import the full version of the math library itself

In [7]:
```
print 'Sqrt of 1000 is ', math.sqrt(1000)
#print 'hello' + 'jackass'
print 'hello' + 'jackass'
```

```
Sqrt of 1000 is  31.6227766017
hellojackass
```

In [8]:
```
import math
print(dir(math))
print '-'*120
print 'Sqrt of 1000 is ', math.sqrt(1000)
print 'hello' + 'jackass'
```

```
['__doc__', '__name__', '__package__', 'acos', 'acosh', 'asin',
'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh',
'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial',
'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'hypot', 'isinf', 'isnan',
'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'modf', 'pi', 'pow',
'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
------------------------------------------------------------------------
------------------------------------------------
Sqrt of 1000 is  31.6227766017
hellojackass
```

You can define variables using the equals (=) sign:

In [9]:
```
width = 20.
print type(width)
length = 30
area = length*width
print area
type(area)
```

```
<type 'float'>
600.0
```

Out [9]:
```
float
```

But the output is not necessary a integer number. Check below how to define the format of the outputs:

In [10]:
```
from math import pi
print "Pi as a decimal = %d" % pi
print "Pi as a float = %f" % pi
print "Pi with 4 decimal places = %.4f" % pi
print "Pi with overall fixed length of 10 spaces, \
       with 6 decimal places = %10.6f" % pi
print "Pi as in exponential format = %e" % pi
```

```
Pi as a decimal = 3
Pi as a float = 3.141593
Pi with 4 decimal places = 3.1416
Pi with overall fixed length of 10 spaces, with 6 decimal places =
3.141593
Pi as in exponential format = 3.141593e+00
```

3.2 Data Structures

- Strings
- Lists
- Tuples
- Dictionaries

Strings

Strings are lists of printable characters, and can be defined using either double or single quotes. Substring selection, concatenation and repetition are all supported as a methods of the object string. Also you can expect methods as `find, startwith, endwith, replace` and so forth, because string is a class. Here you have some simple examples:

In [11]:
```
'Hello, Dusseldorf'
```

Out [11]:
```
'Hello, Dusseldorf'
```

In [85]:
```
greeting = "Hello, Dusseldorf!"
print greeting

# Properties of the string
print 'Type of variable:',type(greeting)
print 'Lenght of variable:',len(greeting)

# Find and replace
greeting = greeting.replace('Dusseldorf','Cologne')
print greeting
```

```
print 'And Cologne is in position ', greeting.find('Cologne')

print'='*80

# An interesting example...

statement = ['Hello', 'Dusseldorf']
print 'Yes, I am in',statement[1]
print 'and now, I am in',greeting[7:-1]
```

```
Hello, Dusseldorf!
Type of variable: <type 'str'>
Lenght of variable: 18
Hello, Cologne!
And Cologne is in position  7
================================================================================
Yes, I am in Dusseldorf
and now, I am in Cologne
```

And in case that you need help...

In [14]:
```
help(len)

# or in iPhython

len?
```

```
Help on built-in function len in module __builtin__:

len(...)
    len(object) -> integer

    Return the number of items of a sequence or mapping.
```

Lists

Python has a data type called **lists** defined as a sequential container of objects. This is just a way to collect all type of objects in a list that is *mutable* (changeable or read-write). List are written as a comma-separated series of values enclosed in square brackets.

In [27]:
```
days_of_the_week = ["Sunday","Monday","Tuesday",\
                    "Wednesday","Thursday","Friday","Saturday"]
print days_of_the_week
```

```
['Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday',
'Saturday']
```

The same slicing that was available to us with strings actually works with the list. You can access members of the list using the index of that item:

In [28]:
```
print days_of_the_week[4]
print days_of_the_week[-2]
print 'Number of days of a week:',len(days_of_the_week)
```

```
Thursday
Friday
Number of days of a week: 7
```

In [29]:
```
languages = ["Fortran","C","C++"]
print type('languages')

# Appending new elements in the list
languages.append("Python")
print languages
print 'Last two languages:',languages[:len(languages)-2]
languages.append("Haskell")
print 'Last two languages:',languages[len(languages)-2:]

# Removing the last element in the list of languages
languages.remove(languages[len(languages)-1])
print 'Last two languages:',languages[len(languages)-2:]
```

```
<type 'str'>
['Fortran', 'C', 'C++', 'Python']
Last two languages: ['Fortran', 'C']
Last two languages: ['Python', 'Haskell']
Last two languages: ['C++', 'Python']
```

The **`range(start,stop,step)`** command is a convenient way to make sequential lists of numbers:

In [30]:
```
l = range(-50,50,10)
print l
```

```
[-50, -40, -30, -20, -10, 0, 10, 20, 30, 40]
```

In [31]:
```
evens = range(0,20,2)
print evens

# Printing elements of the list
print evens[3]

# Sorting elements of the list
print evens.sort(reverse=True)
```

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
6
None
```

In [35]:
```
["Today",7,99.3,""]
```

Out [35]:
```
['Today', 7, 99.3, '']
```

In [36]:
```
help(len)
```

```
Help on built-in function len in module __builtin__:

len(...)
```

```
        len(object) -> integer

        Return the number of items of a sequence or mapping.
```

```
In [37]: len(evens)
```

```
Out [37]:
         10
```

A list literal enclosed in square brackets ([...]) can be a simple list of expressions or a list comprehension expression of the following form:

```
[expression for expr1 in iterable1 [if condition1] for expr2 in iterable2 [if
condition2] ...  for exprN in iterableN [if conditionN] ]
```

List comprehensions construct result lists: they collect all values of expression, for each iteration of all nested for loops, for which each optional condition is true. The second through nth for loops and all if parts are optional, and expression and condition can use variables assigned by nested for loops. Names bound inside the comprehension are created in the scope where the comprehension resides.

Comprehensions are similar to the map() built-in function:

```
In [17]: [ord(x) for x in 'spam']
```

```
Out [17]:
         [115, 112, 97, 109]
```

```
In [18]: map(ord,'spam')
```

```
Out [18]:
         [115, 112, 97, 109]
```

```
In [19]: [x**2 for x in range(5)]
```

```
Out [19]:
         [0, 1, 4, 9, 16]
```

```
In [20]: map((lambda x: x**2), range(5))
```

```
Out [20]:
         [0, 1, 4, 9, 16]
```

Comprehensions with conditions are similar to filter:

```
In [21]: [x for x in range(5) if x % 2 == 0]
```

```
Out [21]:
         [0, 2, 4]
```

```
In [22]: filter((lambda x: x % 2 == 0), range(5))
```

```
Out [22]:
          [0, 2, 4]
```

Tuples

A **tuple** is a sequence object like a list or a string. It's constructed by grouping a sequence of objects together with commas, either without brackets, or with parentheses. Tuples are *inmutable or read-only* arrays of object references, meaning that they cannot be modified once created (you can't append to them or change the elements of them). Tuples are written as comma-separated series of values enclosed in parentheses.

```
In [34]:
# An empty tuple
no_language = ()

# A three-item tuple
languages_t = ("Fortran","C","C++")
print languages_t
print 'Size:',len(languages_t),'||','Type:',type(languages_t)

# Unpacking the tuple
x,y,z = languages_t
print z

# A nested tuple
lang_added = (languages_t,('Haskell', 'Lisp'))
print lang_added
```

```
('Fortran', 'C', 'C++')
Size: 3 || Type: <type 'tuple'>
C++
(('Fortran', 'C', 'C++'), ('Haskell', 'Lisp'))
```

Tuples are useful anytime you want to group different pieces of data together in an object, but don't want to create a full-fledged class (see below) for them. For example, let's say you want the Cartesian coordinates of some objects in your program.

Dictionaries

A **Dictionary** is a collection of {key:value} pairs, also called associative arrays or hash maps in other languages

```
In [59]:
ages = {"Rick": 46, "Bob": 86, "Fred": 21}
print "Rick's age is ",ages["Rick"]
```

```
Rick's age is  46
```

```
In [112]:
ages2 = dict(Rick=46,Bob=86,Fred=20)
print "Rick's age is ",ages2["Rick"]
```

```
Rick's age is  46
```

```
In [99]:
data = {}
data['k1']=True
data['k2']=2
data['k3']=3.0
```

```
print data
print 'Size:',len(data),'||','Type:',type(data)

# Add-in a new element in the list
data['k4']=100
print 'New data:',data['k4']
print data

# Getting data using the label or index
x = 'k4'
print 'Retriving from ',x, 'the data', data.get('k4',0)
```

```
{'k3': 3.0, 'k2': 2, 'k1': True}
Size: 3 || Type: <type 'dict'>
New data: 100
{'k3': 3.0, 'k2': 2, 'k1': True, 'k4': 100}
Retriving from  k4 the data 100
```

3.3 Control Flow

- `if`,`elif` and `else`
- `for` loops
- `while` loops
- `exception` handling
- Iteration, Indetation and Blocks
- Comprenhension

`if`, `elif`, and `else`

The `if` statement selects from among one or more actions (statement blocks), and it runs the suite associated with the first `if` or `elif` test that is `true`, or the `else` suite if all are `false`.

In [61]:
```
car = 10
bus = 2.5
cash = 1.5

if car < cash:
    print 'I can drive'

if bus < cash and car > cash:
    print "I'll take the bus"

if bus > cash:
    print 'Walking.'
```

```
Walking.
```

In [62]:
```
if car < cash:
    print 'Enjoy your car ride'
elif bus < cash:
    print 'Another one rides the bus'
else:
    print 'Walking..'
```

```
Walking..
```

In [63]:
```
action = 'car' if car < cash else 'Walking...' #one-liner

print action
```

```
Walking...
```

The `for` loop

The `for` loop is a sequence (or other `iterable`) iteration that assigns items in `iterable` to `target` and runs the first suite for each. The `for` statement runs the else suite if the loop exits without hitting a break statement. `target` can be anything that can appear on the left side of an = assignment statement (e.g. `for (x, y) in tuplelist:`).

In [64]:
```
numbers = [1,2,3,4,5]

for i in numbers:
    print i,

print '  '

for num in numbers:
    print num,
```

```
1 2 3 4 5
1 2 3 4 5
```

In [130]:
```
for i in xrange(10):
    print i,
```

```
0 1 2 3 4 5 6 7 8 9
```

In [67]:
```
for letter in "Sunday":
    print letter
```

```
S
u
n
d
a
y
```

In [134]:
```
for i in xrange(10):
    print i, i%2
    if i % 2:
        continue
    print "still here",
```

```
0 0
still here 1 1
2 0
```

```
still here 3 1
4 0
still here 5 1
6 0
still here 7 1
8 0
still here 9 1
```

In [135]:
```
for i in xrange(10):
    if i == 5:
        break
    print i,
```

```
0 1 2 3 4
```

The `break` statement immediately exits the closest enclosing `if`, `while` or `for` loop statement, skipping its associated `else` (if any). Conversely, the `continue` statement immediately goes to the top of the closest enclosing `if`,`while` or `for` loop statement; it resumes in the loop header line.

In [68]:
```
index = 0
for day in days_of_the_week:
    print index, days_of_the_week[index], day
    index += 1
```

```
0 Sunday Sunday
1 Monday Monday
2 Tuesday Tuesday
3 Wednesday Wednesday
4 Thursday Thursday
5 Friday Friday
6 Saturday Saturday
```

In [147]:
```
for i, day in enumerate(days_of_the_week):
    print i, days_of_the_week[i], day
```

```
0 Sunday Sunday
1 Monday Monday
2 Tuesday Tuesday
3 Wednesday Wednesday
4 Thursday Thursday
5 Friday Friday
6 Saturday Saturday
```

`while` loop

In [136]:
```
count = 0
while (count < 10):
   print count,
   count += 1
```

```
0 1 2 3 4 5 6 7 8 9
```

exception handling

In [38]:
```
num = '123.4d'
print float(num)
```

```
---------------------------------------------------------------------

ValueError                    Traceback (most recentcall last)

    <ipython-input-38-320d5b550040> in <module>()
      1 num = '123.4d'
----> 2 print float(num)

    ValueError: invalid literal for float(): 123.4d
```

In [39]:
```
try:
    print float(num)
except (ValueError, TypeError),e:
    print e
    print 'DO SOMETHING ELSE HERE'
```

```
invalid literal for float(): 123.4d
DO SOMETHING ELSE HERE
```

In [139]:
```
values = [1,2,3]
try:
    print float(values)
except ValueError:
    print "nope"
except TypeError:
    print float(values[0])
```

```
1.0
```

In [140]:
```
try:
    print float("454c")
    print float([1,2,3])
except:
    print "error"
```

```
error
```

Iteration, Indentation, and Blocks

One of the most useful things you can do with lists is to *iterate* through them, i.e. to go through each element one at a time. To do this in Python, we use the **for** statement:

In [105]:
```
for day in days_of_the_week:
    print day
```

```
Sunday
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
```

This code snippet goes through each element of the list called **days_of_the_week** and assigns it to the variable **day**. It then executes everything in the indented block (in this case only one line of code, the print statement) using those variable assignments. When the program has gone through every element of the list, it exists the block.

Every programming language defines blocks or scopes of code in some way. In Fortran, one uses END statements (ENDDO, ENDIF, etc.) to define code blocks. In C, C++, and Perl, one uses curly braces {} to define the scope or blocks.

Python uses a colon (":"), followed by **indentation level** to define code blocks. Everything at a higher level of indentation is taken to be in the same block.

In [106]:
```
for day in days_of_the_week:
    statement = "Today is " + day
    print statement
```

```
Today is Sunday
Today is Monday
Today is Tuesday
Today is Wednesday
Today is Thursday
Today is Friday
Today is Saturday
```

In [107]:
```
for i in range(20):
    print "The square of ",i," is ",i*i
```

```
The square of  0  is  0
The square of  1  is  1
The square of  2  is  4
The square of  3  is  9
The square of  4  is  16
The square of  5  is  25
The square of  6  is  36
The square of  7  is  49
The square of  8  is  64
The square of  9  is  81
The square of  10  is  100
The square of  11  is  121
The square of  12  is  144
The square of  13  is  169
The square of  14  is  196
The square of  15  is  225
The square of  16  is  256
The square of  17  is  289
The square of  18  is  324
```

```
The square of  19  is  361
```

Comprenhension

In [155]:
```
days_of_the_week = ["Sunday","Monday","Tuesday",\
                    "Wednesday","Thursday","Friday",\
                    "Saturday"]

days_of_the_week.append("My_Day")
print days_of_the_week

days_of_the_week.remove("My_Day")
print days_of_the_week
```

```
['Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday',
'Saturday', 'My_Day']
['Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday',
'Saturday']
```

In [158]:
```
S_days = []
for day in days_of_the_week:
    if day.startswith('S'):
        S_days.append(day.lower())
print S_days
```

```
['sunday', 'saturday']
```

3.4 Code Example 01: The Fibonacci Sequence

The Fibonacci sequence is a sequence of numbers that starts with 0 and 1, and then each successive entry is the sum of the previous two. Thus, the sequence goes 0,1,1,2,3,5,8,13,21,34,55,89,...

A very common exercise in programming books is to compute the Fibonacci sequence up to some number **n**. Please, notice the comments below.

In [1]:
```
n = 20
sequence = [0,1]
for i in range(2,n):
    # This is going to be a problem if we ever set n <= 2!
    sequence.append(sequence[i-1]+sequence[i-2])
print sequence
```

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987,
1597, 2584, 4181]
```

Comments to the code above

- *First*, we define the variable **n**, and set it to the integer 20. **n** is the length of the sequence we're going to form, and should probably have a better variable name.
- *Second*, we create a variable called **sequence**, and initialize it to the list with the integers 0 and 1 in it, the first two elements of the Fibonacci sequence. We have to create these elements "by hand", since the iterative part of the sequence requires two previous elements.

- *Third*, we then have a for loop over the list of integers from 2 (the next element of the list) to **n** (the length of the sequence). Notice that after the colon, we see a hash tag "#", and then a **comment** that if we had set **n** to some number less than 2 we would have a problem. Comments in Python start with #, and are good ways to make notes to yourself or to a user of your code explaining why you did what you did. Better than the comment here would be to test to make sure the value of **n** is valid, and to complain if it isn't; we'll try this later.
- *Fourth*, in the body of the loop, we append to the list an integer equal to the sum of the two previous elements of the list.
- *And finally*, exiting the loop (ending the indentation) we then print out the whole list.

3.5 Functions

In simple terms, a function is a device that groups a set of statements so they can be run more than once in a program?a packaged procedure invoked by name. Functions also can compute a result value and let us specify parameters that serve as function inputs and may differ each time the code is run. Coding an operation as a function makes it a generally useful tool, which we can use in a variety of contexts.

A function in Python is defined using the statement or keyword `def`, followed by a function `name`, a signature within parenthises `()`, and a colon `:`. It creates a function object and assigns it to variable `name`. The Python `def` is a true executable statement: when it runs, it creates a new function object and assigns it to a name. (Remember, all we have in Python is runtime; there is no such thing as a separate compile time.) Each call to a function object generates a new, local scope, where assigned names are local to the function call by default (unless declared global).The following code, with one additional level of indentation, is the function body.

We might want to use the Fibonacci snippet with different sequence lengths. We could cut an paste the code into another cell, changing the value of **n**, but it's easier and more useful to make a function out of the code. We do this with the `def` statement in Python:

In [6]:
```
def fibonacci(sequence_length):
    "Return the Fibonacci sequence of length *sequence_length*"
    sequence = [0,1]
    if sequence_length < 1:
        print "Fibonacci sequence only defined for length 1 or greater"
        return
    if 0 < sequence_length < 3:
        return sequence[:sequence_length]
    for i in range(2,sequence_length):
        sequence.append(sequence[i-1]+sequence[i-2])
    return sequence
```

In [7]:
```
fibonacci(12)
```

Out [7]:
```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

In [8]:
```
help(fibonacci)
```

```
Help on function fibonacci in module __main__:

fibonacci(sequence_length)
    Return the Fibonacci sequence of length *sequence_length*
```

Functions can also call themselves, something that is often called **recursion**. Here you have an example of recursion by computing the factorial function. The factorial is defined for a positive integer **n** as

$$n! = n(n-1)(n-2)\cdots 1$$

First, note that we don't need to write a function at all, since this is a function built into the standard math library. However, if we did want to write a function ourselves, we could do recursively by:

In [9]:
```
def fact(n):
    if n <= 0:
        return 1
    return n*fact(n-1)
```

In [10]:
```
fact(10)
```

Out [10]:
```
3628800
```

We can return multiple values from a function using tuples (see above):

In [11]:
```
def powers(x):
    """
    Return a few powers of x.
    """
    return x ** 2, x ** 3, x ** 4
```

In [12]:
```
powers(90)
```

Out [12]:
```
(8100, 729000, 65610000)
```

In [13]:
```
x2, x3, x4 = powers(3)

print(x3)
```

```
27
```

In [14]:
```
x2, x3, x4 = powers(3)

print(x3)
```

```
27
```

In Python we can also create unnamed functions, using the `lambda` keyword:

In [15]:
```
f1 = lambda x: x**2

# is equivalent to

def f2(x):
    return x**2
```

In [16]:
```
f1(2), f2(2)
```

Out [16]:
```
(4, 4)
```

In [17]:
```
# map is a built-in python function that allows us to treat
function as objects...

map(lambda x: x**2, range(-3,4))
```

Out [17]:
```
[9, 4, 1, 0, 1, 4, 9]
```

CHAPTER

FOUR

READING AND WRITING DATA

- Text files
- Structured Text Files
  - CSV files
  - JSON
  - HTML

4.1 Text Files

In [18]:
```
text = "When you are courting a nice girl, an hour seems like a second.\n"
text += "When you sit on a red-hot cinder, a second seems like an hour.\n"
text += "That's relativity."
```

The built-in `open()` function creates a file object, the most common file interface. File objects export the data transfer method calls in the next chapter. In Python 2.X only, the name `file()` can be used as a synonym for `open()` when creating a file object, but `open()` is the generally recommended spelling; in Python 3.0, `file()` is no longer available.

In [19]:
```
# FIRST: Create an outfile object
outfile = open('albert.txt','w')

# SECOND: Write the text data
outfile.write(text)

# THIRD: Close the outfile object
outfile.close()
```

In standard Python (CPython), file objects normally close themselves when garbage collected if still open. Because of this, temporary files (e.g., `open('name').read()`) need not be closed explicitly. To guarantee closes after a block of code exits, regardless of whether the block raises an exception, use the `try` and `finally` statement and manual closes:

In [20]:
```
infile = open('albert.txt','r')

try:
    text = infile.read()
finally:
    infile.close()

print text
```

```
When you are courting a nice girl, an hour seems like a second.
When you sit on a red-hot cinder, a second seems like an hour.
That's relativity.
```

In [21]:
```
with open('albert.txt','r') as infile:
    text = infile.read()
```

In [22]:
```
print len(text)
```

```
145
```

4.2 Structured Text Data

- CSV: comma-separated values
- JSON: JavaScript Object Notation
- XML: EXtensible Markup Language
- HTML: webAPIs with urllib2

In [23]:
```
import csv
with open('numbers.csv','w') as outfile:
    writer = csv.writer(outfile)
    writer.writerow(('first','second','third'))
    writer.writerow((1,3,6,8)) #oops
    writer.writerow((2,4,6))


with open('numbers.csv','r') as infile:
    reader = csv.reader(infile)
    for lines in reader:
        print lines
```

```
['first', 'second', 'third']
['1', '3', '6', '8']
['2', '4', '6']
```

JSON is a more flexible format than CSV for structured text

In [24]:
```
data = {'name': 'Guido van Rossum',
        'language': 'Python',
        'similar': ['c','Modula-3','lisp'],
        'users': 1000000}

import json
json.dumps(data)
```

Out [24]:
```
'{"similar": ["c", "Modula-3", "lisp"], "name": "Guido van Rossum",
"language": "Python", "users": 1000000}'
```

In [25]:
```
with open('data.json','w') as outfile:
    outfile.write(json.dumps(data))
```

In [26]:
```
with open('data.json','r') as infile:
    text = infile.read()
    data = json.loads(text)


print data
print data['name']
```

```
{u'similar': [u'c', u'Modula-3', u'lisp'], u'name': u'Guido van
Rossum', u'language': u'Python', u'users': 1000000}
Guido van Rossum
```

4.3 HTML: webAPIs with urllib2

In [27]:
```
import urllib2
response = urllib2.urlopen('http://vimeo.com/api/v2/video/57733101.json')
data = json.load(response)[0]
```

In [28]:
```
print data
print '-'*100
print data['title']
print data['url']
print data['duration']
print data['stats_number_of_plays']
```

```
{u'upload_date': u'2013-01-19 04:01:15', u'height': 360, u'duration':
143, u'id': 57733101, u'user_id': 1334563, u'stats_number_of_likes':
12, u'thumbnail_medium':
u'http://b.vimeocdn.com/ts/436/057/436057121_200.jpg', u'title': u'The
Good Man trailer', u'user_url': u'http://vimeo.com/manifestofilms',
u'width': 640, u'user_portrait_large':
u'http://b.vimeocdn.com/ps/477/830/4778306_100.jpg', u'embed_privacy':
u'anywhere', u'user_name': u'Manifesto Films', u'user_portrait_small':
u'http://b.vimeocdn.com/ps/477/830/4778306_30.jpg',
u'user_portrait_medium':
u'http://b.vimeocdn.com/ps/477/830/4778306_75.jpg', u'description':
u'Trailer for the forthcoming Manifesto Films production, The Good
Man. Co-production with Jet Black Entertainment. Starring Aidan
Gillen, Thabang Sidloyi, Kelly Campbell, Lunathi Mampofu. Written &
directed by Phil Harrison. More info at www.thegoodmanfilm.com<br
/>\r\n<br />\r\nCheck out the Variety Magazine review here:
http://www.variety.com/review/VE1117947909/', u'tags': u'film,
ireland, africa, cape town, belfast', u'stats_number_of_plays': 2819,
u'stats_number_of_comments': 0, u'thumbnail_small':
u'http://b.vimeocdn.com/ts/436/057/436057121_100.jpg',
u'thumbnail_large':
u'http://b.vimeocdn.com/ts/436/057/436057121_640.jpg', u'url':
u'http://vimeo.com/57733101', u'user_portrait_huge':
u'http://b.vimeocdn.com/ps/477/830/4778306_300.jpg'}
----------------------------------------------------------------------
The Good Man trailer
http://vimeo.com/57733101
143
```

```
2819
```

CHAPTER

FIVE

PLOTTING WITH MATPLOTLIB

First rule in statistical modelling, plot your data if you can. Python has a great plotting library called Matplotlib. The IPython notebook interface we are using for these notes has that functionality built in.

Matplotlib is an excellent 2D and 3D graphics library for generating scientific figures. Some of the many advantages of this library includes:

- Easy to get started
- Support for LaTeX formatted labels and texts
- Great control of every element in a figure, including figure size and DPI.
- High-quality output in many formats, including PNG, PDF, SVG, EPS.
- GUI for interactively exploring figures *and* support for headless generation of figure files (useful for batch jobs).

The `matplotlib.pyplot` module contains the API for functions, such as `plot`. The convention is to import it as `plt`:

```
import matplotlib.pyplot as plt
```

You can lauch IPython with `matplotlib` integration by typing:

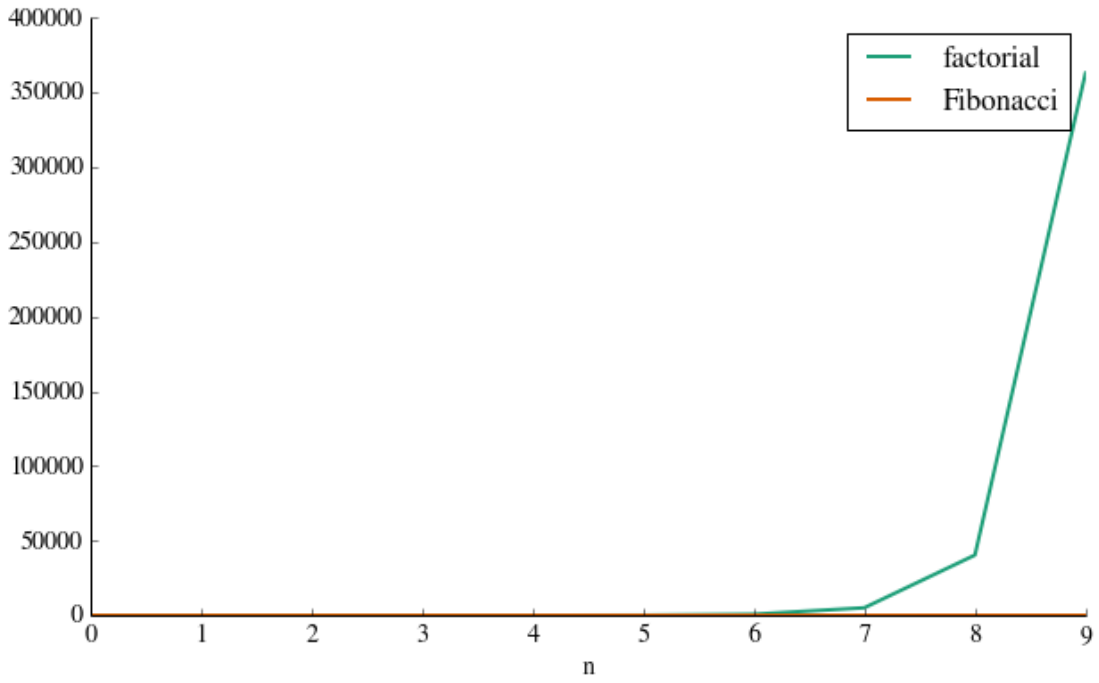
```
ipython --pylab
```

```
ipython notebook --pylab=inline
```

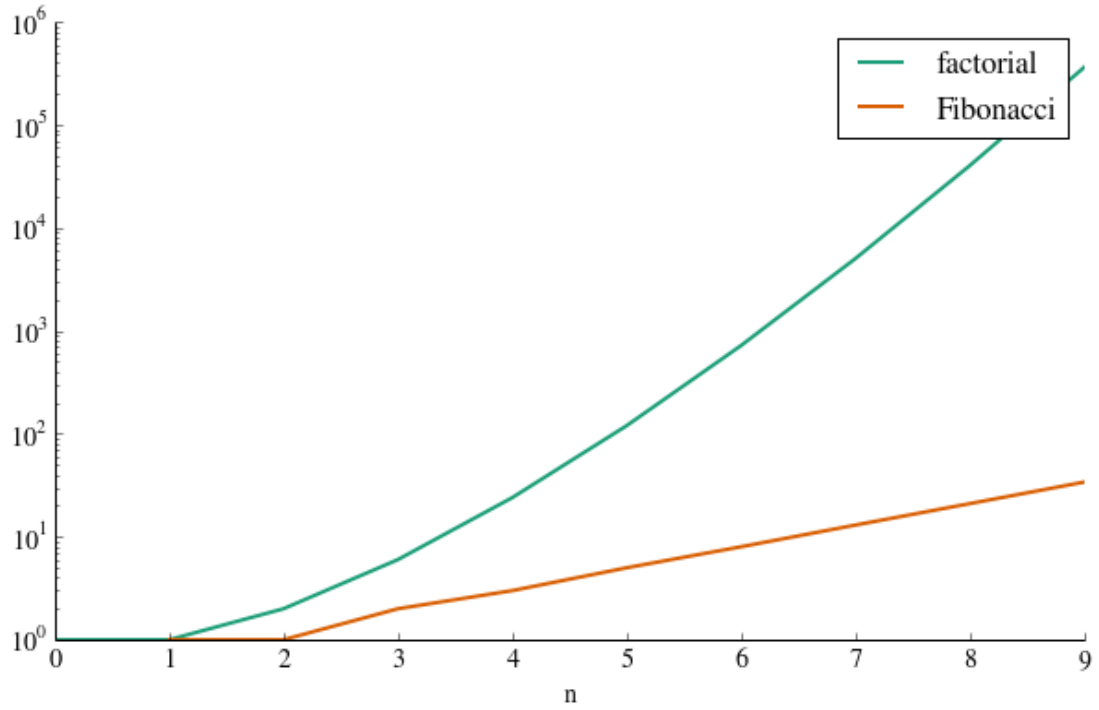
These two methods are the best ways to use matplotlib interactively. We you do this, you will have access to several `numpy` and `pyplot` in the same namespace.

As an example, we have looked at two different functions, the Fibonacci function, and the factorial function, both of which grow faster than polynomially. Which one grows the fastest? Let's plot them.

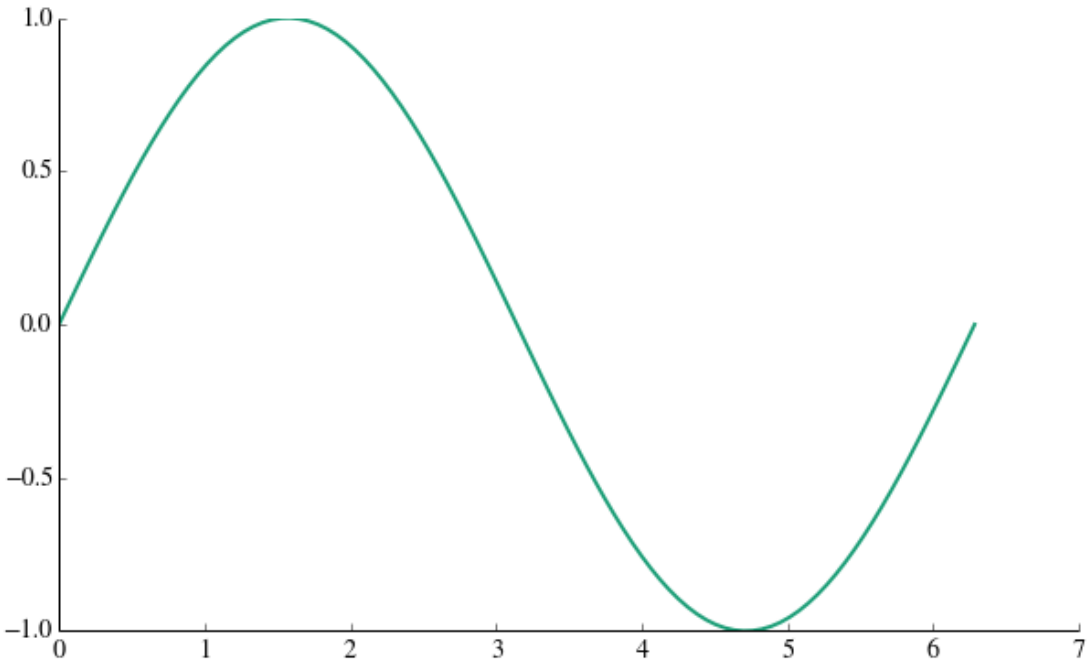
In [31]:
```
fibs = fibonacci(10)
facts = []
for i in range(10):
    facts.append(fact(i))

# figsize(8,6)
plt.plot(facts,label="factorial")
plt.plot(fibs,label="Fibonacci")
plt.xlabel("n")
plt.legend()
remove_border()
```



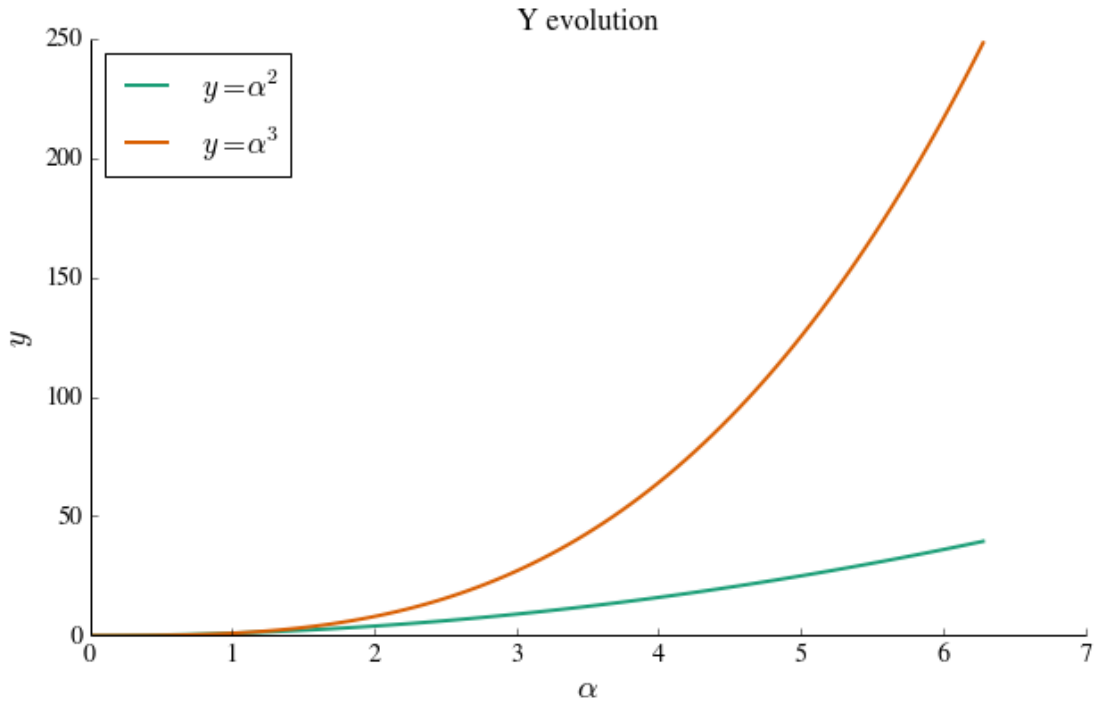
In [32]:
```
# let us plot it on a semilog plot and we can see both more clearly

plt.semilogy(facts,label="factorial")
plt.semilogy(fibs,label="Fibonacci")
plt.xlabel("n")
plt.legend()
remove_border()
```



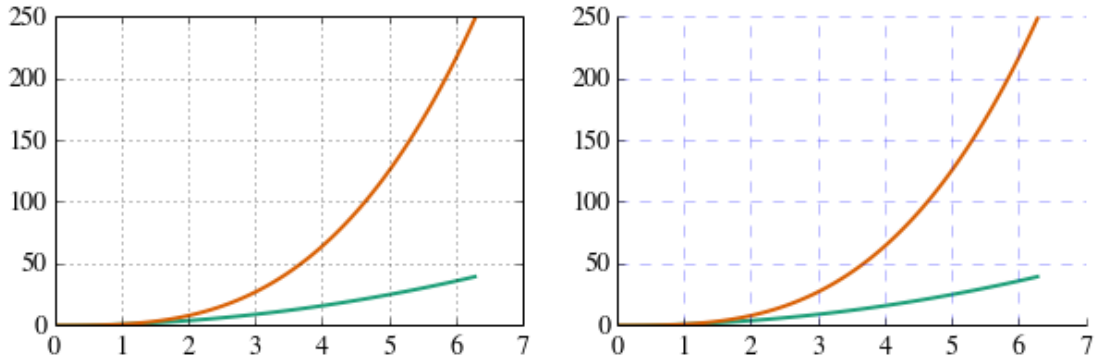
In [86]:
```
x = np.linspace(0, 2*math.pi,100)
#print x
y = np.sin(x)
#print y

plt.plot (x,y)
remove_border()
```



In [91]:
```
fig, ax = plt.subplots()

ax.plot(x, x**2, label=r"$y = \alpha^2$")
ax.plot(x, x**3, label=r"$y = \alpha^3$")
ax.set_xlabel(r'$\alpha$', fontsize=18)
ax.set_ylabel(r'$y$', fontsize=18)
ax.set_title("Y evolution")
ax.legend(loc=2); # upper left corner
remove_border()
```

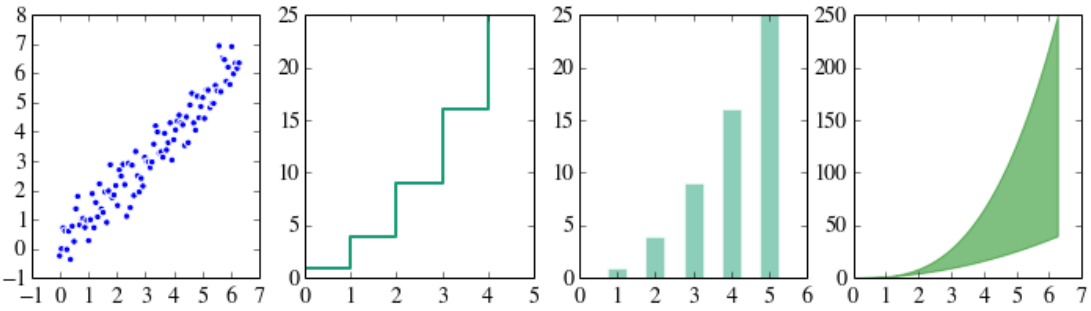


In [37]:
```
fig, axes = plt.subplots(1, 2, figsize=(10,3))

# default grid appearance
axes[0].plot(x, x**2, x, x**3, lw=2)
axes[0].grid(True)
remove_border()
# custom grid appearance
axes[1].plot(x, x**2, x, x**3, lw=2)
axes[1].grid(color='b', alpha=0.5, linestyle='dashed', linewidth=0.5)
remove_border()
```

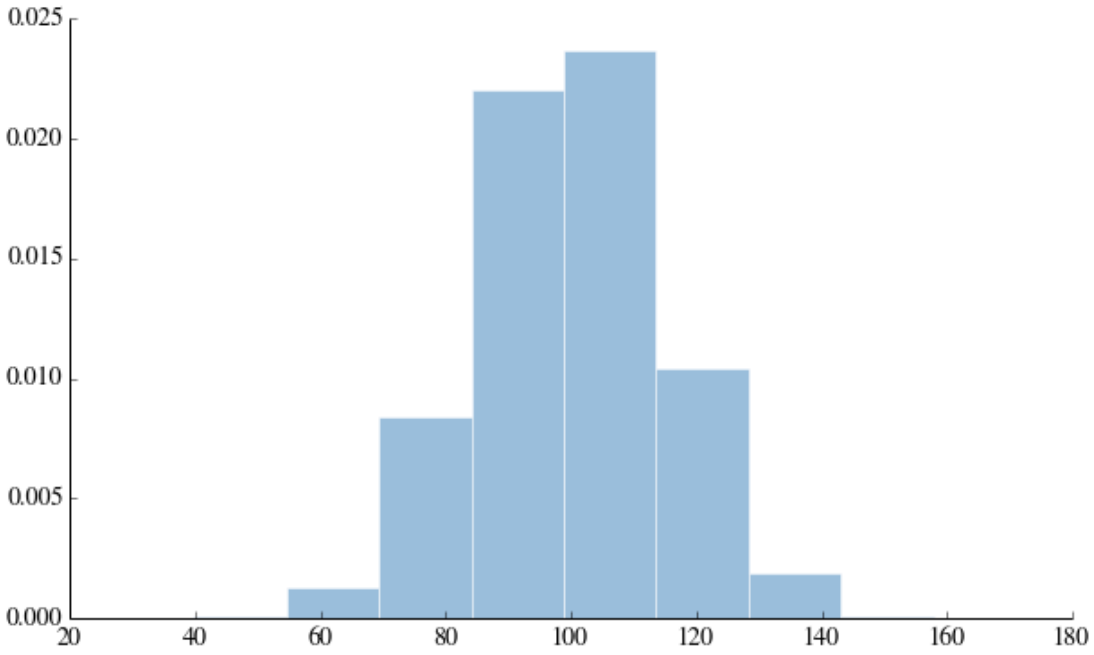


In [41]:
```
n = np.array([0,1,2,3,4,5])
fig, axes = plt.subplots(1, 4, figsize=(12,3))

axes[0].scatter(x, x +0.5*np.random.randn(len(x)))

axes[1].step(n, n**2, lw=2)

axes[2].bar(n, n**2, align="center", width=0.5, alpha=0.5)
```

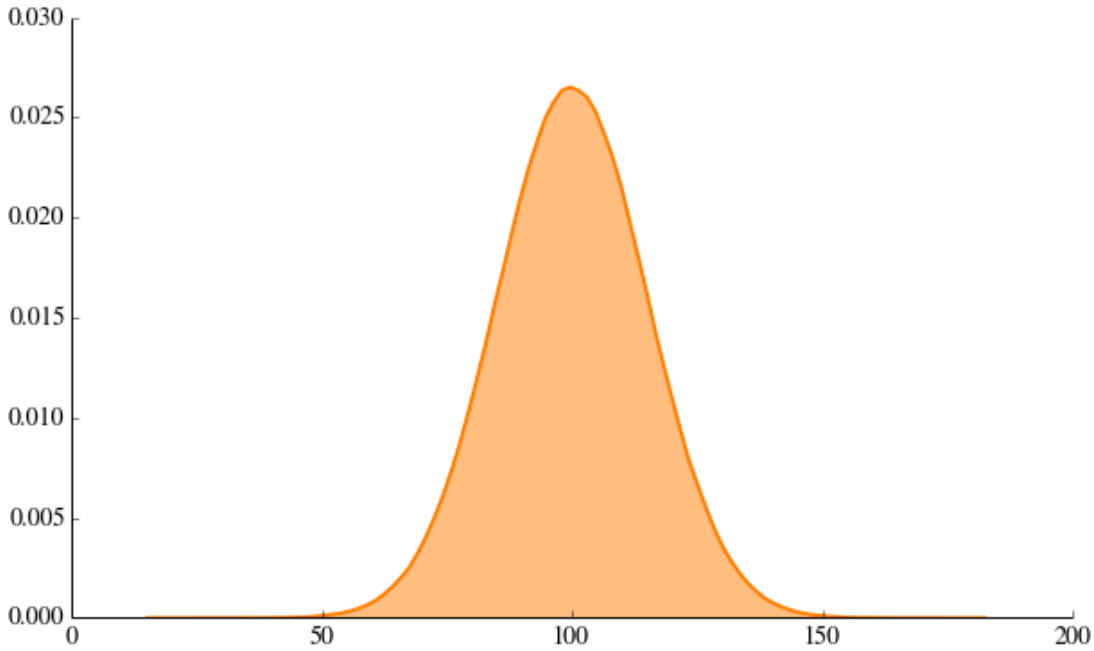
```
axes[3].fill_between(x, x**2, x**3, color="green", alpha=0.5);
```



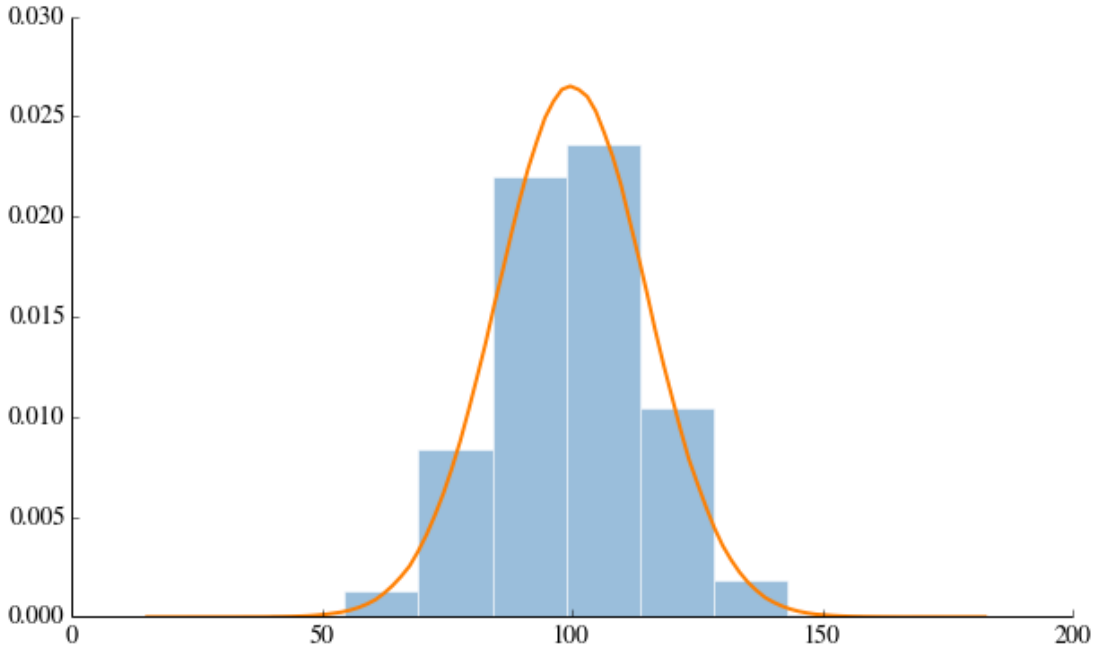
In [42]:
```
# Histograms

mu = 100
sigma = 15
x = mu + sigma * np.random.randn(3000000)

plt.hist(x, 10, normed=1, alpha=0.5, facecolor='#377EB8')
remove_border()
plt.show()
```



In [43]:
```
density = stats.kde.gaussian_kde(x)
xd = np.linspace(min(x)-10, max(x)+10, 100)
plt.plot(xd, density(xd), lw=2, color='#FF7F00') #line
plt.fill_between(xd, 0, density(xd), alpha=0.5, color='#FF7F00')
remove_border()
plt.show()
```

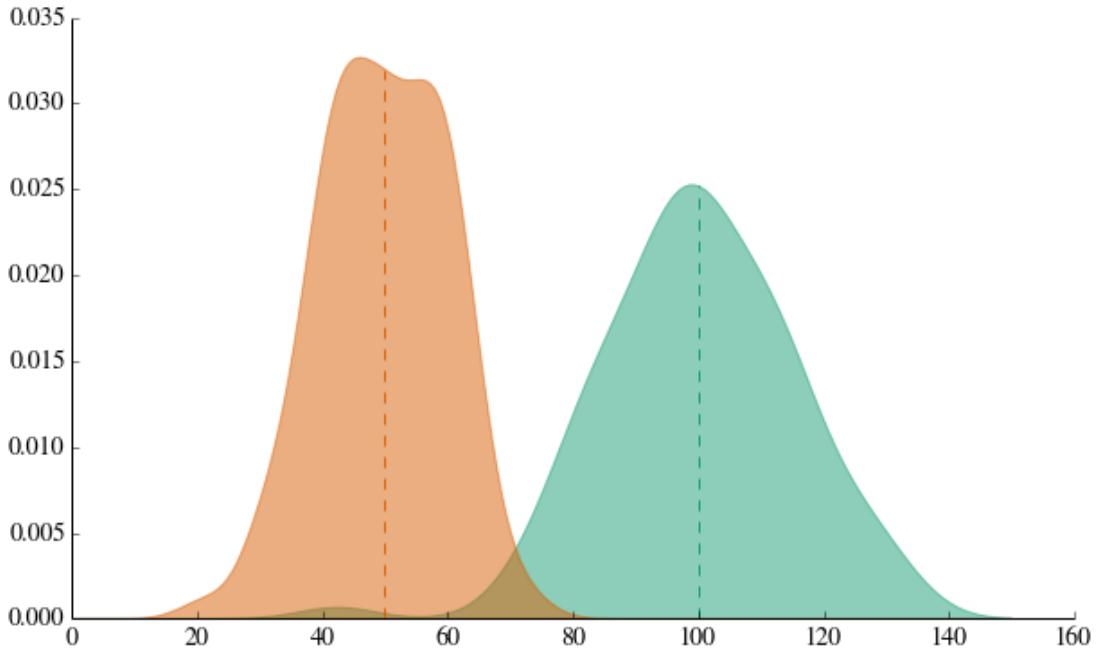


In [44]:
```
plt.hist(x, 10, normed=1, facecolor='#377EB8', alpha=0.5)
plt.plot(xd, density(xd), lw=2, color='#FF7F00')
remove_border()
plt.show()
```



In [82]:
```
from scipy import stats
x1 = 100 + 15 * np.random.randn(100)
x2 = 50 + 10 * np.random.randn(100)

density1 = stats.kde.gaussian_kde(x1)
density2 = stats.kde.gaussian_kde(x2)

x = np.linspace(0, 150, 200)
```

In [83]:
```
plt.fill_between(x, 0, density1(x), alpha=0.5, color='#1B9E77')
plt.fill_between(x, 0, density2(x), alpha=0.5, color='#D95F02')

plt.plot([100,100],[0,density1(100)], color ='#1B9E77',\
         linewidth=1, linestyle="--")
remove_border()
plt.plot([50,50],[0,density2(50)], color ='#D95F02',\
         linewidth=1, linestyle="--")
remove_border()
plt.show()
```



CHAPTER

SIX

NUMPY AND SCIPY: COMPUTING WITH VECTORS AND ARRAYS

Numpy contains core routines for doing fast vector, matrix, and linear algebra-type operations in Python. Scipy contains additional routines for optimisation, special functions, and so on. Both contain modules written in C and Fortran so that they're as fast as possible. Together, they give Python roughly the same capability that the Matlab program offers, but with an increase of speed that use to be higher than x300. (In fact, if you're an experienced Matlab user, there a guide to Numpy for Matlab users just for you.)

6.1 Working with vectors and matrices

Fundamental to both Numpy and Scipy is the ability to work with vectors and matrices. You can create vectors from lists using the **array** command. You can pass in a second argument to **array** that gives the numeric type. There are a number of types listed here that your matrix can be. Some of these are aliased to single character codes. The most common ones are 'd' (double precision floating point number), 'D' (double precision complex number), and 'i' (int32).

In [48]:
```
print np.array([1,2,3,4,5,6])
print np.array([1,2,3,4,5,6],'d')
print np.array([1,2,3,4,5,6],'D')
print np.array([1,2,3,4,5,6],'i')
print np.array([[0,1],[1,0]],'d')
```

```
[1 2 3 4 5 6]
[ 1.  2.  3.  4.  5.  6.]
[ 1.+0.j  2.+0.j  3.+0.j  4.+0.j  5.+0.j  6.+0.j]
[1 2 3 4 5 6]
[[ 0.  1.]
 [ 1.  0.]]
```

In [49]:
```
print np.zeros((3,3),'d')
print np.zeros(3,'d')
print np.zeros((1,3),'d')
print np.zeros((3,1),'d')
print np.identity(3,'d')
print np.identity(3,'d')*333
print np.ones((3,3),'d')
print np.ones((3,3),'d')*333
```

```
[[ 0.  0.  0.]
 [ 0.  0.  0.]
 [ 0.  0.  0.]]
[ 0.  0.  0.]
[[ 0.  0.  0.]]
[[ 0.]
 [ 0.]
 [ 0.]]
[[ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]]
[[ 333.    0.    0.]
 [   0.  333.    0.]
 [   0.    0.  333.]]
[[ 1.  1.  1.]
 [ 1.  1.  1.]
 [ 1.  1.  1.]]
[[ 333.  333.  333.]
 [ 333.  333.  333.]
 [ 333.  333.  333.]]
```

In [50]:
```
np.identity(2,'d') + np.array([[1,1],[1,2]])
```

Out [50]:
```
array([[ 2.,  1.],
       [ 1.,  3.]])
```

In [51]:
```
# elementwise multiplication
print np.identity(2)*np.ones((2,2))

print '='*20

# matrix multiplication
print np.dot(np.identity(2),np.ones((2,2)))
```

```
[[ 1.  0.]
 [ 0.  1.]]
====================
[[ 1.  1.]
 [ 1.  1.]]
```

In [52]:
```
# Transpose matrices
m = np.array([[1,2],[3,4]])
m.T
```

Out [52]:
```
array([[1, 3],
       [2, 4]])
```

In [53]:
```
# Diagonal of square matrix
np.diag([1,2,3,4,5])
```

Out [53]:
```
array([[1, 0, 0, 0, 0],
       [0, 2, 0, 0, 0],
       [0, 0, 3, 0, 0],
       [0, 0, 0, 4, 0],
       [0, 0, 0, 0, 5]])
```

In [54]:
```
# Solve systems of linear equations using the solve command

A = np.array([[1,1,1],[0,2,5],[2,5,-1]])
b = np.array([6,-4,27])
print np.linalg.solve(A,b)
print np.linalg.lstsq(A,b)
```

```
[ 5.  3. -2.]
(array([ 5.,  3., -2.]), array([], dtype=float64), 3, array([
6.08222134,  4.95180933,  0.69725745]))
```

In [55]:
```
A = np.array([[13,-4],[-4,7]],'d')
np.linalg.eigvals(A)
```

Out [55]:
```
array([ 15.,   5.])
```

6.2 Code Example 02: Least-squares fitting

Very often we deal with some data that we want to fit to some sort of expected behaviour. Say we have the following:

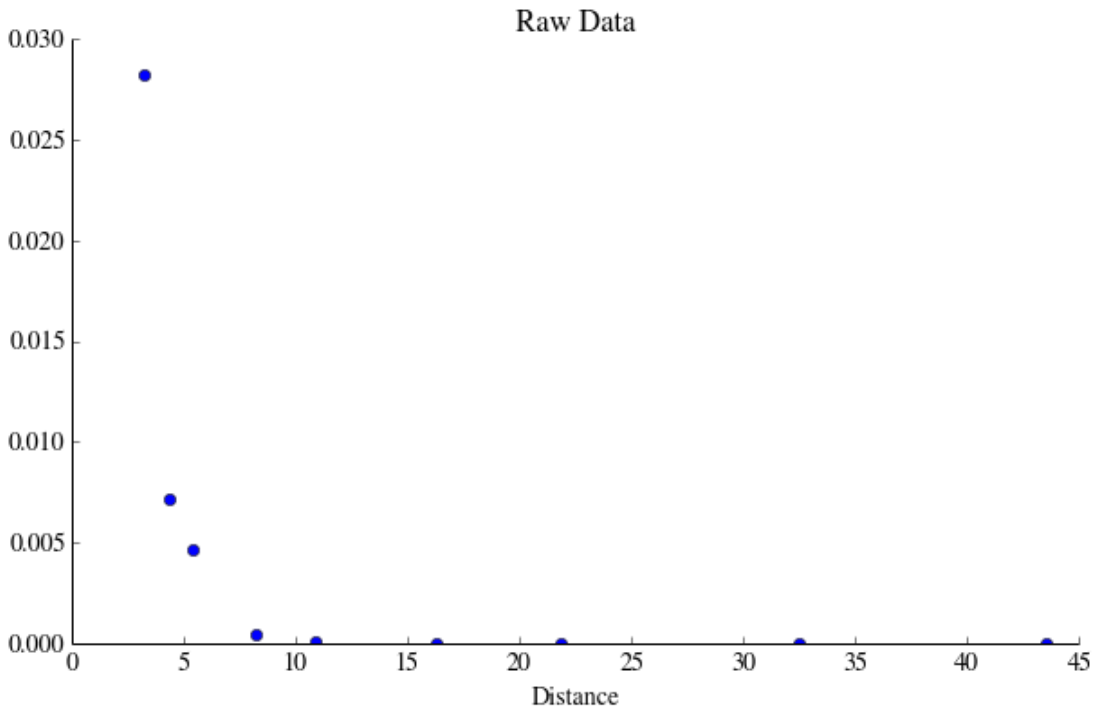
In [56]:
```
raw_data = """\
3.1905781584582433,0.028208609537968457
4.346895074946466,0.007160804747670053
5.374732334047101,0.0046962988461934805
8.201284796573875,0.0004614473299618756
10.899357601713055,0.00005038370219939726
16.295503211991434,4.377451812785309e-7
21.82012847965739,3.0799922117601088e-9
32.48394004282656,1.524776208284536e-13
43.53319057815846,5.5012073588707224e-18"""

type(raw_data)
```

Out [56]:
```
str
```

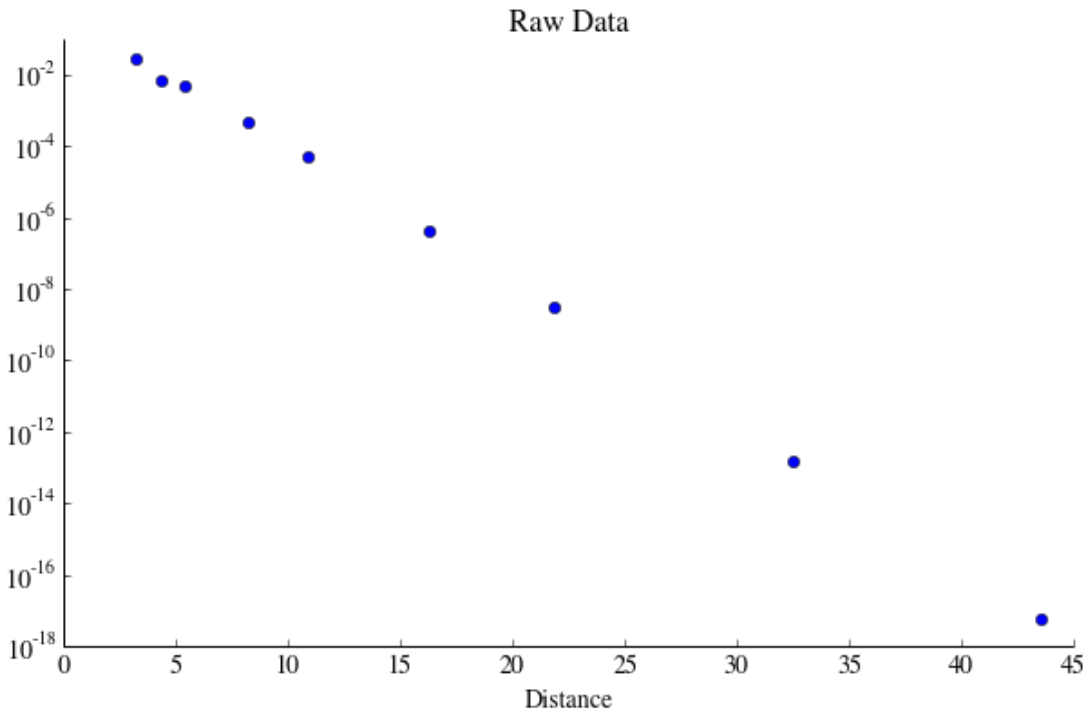
In [57]:
```
# DATA WRANGLING: Let us work a little with raw_data to make it mathematically accepta

data = []
for line in raw_data.splitlines():
    words = line.split(',')
    data.append(map(float,words))
data = np.array(data)
print data
```

```
[[  3.19057816e+00   2.82086095e-02]
 [  4.34689507e+00   7.16080475e-03]
 [  5.37473233e+00   4.69629885e-03]
 [  8.20128480e+00   4.61447330e-04]
 [  1.08993576e+01   5.03837022e-05]
 [  1.62955032e+01   4.37745181e-07]
 [  2.18201285e+01   3.07999221e-09]
 [  3.24839400e+01   1.52477621e-13]
 [  4.35331906e+01   5.50120736e-18]]
```

In [58]:
```
# DATA VISUALIZATION: How these data looks like...

plt.title("Raw Data")
plt.xlabel("Distance")
plt.plot(data[:,0],data[:,1],'bo')
remove_border()
```



In [60]:
```
plt.title("Raw Data")
plt.xlabel("Distance")
plt.semilogy(data[:,0],data[:,1],'bo')
remove_border()
```



For a pure exponential decay like this, we can fit the log of the data to a straight line. The above plot suggests this is a good approximation. Given a function

$$y = Ae^{-ax}$$

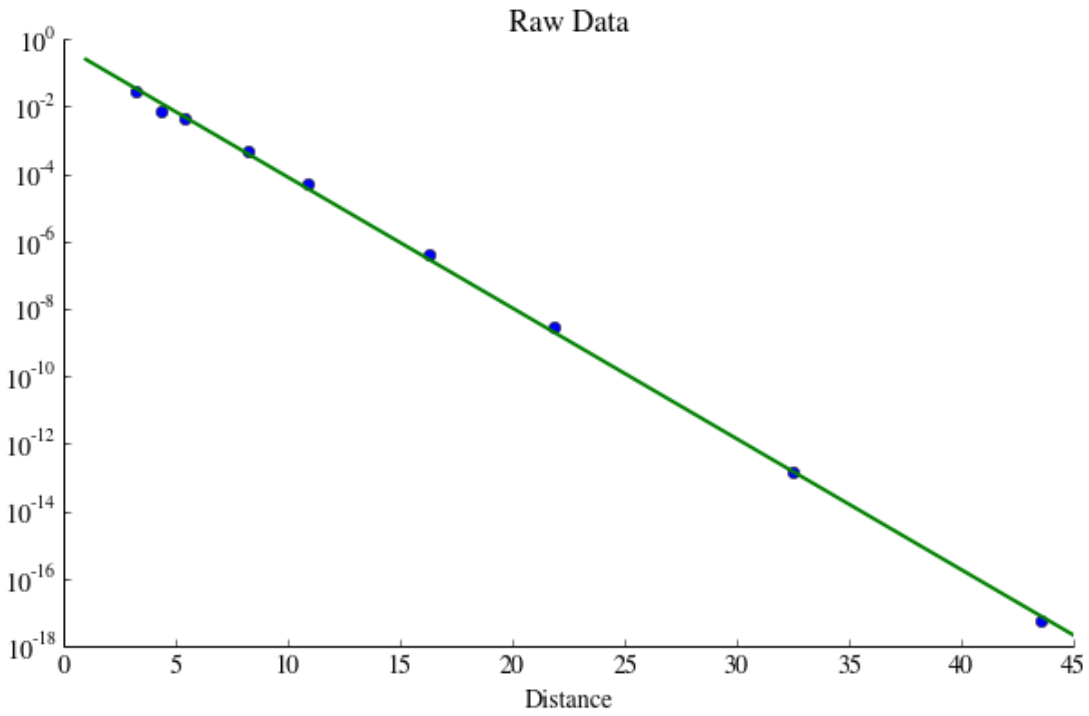
$$\log(y) = \log(A) - ax$$

Thus, if we fit the log of the data versus x, we should get a straight line with slope a, and an intercept that gives the constant A.

There's a numpy function called **polyfit** that will fit data to a polynomial form. We'll use this to fit to a straight line (a polynomial of order 1)

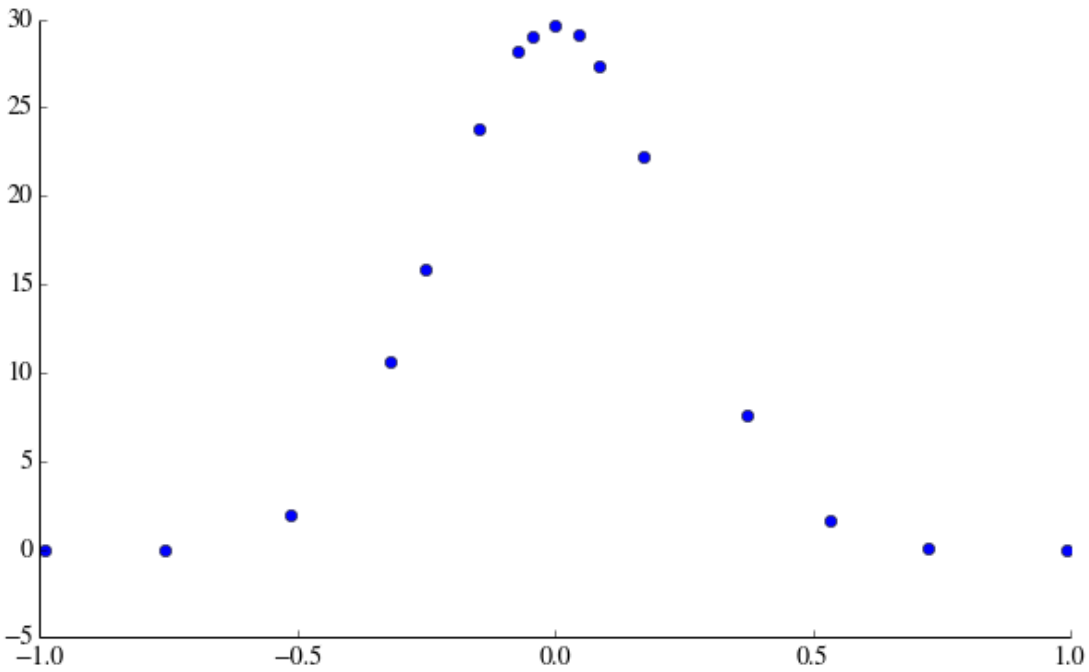
In [63]:
```
params = np.polyfit(data[:,0],np.log(data[:,1]),1)
a = params[0]
A = np.exp(params[1])
```

In [65]:
```
x = np.linspace(1,45)
plt.title("Raw Data")
plt.xlabel("Distance")
plt.semilogy(data[:,0],data[:,1],'bo')
remove_border()
plt.semilogy(x,A*np.exp(a*x),'g-')
remove_border()
```



Let us consider another example a little bit more complicate:

In [67]:
```
gauss_data = """\
-0.9902286902286903,1.4065274110372852e-19
-0.7566104566104566,2.2504438576596563e-18
-0.5117810117810118,1.9459459459459454
-0.31887271887271884,10.621621621621626
-0.250997150997151,15.891891891891893
-0.1463309463309464,23.756756756756754
-0.07267267267267263,28.135135135135133
-0.04426734426734419,29.02702702702703
-0.0015939015939017698,29.675675675675677
0.0468930468930468,29.10810810810811
0.0840994840994842,27.324324324324326
0.1700546700546699,22.216216216216214
0.370878570878571,7.540540540540545
0.5338338338338338,1.621621621621618
0.722014322014322,0.08108108108108068
0.9926849926849926,-0.08108108108108646"""

data = []
for line in gauss_data.splitlines():
    words = line.split(',')
    data.append(map(float,words))
data = np.array(data)

plt.plot(data[:,0],data[:,1],'bo')
remove_border()
```

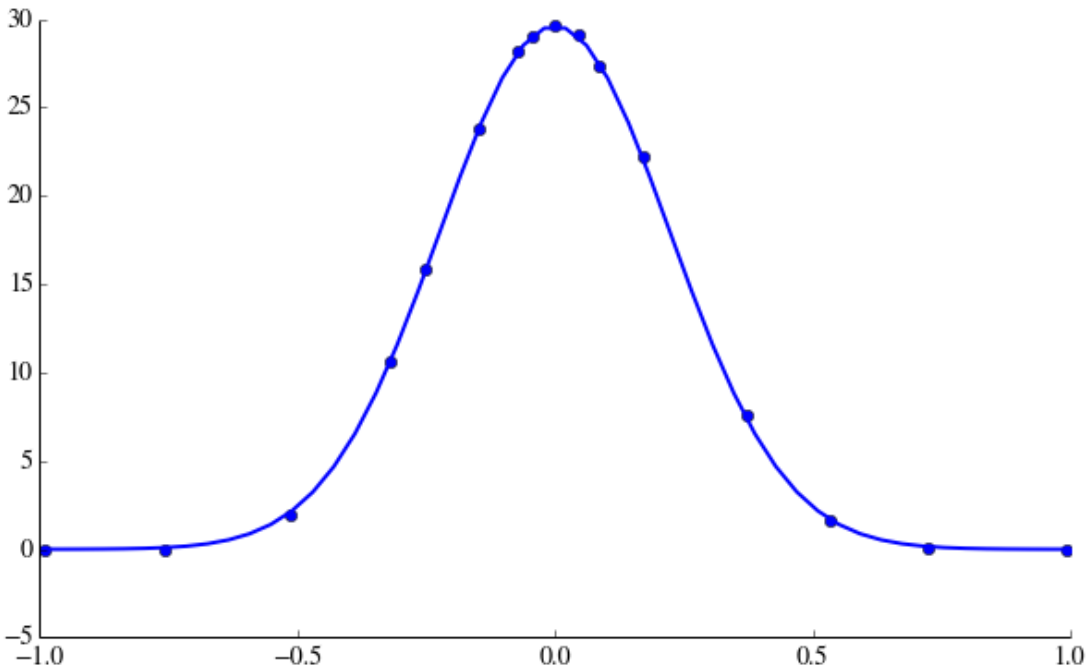


let's use the **curve_fit** function from Scipy, which can fit to arbitrary functions. You can learn more using help(curve_fit).

First define a general Gaussian function to fit to.

In [68]:
```
def gauss(x,A,a): return A*np.exp(a*x**2)
```

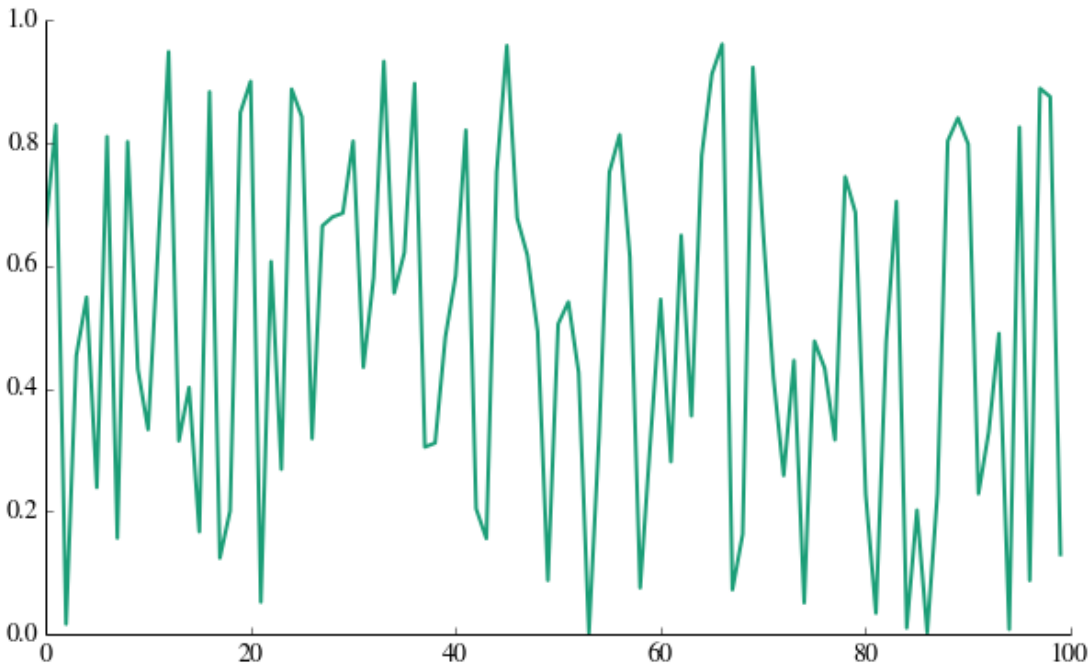
In [70]:
```
from scipy.optimize import curve_fit

params,conv = curve_fit(gauss,data[:,0],data[:,1])
x = np.linspace(-1,1)
plt.plot(data[:,0],data[:,1],'bo')
remove_border()
A,a = params
plt.plot(x,gauss(x,A,a),'b-')
remove_border()
```



6.3 Code Example 03: Monte-Carlo simulation and π computation

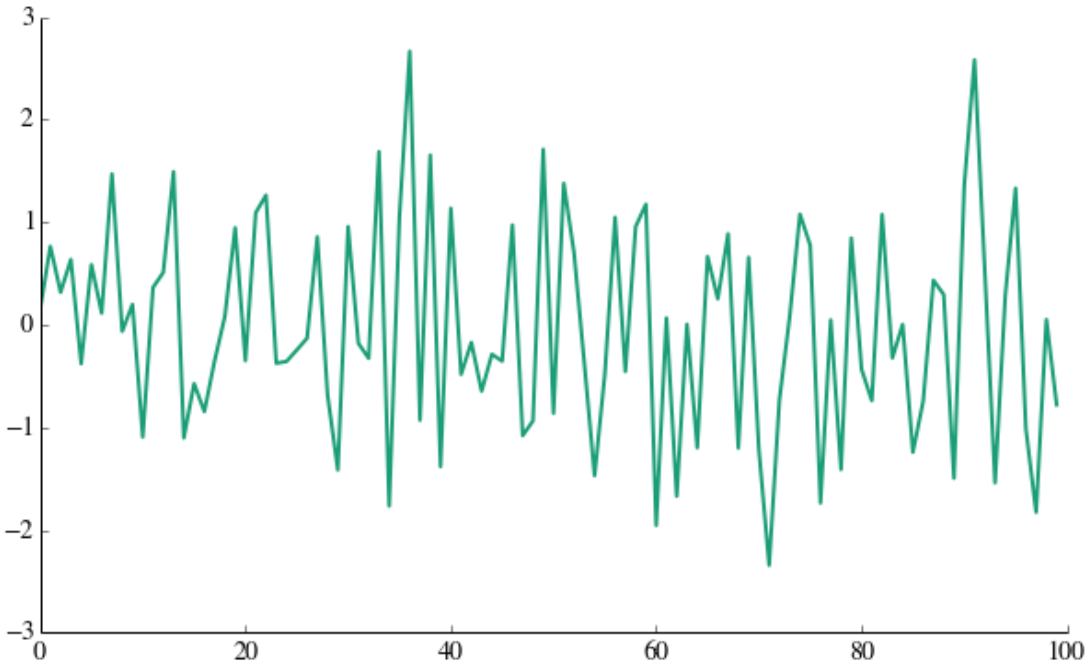
Python has good random number generators in the standard library. The **random()** function gives pseudorandom numbers uniformly distributed between 0 and 1:

In [72]:
```
from random import random
rands = []
for i in range(100):
    rands.append(random())
plt.plot(rands)
remove_border()
```



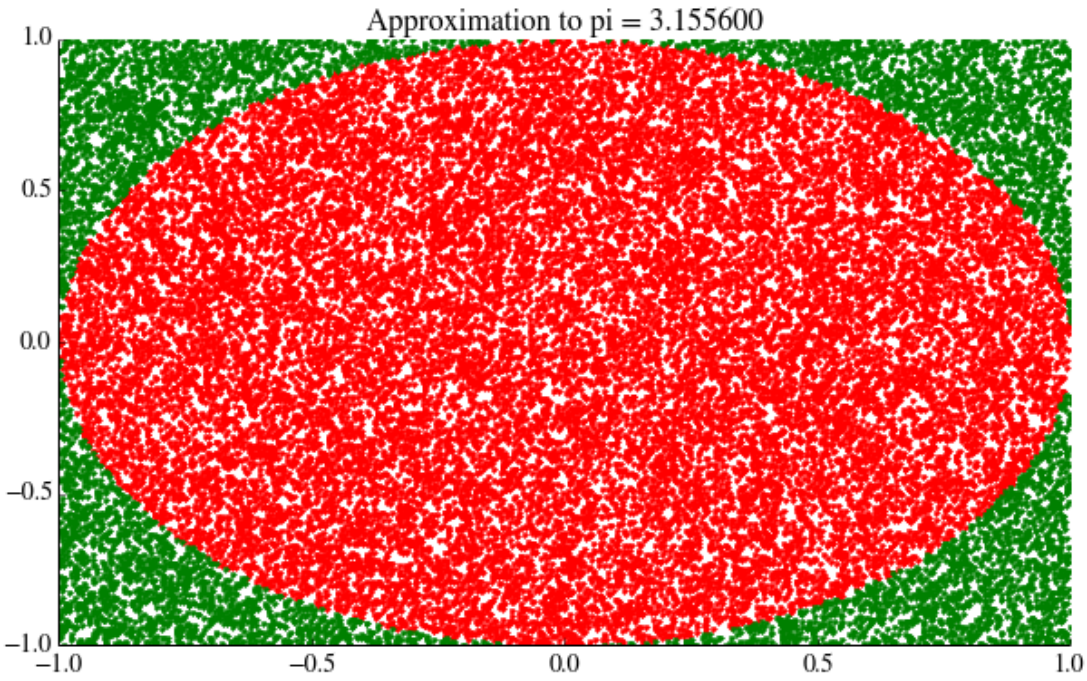
random() uses the Mersenne Twister algorithm, which is a highly regarded pseudorandom number generator. There are also functions to generate random integers, to randomly shuffle a list, and functions to pick random numbers from a particular distribution, like the normal distribution:

In [73]:
```
from random import gauss
grands = []
for i in range(100):
    grands.append(gauss(0,1))
plt.plot(grands)
remove_border()
```



Here you have an interesting program to compute π by taking random numbers as x and y coordinates, and counting how many of them were in the unit circle. For example:

In [81]:
```
npts = 30000
xs = 2*np.random.rand(npts)-1
ys = 2*np.random.rand(npts)-1
r = xs**2+ys**2
ninside = (r<1).sum()
# figsize(6,6) # make the figure square
plt.title("Approximation to pi = %f" % (4*ninside/float(npts)))
plt.plot(xs[r<1],ys[r<1],'r.')
remove_border()
plt.plot(xs[r>1],ys[r>1],'g.')
remove_border()
#plt.figsize(8,6) # change the figsize back to 4x3 for the rest of the notebook
```


Approximation to pi = 3.155600
1.0
0.5
0.0
−0.5
−1.0
−1.0
−0.5
0.0
0.5
1.0

CHAPTER

SEVEN

DATA SCRAPING: HTML, WEBAPI, CLIPBOARD, SQLITE...

Data Scraping is an automated process that retrieve and download pages, grab the content and store it in databases or text files. There is several methods in Python to do DataScraping: **urlparse** (to manipulate url-strings), **urllib** (to download data through different protocols like http, ftp...), **BeautifulSoup** (html/xml parser), **lxml** (library for processing XML and HTML), **twill**, **Selenium**... and of course, some of them are included in pandas.

The Pandas I/O api is a set of top level reader functions accessed like pd.read_csv() that generally return a pandas object.

- read_csv
- read_excel
- read_hdf
- read_sql
- read_json
- read_html
- read_stata
- read_clipboard
- read_pickle

The corresponding writer functions are object methods that are accessed like df.to_csv()

- to_csv
- to_excel
- to_hdf
- to_sql
- to_json
- to_html
- to_stata
- to_clipboard
- to_pickle

7.1 Scraping from Web-HTML

In [2]:
```
# Just in case you do not have intalled: run directly
# !pip install html5lib

url = 'http://www.fdic.gov/bank/individual/failed/banklist.html'
dfs = pd.read_html(url)
```

```
df = dfs[0]
print dfs
```

```
[<class 'pandas.core.frame.DataFrame'>
Int64Index: 516 entries, 0 to 515
Data columns (total 8 columns):
Bank Name                516  non-null values
City                     516  non-null values
ST                       516  non-null values
CERT                     516  non-null values
Acquiring Institution    516  non-null values
Closing Date             516  non-null values
Updated Date             516  non-null values
Loss Share Type          516  non-null values
dtypes: datetime64[ns](2), int64(1), object(5)]
```

In [3]:
```
df.head()
```

Out [3]:

```
                                              Bank Name           City
ST   CERT                      Acquiring Institution        Closing
Date        Updated Date Loss Share Type
0         Texas Community Bank, National Association  The Woodlands
TX  57431                   Spirit of Texas Bank, SSB 2013-12-13
00:00:00 2013-12-24 00:00:00
1                            Bank of Jackson County     Graceville
FL  14794             First Federal Bank of Florida 2013-10-30
00:00:00 2013-12-09 00:00:00            none
2  First National Bank also operating as The  Nat...       Edinburg
TX  14318                        PlainsCapital Bank 2013-09-13
00:00:00 2013-11-01 00:00:00         SFR/NSF
3                              The Community's Bank     Bridgeport
CT  57041                             No Acquirer 2013-09-13
00:00:00 2013-12-20 00:00:00            none
4                           Sunrise Bank of Arizona        Phoenix
AZ  34707  First Fidelity Bank, National Association 2013-08-23
00:00:00 2013-11-01 00:00:00            none
```

On the other hand, Beautiful Soup is a Python library for pulling data out of HTML and XML files. It works with your favourite parser to provide idiomatic ways of navigating, searching, and modifying the parse tree. It commonly saves programmers hours or days of work.

In [16]:
```
# Installing the package BeautifulSoup
! pip install BeautifulSoup
```

```
Requirement already satisfied (use --upgrade to upgrade):
BeautifulSoup4 in c:\anaconda\lib\site-packages
Cleaning up...
Downloading/unpacking pattern
  Cannot fetch index base URL https://pypi.python.org/simple/
  Could not find any downloads that satisfy the requirement pattern
Cleaning up...
No distributions at all found for pattern
```

```
Storing complete log in C:\Users\Suso\pip\pip.log
```

In [134]:
```
import requests
from bs4 import BeautifulSoup
```

In [135]:
```
url = 'http://www.imdb.com/search/\
    title?sort=num_votes,desc&start=1&\
    title_type=feature&year=1950,2012'
r = requests.get(url)
print r.url
```

```
http://www.imdb.com/search/title?sort=num_votes,desc&start=1&title_typ
e=feature&year=1950,2012
```

In [136]:
```
bs = BeautifulSoup(r.text)
for movie in bs.findAll('td', 'title'):
    title = movie.find('a').contents[0]
    genres = movie.find('span', 'genre').findAll('a')
    genres = [g.contents[0] for g in genres]
    runtime = movie.find('span', 'runtime').contents[0]
    rating = movie.find('span', 'value').contents[0]
    print title, genres, runtime, rating
```

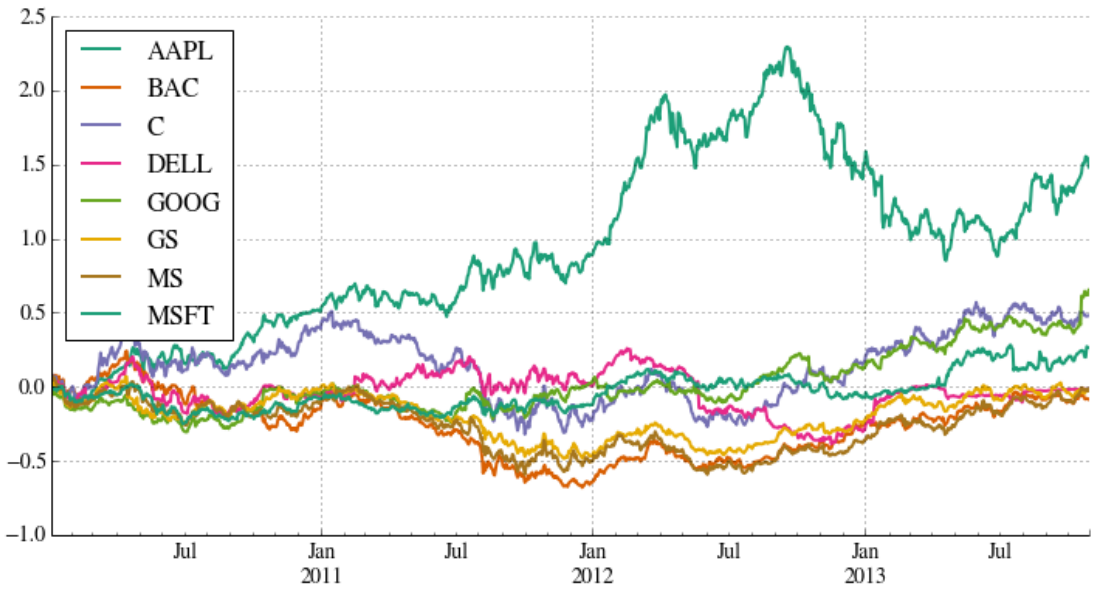
```
The Shawshank Redemption [u'Crime', u'Drama'] 142 mins. 9.3
The Dark Knight [u'Action', u'Crime', u'Drama', u'Thriller'] 152 mins.
9.0
Inception [u'Action', u'Adventure', u'Mystery', u'Sci-Fi',
u'Thriller'] 148 mins. 8.8
Pulp Fiction [u'Crime', u'Drama', u'Thriller'] 154 mins. 9.0
Fight Club [u'Drama'] 139 mins. 8.8
The Lord of the Rings: The Fellowship of the Ring [u'Action',
u'Adventure', u'Fantasy'] 178 mins. 8.8
The Matrix [u'Action', u'Adventure', u'Sci-Fi'] 136 mins. 8.7
The Lord of the Rings: The Return of the King [u'Action',
u'Adventure', u'Fantasy'] 201 mins. 8.9
The Godfather [u'Crime', u'Drama'] 175 mins. 9.2
Forrest Gump [u'Drama', u'Romance'] 142 mins. 8.7
The Lord of the Rings: The Two Towers [u'Action', u'Adventure',
u'Fantasy'] 179 mins. 8.7
The Dark Knight Rises [u'Action', u'Crime', u'Thriller'] 165 mins. 8.6
Se7en [u'Crime', u'Mystery', u'Thriller'] 127 mins. 8.7
Avatar [u'Action', u'Adventure', u'Fantasy', u'Sci-Fi'] 162 mins. 7.9
Batman Begins [u'Action', u'Adventure', u'Crime', u'Drama'] 140 mins.
8.3
Gladiator [u'Action', u'Adventure', u'Drama'] 155 mins. 8.5
Star Wars [u'Action', u'Adventure', u'Fantasy', u'Sci-Fi'] 121 mins.
8.7
The Avengers [u'Action', u'Fantasy'] 143 mins. 8.2
Memento [u'Mystery', u'Thriller'] 113 mins. 8.5
American Beauty [u'Drama'] 122 mins. 8.5
Saving Private Ryan [u'Action', u'Drama', u'War'] 169 mins. 8.6
Schindler's List [u'Biography', u'Drama', u'History', u'War'] 195
```

```
mins. 8.9
The Departed
```

Another typical example is how to retrive **Time Series** of market prices from web pages. Market prices can be retrived for free from **Yahoo Finance**, **Google Finance** or **Quandl.com**. Here you have an example.

In [4]:
```
import pandas.io.data as web

# We can retrive data from internet one-by-one:
# aapl = web.get_data_yahoo('AAPL', '2010-01-01')['Adj Close']
# msft = web.get_data_yahoo('MSFT', '2010-01-01')['Adj Close']

# or we can retrive the whole list
names = ['AAPL', 'GOOG', 'MSFT', 'DELL', 'GS', 'MS', 'BAC', 'C']

def get_prices(stock, start, end):
    return web.get_data_yahoo(stock, start, end)['Adj Close']

px = pd.DataFrame({n: get_prices(n, '2010-01-01', now)
                   for n in names}).dropna()
```

In [5]:
```
px = px.asfreq('B').fillna(method='pad')
rets = px.pct_change()
cum_rets = ((1+rets).cumprod()-1).dropna()
cum_rets.plot(figsize=(12,6))
remove_border()
```



In [6]:
```
cum_rets.head()
```

Out [6]:

	AAPL	BAC	C	DELL	GOOG	GS	MS	MSFT
2010-01-05	0.001740	0.032362	0.038348	0.022033	-0.004404	0.017685	0.036589	0.000359
2010-01-06	-0.014208	0.044660	0.070501	0.002132	-0.029501	0.006867	0.050017	-0.005739

```
2010-01-07 -0.016044  0.078964  0.073451  0.012082 -0.052094  0.026557
0.065123 -0.016141
2010-01-08 -0.009472  0.069256  0.055752  0.021322 -0.039458  0.007110
0.043639 -0.009326
2010-01-11 -0.018219  0.078964  0.067552  0.021322 -0.040909 -0.008751
0.036589 -0.021879
```

7.2 Scrapping using a Web API: The Twitter Example

Many web APIs will return a JSON string that must be loaded into a Python object. Here we will use the Twitter API as example, following the example in the Chapter 1 of Mining the Social Web (2nd Edition).

Twitter might be described as a real-time, highly social microblogging service that allows users to post short status updates, called tweets, that appear on timelines. Twitter has a simple RESTful API that is intuitive and easy to use. Even so, there are great libraries available to further mitigate the work involved in making API requests. A particularly Python package that wraps the Twitter API and mimics the public API semantics almost one-to-one is twitter. Like most other Python packages, you can install it with pip by typing pip install twitter in a terminal.

In [12]:
```
# Installing the Twitter package using pip
! pip install twitter
```

```
Downloading/unpacking twitter
  Downloading twitter-1.10.2.tar.gz
  Running setup.py egg_info for package twitter

Installing collected packages: twitter
  Running setup.py install for twitter

    Installing twitter-log-script.py script to C:\Anaconda\Scripts
    Installing twitter-log.exe script to C:\Anaconda\Scripts
    Installing twitter-log.exe.manifest script to C:\Anaconda\Scripts
    Installing twitter-script.py script to C:\Anaconda\Scripts
    Installing twitter.exe script to C:\Anaconda\Scripts
    Installing twitter.exe.manifest script to C:\Anaconda\Scripts
    Installing twitterbot-script.py script to C:\Anaconda\Scripts
    Installing twitterbot.exe script to C:\Anaconda\Scripts
    Installing twitterbot.exe.manifest script to C:\Anaconda\Scripts
    Installing twitter-follow-script.py script to C:\Anaconda\Scripts
    Installing twitter-follow.exe script to C:\Anaconda\Scripts
    Installing twitter-follow.exe.manifest script to
C:\Anaconda\Scripts
    Installing twitter-stream-example-script.py script to
C:\Anaconda\Scripts
    Installing twitter-stream-example.exe script to
C:\Anaconda\Scripts
    Installing twitter-stream-example.exe.manifest script to
C:\Anaconda\Scripts
    Installing twitter-archiver-script.py script to
C:\Anaconda\Scripts
    Installing twitter-archiver.exe script to C:\Anaconda\Scripts
    Installing twitter-archiver.exe.manifest script to
C:\Anaconda\Scripts
Successfully installed twitter
```

```
Cleaning up...
```

In [2]:
```
import twitter

'''
XXX: Go to http://dev.twitter.com/apps/new
    to create an app and get values
    for these credentials, which you'll need
    to provide in place of these
XXX string values that are defined as placeholders.
    See https://dev.twitter.com/docs/auth/oauth
    for more information
    on Twitter's OAuth implementation.
'''

CONSUMER_KEY = 'XXX'
CONSUMER_SECRET = 'XXX'
OAUTH_TOKEN = 'XXX'
OAUTH_TOKEN_SECRET = 'XXX'

auth = twitter.oauth.OAuth(OAUTH_TOKEN, OAUTH_TOKEN_SECRET,
                           CONSUMER_KEY, CONSUMER_SECRET)

twitter_api = twitter.Twitter(auth=auth)

# Nothing to see by displaying twitter_api
# except that it's now a defined variable

print twitter_api
```

```
<twitter.api.Twitter object at 0x02DC7C10>
```

Example 1: Exploring Trends

With an authorised API connection in place, you can now issue a request. Here we show how to ask Twitter for the topics that are currently trending worldwide, but keep in mind that the API can easily be parameterized to constrain the topics to more specific locales if you feel inclined to try out some of the possibilities. The device for constraining queries is via Yahoo! GeoPlanet's Where On Earth (WOE) ID system, which is an API unto itself that aims to provide a way to map a unique identifier to any named place on Earth (or theoretically, even in a virtual world). Here we collects a set of trends for both the entire world and just the United States.

In [3]:
```
# The Yahoo! Where On Earth ID for the entire world is 1.
# See https://dev.twitter.com/docs/api/1.1/get/trends/place and
# http://developer.yahoo.com/geo/geoplanet/

WORLD_WOE_ID = 1
US_WOE_ID = 23424977

# Prefix ID with the underscore for query string parametrization.
# Without the underscore, the twitter package appends the ID value
# to the URL itself as a special case keyword argument.

world_trends = twitter_api.trends.place(_id=WORLD_WOE_ID)
us_trends = twitter_api.trends.place(_id=US_WOE_ID)

print world_trends
print
print us_trends
```

[{u'created_at': u'2013-12-23T09:37:00Z', u'trends': [{u'url': u'http:
//twitter.com/search?q=%22B%C3%BClent+Y%C4%B1ld%C4%B1r%C4%B1m%22',
u'query': u'%22B%C3%BClent+Y%C4%B1ld%C4%B1r%C4%B1m%22', u'name':
u'B\xfclent Y\u0131ld\u0131r\u0131m', u'promoted_content': None,
u'events': None}, {u'url':
u'http://twitter.com/search?q=%23Erdo%C4%9FanaG%C3%BCvenimizTam',
u'query': u'%23Erdo%C4%9FanaG%C3%BCvenimizTam', u'name':
u'#Erdo\u011fanaG\xfcvenimizTam', u'promoted_content': None,
u'events': None}, {u'url': u'http://twitter.com/search?q=%23EdepYaHu',
u'query': u'%23EdepYaHu', u'name': u'#EdepYaHu', u'promoted_content':
None, u'events': None}, {u'url': u'http://twitter.com/search?q=%23Ayti
l%C4%B0%C3%A7inD%C3%B6n%C3%BCmNoktas%C4%B1', u'query':
u'%23Aytil%C4%B0%C3%A7inD%C3%B6n%C3%BCmNoktas%C4%B1', u'name':
u'#Aytil\u0130\xe7inD\xf6n\xfcmNoktas\u0131', u'promoted_content':
None, u'events': None}, {u'url':
u'http://twitter.com/search?q=%23fandommemories2013', u'query':
u'%23fandommemories2013', u'name': u'#fandommemories2013',
u'promoted_content': None, u'events': None}, {u'url':
u'http://twitter.com/search?q=%23DEMIin2013', u'query':
u'%23DEMIin2013', u'name': u'#DEMIin2013', u'promoted_content': None,
u'events': None}, {u'url': u'http://twitter.com/search?q=%22Sar%C4%B1k
am%C4%B1%C5%9FDondu+Y%C3%BCreklerimizYand%C4%B1%22', u'query':
u'%22Sar%C4%B1kam%C4%B1%C5%9FDondu+Y%C3%BCreklerimizYand%C4%B1%22',
u'name': u'Sar\u0131kam\u0131\u015fDondu Y\xfcreklerimizYand\u0131',
u'promoted_content': None, u'events': None}, {u'url': u'http://twitter
.com/search?q=%22OperasyonK%C4%B1l%C4%B1f+HedefT%C3%BCrkiye%22',
u'query': u'%22OperasyonK%C4%B1l%C4%B1f+HedefT%C3%BCrkiye%22',
u'name': u'OperasyonK\u0131l\u0131f HedefT\xfcrkiye',
u'promoted_content': None, u'events': None}, {u'url': u'http://twitter
.com/search?q=%22AKPninA%C3%A7%C4%B1l%C4%B1m%C4%B1+Ayakkab%C4%B1Kutusu
Partisi%22', u'query':
u'%22AKPninA%C3%A7%C4%B1l%C4%B1m%C4%B1+Ayakkab%C4%B1KutusuPartisi%22',
u'name': u'AKPninA\xe7\u0131l\u0131m\u0131
Ayakkab\u0131KutusuPartisi', u'promoted_content': None, u'events':
None}, {u'url':
u'http://twitter.com/search?q=%22%C5%9Eampiyon+Be%C5%9Fikta%C5%9F%22',
u'query': u'%22%C5%9Eampiyon+Be%C5%9Fikta%C5%9F%22', u'name':
u'\u015eampiyon Be\u015fikta\u015f', u'promoted_content': None,
u'events': None}], u'as_of': u'2013-12-23T09:41:25Z', u'locations':
[{u'woeid': 1, u'name': u'Worldwide'}]}]

[{u'created_at': u'2013-12-23T09:22:30Z', u'trends': [{u'url':
u'http://twitter.com/search?q=%23fandommemories2013', u'query':
u'%23fandommemories2013', u'name': u'#fandommemories2013',
u'promoted_content': None, u'events': None}, {u'url':
u'http://twitter.com/search?q=%23DEMIin2013', u'query':
u'%23DEMIin2013', u'name': u'#DEMIin2013', u'promoted_content': None,
u'events': None}, {u'url': u'http://twitter.com/search?q=%23DALvsWAS',
u'query': u'%23DALvsWAS', u'name': u'#DALvsWAS', u'promoted_content':
None, u'events': None}, {u'url':
u'http://twitter.com/search?q=%23AskZach', u'query': u'%23AskZach',
u'name': u'#AskZach', u'promoted_content': None, u'events': None},
{u'url': u'http://twitter.com/search?q=%23PeopleWhoMadeMyYearGood',

```
u'query': u'%23PeopleWhoMadeMyYearGood', u'name':
u'#PeopleWhoMadeMyYearGood', u'promoted_content': None, u'events':
None}, {u'url': u'http://twitter.com/search?q=%22Micah+Hyde%22',
u'query': u'%22Micah+Hyde%22', u'name': u'Micah Hyde',
u'promoted_content': None, u'events': None}, {u'url':
u'http://twitter.com/search?q=%22Christmas+is+in+3%22', u'query':
u'%22Christmas+is+in+3%22', u'name': u'Christmas is in 3',
u'promoted_content': None, u'events': None}, {u'url':
u'http://twitter.com/search?q=%228+Mile%22', u'query':
u'%228+Mile%22', u'name': u'8 Mile', u'promoted_content': None,
u'events': None}, {u'url':
u'http://twitter.com/search?q=%22Von+Miller%22', u'query':
u'%22Von+Miller%22', u'name': u'Von Miller', u'promoted_content':
None, u'events': None}, {u'url':
u'http://twitter.com/search?q=%22Happy+Holidays%22', u'query':
u'%22Happy+Holidays%22', u'name': u'Happy Holidays',
u'promoted_content': None, u'events': None}], u'as_of':
u'2013-12-23T09:41:26Z', u'locations': [{u'woeid': 23424977, u'name':
u'United States'}]}]
```

Example 2: Displaying API responses as pretty-printed JSON

Although it hasn't explicitly been stated yet, the semi-readable output from Example 1, "Exploring trends" is printed out as native Python data structures. While an IPython interpreter will "pretty print" the output for you automatically, IPython Notebook and a standard Python interpreter will not. If you find yourself in these circumstances, you may find it handy to use the built-in json package to force a nicer display, as illustrated in Example 2, "Displaying API responses as pretty-printed JSON".

In [4]:
```
import json

print json.dumps(world_trends, indent=1)
print
print json.dumps(us_trends, indent=1)
```

```
[
 {
  "created_at": "2013-12-23T09:37:00Z",
  "trends": [
   {
    "url": "http://twitter.com/search?q=%22B%C3%BClent+Y%C4%B1ld%C4%B1
r%C4%B1m%22",
    "query": "%22B%C3%BClent+Y%C4%B1ld%C4%B1r%C4%B1m%22",
    "name": "B\u00fclent Y\u0131ld\u0131r\u0131m",
    "promoted_content": null,
    "events": null
   },
   {
    "url":
"http://twitter.com/search?q=%23Erdo%C4%9FanaG%C3%BCvenimizTam",
    "query": "%23Erdo%C4%9FanaG%C3%BCvenimizTam",
    "name": "#Erdo\u011fanaG\u00fcvenimizTam",
    "promoted_content": null,
    "events": null
   },
```

```
    {
     "url": "http://twitter.com/search?q=%23EdepYaHu",
     "query": "%23EdepYaHu",
     "name": "#EdepYaHu",
     "promoted_content": null,
     "events": null
    },
    {
     "url": "http://twitter.com/search?q=%23Aytil%C4%B0%C3%A7inD%C3%B6n
%C3%BCmNoktas%C4%B1",
     "query": "%23Aytil%C4%B0%C3%A7inD%C3%B6n%C3%BCmNoktas%C4%B1",
     "name": "#Aytil\u0130\u00e7inD\u00f6n\u00fcmNoktas\u0131",
     "promoted_content": null,
     "events": null
    },
    {
     "url": "http://twitter.com/search?q=%23fandommemories2013",
     "query": "%23fandommemories2013",
     "name": "#fandommemories2013",
     "promoted_content": null,
     "events": null
    },
    {
     "url": "http://twitter.com/search?q=%23DEMIin2013",
     "query": "%23DEMIin2013",
     "name": "#DEMIin2013",
     "promoted_content": null,
     "events": null
    },
    {
     "url": "http://twitter.com/search?q=%22Sar%C4%B1kam%C4%B1%C5%9FDon
du+Y%C3%BCreklerimizYand%C4%B1%22",
     "query":
"%22Sar%C4%B1kam%C4%B1%C5%9FDondu+Y%C3%BCreklerimizYand%C4%B1%22",
     "name": "Sar\u0131kam\u0131\u015fDondu
Y\u00fcreklerimizYand\u0131",
     "promoted_content": null,
     "events": null
    },
    {
     "url": "http://twitter.com/search?q=%22OperasyonK%C4%B1l%C4%B1f+He
defT%C3%BCrkiye%22",
     "query": "%22OperasyonK%C4%B1l%C4%B1f+HedefT%C3%BCrkiye%22",
     "name": "OperasyonK\u0131l\u0131f HedefT\u00fcrkiye",
     "promoted_content": null,
     "events": null
    },
    {
     "url": "http://twitter.com/search?q=%22AKPninA%C3%A7%C4%B1l%C4%B1m
%C4%B1+Ayakkab%C4%B1KutusuPartisi%22",
     "query":
"%22AKPninA%C3%A7%C4%B1l%C4%B1m%C4%B1+Ayakkab%C4%B1KutusuPartisi%22",
     "name": "AKPninA\u00e7\u0131l\u0131m\u0131
Ayakkab\u0131KutusuPartisi",
```

```
    "promoted_content": null,
    "events": null
   },
   {
    "url":
"http://twitter.com/search?q=%22%C5%9Eampiyon+Be%C5%9Fikta%C5%9F%22",
    "query": "%22%C5%9Eampiyon+Be%C5%9Fikta%C5%9F%22",
    "name": "\u015eampiyon Be\u015fikta\u015f",
    "promoted_content": null,
    "events": null
   }
  ],
  "as_of": "2013-12-23T09:41:25Z",
  "locations": [
   {
    "woeid": 1,
    "name": "Worldwide"
   }
  ]
 }
]

[
 {
  "created_at": "2013-12-23T09:22:30Z",
  "trends": [
   {
    "url": "http://twitter.com/search?q=%23fandommemories2013",
    "query": "%23fandommemories2013",
    "name": "#fandommemories2013",
    "promoted_content": null,
    "events": null
   },
   {
    "url": "http://twitter.com/search?q=%23DEMIin2013",
    "query": "%23DEMIin2013",
    "name": "#DEMIin2013",
    "promoted_content": null,
    "events": null
   },
   {
    "url": "http://twitter.com/search?q=%23DALvsWAS",
    "query": "%23DALvsWAS",
    "name": "#DALvsWAS",
    "promoted_content": null,
    "events": null
   },
   {
    "url": "http://twitter.com/search?q=%23AskZach",
    "query": "%23AskZach",
    "name": "#AskZach",
    "promoted_content": null,
    "events": null
   },
```

```
    {
     "url": "http://twitter.com/search?q=%23PeopleWhoMadeMyYearGood",
     "query": "%23PeopleWhoMadeMyYearGood",
     "name": "#PeopleWhoMadeMyYearGood",
     "promoted_content": null,
     "events": null
    },
    {
     "url": "http://twitter.com/search?q=%22Micah+Hyde%22",
     "query": "%22Micah+Hyde%22",
     "name": "Micah Hyde",
     "promoted_content": null,
     "events": null
    },
    {
     "url": "http://twitter.com/search?q=%22Christmas+is+in+3%22",
     "query": "%22Christmas+is+in+3%22",
     "name": "Christmas is in 3",
     "promoted_content": null,
     "events": null
    },
    {
     "url": "http://twitter.com/search?q=%228+Mile%22",
     "query": "%228+Mile%22",
     "name": "8 Mile",
     "promoted_content": null,
     "events": null
    },
    {
     "url": "http://twitter.com/search?q=%22Von+Miller%22",
     "query": "%22Von+Miller%22",
     "name": "Von Miller",
     "promoted_content": null,
     "events": null
    },
    {
     "url": "http://twitter.com/search?q=%22Happy+Holidays%22",
     "query": "%22Happy+Holidays%22",
     "name": "Happy Holidays",
     "promoted_content": null,
     "events": null
    }
   ],
   "as_of": "2013-12-23T09:41:26Z",
   "locations": [
    {
     "woeid": 23424977,
     "name": "United States"
    }
   ]
  }
 ]
```


Notice that the sample result contains a URL for a trend represented as a search query that corresponds to the hashtag #MentionSomeoneImportantForYou, where %22 is the URL encoding for the hashtag symbol. We'll use this rather

benign hashtag throughout the remainder of the chapter as a unifying theme for examples that follow. Although a sample data file containing tweets for this hashtag is available with the book's source code, you'll have much more fun exploring a topic that's trending at the time you read this as opposed to following along with a canned topic that is no longer trending.

Example 3: Computing the intersection of two sets of trends

Although it's easy enough to skim the two sets of trends and look for commonality, let's use Python's set data structure to automatically compute this for us, because that's exactly the kind of thing that sets lend themselves to doing. In this instance, a set refers to the mathematical notion of a data structure that stores an unordered collection of unique items and can be computed upon with other sets of items and setwise operations. For example, a setwise intersection computes common items between sets, a setwise union combines all of the items from sets, and the setwise difference among sets acts sort of like a subtraction operation in which items from one set are removed from another.

This example demonstrates how to use a Python list comprehension to parse out the names of the trending topics from the results that were previously queried, cast those lists to sets, and compute the setwise intersection to reveal the common items between them. Keep in mind that there may or may not be significant overlap between any given sets of trends, all depending on what's actually happening when you query for the trends. In other words, the results of your analysis will be entirely dependent upon your query and the data that is returned from it.

In [5]:
```
world_trends_set = set([trend['name']
                        for trend in world_trends[0]['trends']])

us_trends_set = set([trend['name']
                     for trend in us_trends[0]['trends']])

common_trends = world_trends_set.intersection(us_trends_set)

print common_trends
```

```
set([u'#DEMIin2013', u'#fandommemories2013'])
```

Example 4: Searching for Tweets

In [9]:
```
# XXX: Set this variable to a trending topic,
# or anything else for that matter. The example query below
# was a trending topic when this content was being developed
# and is used throughout the remainder of this chapter.

q = '#E.On'

count = 100

# See https://dev.twitter.com/docs/api/1.1/get/search/tweets

search_results = twitter_api.search.tweets(q=q, count=count)

statuses = search_results['statuses']


# Iterate through 5 more batches of results by following the cursor

for _ in range(5):
    print "Length of statuses", len(statuses)
    try:
        next_results = search_results['search_metadata']['next_results']
    except KeyError, e:
        # No more results when next_results doesn't exist
```

```
        break

    # Create a dictionary from next_results,
    # which has the following form:
    # ?max_id=313519052523986943&q=NCAA&include_entities=1

    kwargs = dict([ kv.split('=') for kv in next_results[1:].split("&") ])

    search_results = twitter_api.search.tweets(**kwargs)
    statuses += search_results['statuses']

# Show one sample search result by slicing the list...
print json.dumps(statuses[0], indent=1)
```

```
Length of statuses 99
{
 "contributors": null,
 "truncated": false,
 "text": "RT @davethered55: To ALL E'ON customers who, from January
17th, will have to pay \u00a3165 a year standing charge REFUSE. I'm
going to and I'm p\u2026",
 "in_reply_to_status_id": null,
 "id": 415074098109509632,
 "favorite_count": 0,
 "source": "<a href=\"http://twitter.com/download/iphone\"
rel=\"nofollow\">Twitter for iPhone</a>",
 "retweeted": false,
 "coordinates": null,
 "entities": {
  "symbols": [],
  "user_mentions": [
   {
    "id": 873346964,
    "indices": [
     3,
     16
    ],
    "id_str": "873346964",
    "screen_name": "davethered55",
    "name": "davethered"
   }
  ],
  "hashtags": [],
  "urls": []
 },
 "in_reply_to_screen_name": null,
 "in_reply_to_user_id": null,
 "retweet_count": 1,
 "id_str": "415074098109509632",
 "favorited": false,
 "retweeted_status": {
  "contributors": null,
  "truncated": false,
  "text": "To ALL E'ON customers who, from January 17th, will have to
pay \u00a3165 a year standing charge REFUSE. I'm going to and I'm
padlocking my meters",
```

```
  "in_reply_to_status_id": null,
  "id": 415068819653402624,
  "favorite_count": 0,
  "source": "<a href=\"http://twitter.com/download/iphone\"
rel=\"nofollow\">Twitter for iPhone</a>",
  "retweeted": false,
  "coordinates": null,
  "entities": {
   "symbols": [],
   "user_mentions": [],
   "hashtags": [],
   "urls": []
  },
  "in_reply_to_screen_name": null,
  "in_reply_to_user_id": null,
  "retweet_count": 1,
  "id_str": "415068819653402624",
  "favorited": false,
  "user": {
   "follow_request_sent": false,
   "profile_use_background_image": true,
   "default_profile_image": false,
   "id": 873346964,
   "verified": false,
   "profile_text_color": "333333",
   "profile_image_url_https": "https://pbs.twimg.com/profile_images/28
39568064/5c60d9e021399e9183e1aa79ccc4ea8d_normal.jpeg",
   "profile_sidebar_fill_color": "DDEEF6",
   "entities": {
    "description": {
     "urls": []
    }
   },
   "followers_count": 830,
   "profile_sidebar_border_color": "C0DEED",
   "id_str": "873346964",
   "profile_background_color": "C0DEED",
   "listed_count": 5,
   "profile_background_image_url_https":
"https://abs.twimg.com/images/themes/theme1/bg.png",
   "utc_offset": null,
   "statuses_count": 2942,
   "description": "",
   "friends_count": 1436,
   "location": "",
   "profile_link_color": "0084B4",
   "profile_image_url": "http://pbs.twimg.com/profile_images/283956806
4/5c60d9e021399e9183e1aa79ccc4ea8d_normal.jpeg",
   "following": false,
   "geo_enabled": true,
   "profile_background_image_url":
"http://abs.twimg.com/images/themes/theme1/bg.png",
   "screen_name": "davethered55",
   "lang": "en",
```

```
   "profile_background_tile": false,
   "favourites_count": 1,
   "name": "davethered",
   "notifications": false,
   "url": null,
   "created_at": "Thu Oct 11 10:01:57 +0000 2012",
   "contributors_enabled": false,
   "time_zone": null,
   "protected": false,
   "default_profile": true,
   "is_translator": false
  },
  "geo": null,
  "in_reply_to_user_id_str": null,
  "lang": "en",
  "created_at": "Mon Dec 23 10:38:12 +0000 2013",
  "in_reply_to_status_id_str": null,
  "place": null,
  "metadata": {
   "iso_language_code": "en",
   "result_type": "recent"
  }
 },
 "user": {
  "follow_request_sent": false,
  "profile_use_background_image": true,
  "default_profile_image": false,
  "id": 328926204,
  "verified": false,
  "profile_text_color": "333333",
  "profile_image_url_https": "https://pbs.twimg.com/profile_images/378
3167489/03f34729f7266639c5f3c6823fe41374_normal.jpeg",
  "profile_sidebar_fill_color": "F6F6F6",
  "entities": {
   "description": {
    "urls": []
   }
  },
  "followers_count": 400,
  "profile_sidebar_border_color": "EEEEEE",
  "id_str": "328926204",
  "profile_background_color": "ACDED6",
  "listed_count": 6,
  "profile_background_image_url_https":
"https://abs.twimg.com/images/themes/theme18/bg.gif",
  "utc_offset": 0,
  "statuses_count": 6186,
  "description": "wife, mum, grandma, housewife, sister, auntie,
neice, teacher, student and friend. different roles to different
people",
  "friends_count": 609,
  "location": "Hull, UK",
  "profile_link_color": "038543",
  "profile_image_url": "http://pbs.twimg.com/profile_images/3783167489
```

```
/03f34729f7266639c5f3c6823fe41374_normal.jpeg",
  "following": false,
  "geo_enabled": true,
  "profile_banner_url":
"https://pbs.twimg.com/profile_banners/328926204/1355100273",
  "profile_background_image_url":
"http://abs.twimg.com/images/themes/theme18/bg.gif",
  "screen_name": "Theresa_Vaughan",
  "lang": "en",
  "profile_background_tile": false,
  "favourites_count": 46,
  "name": "Theresa Vaughan",
  "notifications": false,
  "url": null,
  "created_at": "Mon Jul 04 08:22:01 +0000 2011",
  "contributors_enabled": false,
  "time_zone": "London",
  "protected": false,
  "default_profile": false,
  "is_translator": false
 },
 "geo": null,
 "in_reply_to_user_id_str": null,
 "lang": "en",
 "created_at": "Mon Dec 23 10:59:11 +0000 2013",
 "in_reply_to_status_id_str": null,
 "place": null,
 "metadata": {
  "iso_language_code": "en",
  "result_type": "recent"
 }
}
```


In essence, all the code does is repeatedly make requests to the Search API. One thing that might initially catch you off guard if you've worked with other web APIs (including version 1 of Twitter's API) is that there's no explicit concept of pagination in the Search API itself. Reviewing the API documentation reveals that this is a intentional decision, and there are some good reasons for taking a cursoring approach instead, given the highly dynamic state of Twitter resources. The best practices for cursoring vary a bit throughout the Twitter developer platform, with the Search API providing a slightly simpler way of navigating search results than other resources such as timelines.

Search results contain a special search_metadata node that embeds a next_results field with a query string that provides the basis of a subsequent query. If we weren't using a library like twitter to make the HTTP requests for us, this preconstructed query string would just be appended to the Search API URL, and we'd update it with additional parameters for handling OAuth. However, since we are not making our HTTP requests directly, we must parse the query string into its constituent key/value pairs and provide them as keyword arguments.

In Python parlance, we are unpacking the values in a dictionary into keyword arguments that the function receives. In other words, the function call inside of the for loop in Example 1.5, "Collecting search results" ultimately invokes a function such as twitter_api.search.tweets(q='%23MentionSomeoneImportantForYou', include_entities=1, max_id=313519052523986943) even though it appears in the source code as twitter_api.search.tweets(**kwargs), with kwargs being a dictionary of key/value pairs.

Example 5: Analyzing the 140 Characters

The online documentation is always the definitive source for Twitter platform objects, and it's worthwhile to bookmark the Tweets page, because it's one that you'll refer to quite frequently as you get familiarized with the basic anatomy of a tweet. No attempt is made here or elsewhere in the book to regurgitate online documentation, but a few notes are of interest given that you might still be a bit overwhelmed by the 5 KB of information that a tweet comprises. For simplicity of nomenclature, let's assume that we've extracted a single tweet from the search results and stored it in a variable named t. For example, t.keys() returns the top-level fields for the tweet and t['id'] accesses the identifier of the tweet.

- The human-readable text of a tweet is available through "text": RT @hassanmusician: #MentionSomeoneImportantForYou God.
- The entities in the text of a tweet are conveniently processed for you and available through "entities": { "user_mentions": [{ "indices": [3, 18], "screen_name": "hassanmusician", "id": 56259379, "name": "Download the NEW LP!", "id_str": "56259379" }], "hashtags": [{ "indices": [20, 50], "text": "MentionSomeoneImportantForYou" }], "urls": [] }
- Clues as to the "interestingness" of a tweet are available through "favourite_count" and "retweet_count", which return the number of times it's been bookmarked or retweeted, respectively.
- If a tweet has been retweeted, the "retweeted_status" field provides significant detail about the original tweet itself and its author. Keep in mind that sometimes the text of a tweet changes as it is retweeted, as users add reactions or otherwise manipulate the text.
- The "retweeted" field denotes whether or not the authenticated user (via an authorised application) has retweeted this particular tweet. Fields that vary depending upon the point of view of the particular user are denoted in Twitter's developer documentation as perspectival, which means that their values will vary depending upon the perspective of the user.
- Additionally, note that only original tweets are retweeted from the standpoint of the API and information management. Thus, the retweet_count reflects the total number of times that the original tweet has been retweeted and should reflect the same value in both the original tweet and all subsequent retweets. In other words, retweets aren't retweeted. It may be a bit counterintuitive at first, but if you think you're retweeting a retweet, you're actually just retweeting the original tweet that you were exposed to through a proxy. See "Examining Patterns in Retweets" later in this chapter for a more nuanced discussion about the difference between retweeting vs quoting a tweet.

Next, let's distill the entities and the text of the tweets into a convenient data structure for further examination. Next example extracts the text, screen names, and hashtags from the tweets that are collected and introduces a Python idiom called a double (or nested) list comprehension. If you understand a (single) list comprehension, the code formatting should illustrate the double list comprehension as simply a collection of values that are derived from a nested loop as opposed to the results of a single loop. List comprehensions are particularly powerful because they usually yield substantial performance gains over nested lists and provide an intuitive (once you're familiar with them) yet terse syntax.

In [10]:
```
status_texts = [ status['text']
                 for status in statuses ]

screen_names = [ user_mention['screen_name']
                 for status in statuses
                     for user_mention in status['entities']
                     ['user_mentions'] ]

hashtags = [ hashtag['text']
             for status in statuses
                 for hashtag in status['entities']['hashtags'] ]

# Compute a collection of all words from all tweets
words = [ w
```

```
          for t in status_texts
              for w in t.split() ]

# Explore the first 5 items for each...

print json.dumps(status_texts[0:5], indent=1)
print json.dumps(screen_names[0:5], indent=1)
print json.dumps(hashtags[0:5], indent=1)
print json.dumps(words[0:5], indent=1)
```

```
[
 "RT @davethered55: To ALL E'ON customers who, from January 17th, will
have to pay \u00a3165 a year standing charge REFUSE. I'm going to and
I'm p\u2026",
 "RT @davethered55: that's so I know how often E'ON check my meters.
So much for cheaper tariff. Mine's gone up from \u00a385 annually to
\u00a3254. Sh\u2026",
 "RT @davethered55: @RachelReevesMP Just been told by E'ON I'm NOT on
Age UK tariff but now on Energy Plan so bill rises from \u00a385 to
\u00a3254 ann\u2026",
 "@Number10gov I'm going to REFUSE to pay additional \u00a3165 a year
and when E'ON take me to court gonna call Cameron to explain how it's
CHEAPER",
 "@Number10gov E'ON cancelled my Age UK tariff and put me on Energy
Plan INCREASING my bill from \u00a385 to \u00a3254 a year."
]
[
 "davethered55",
 "davethered55",
 "davethered55",
 "RachelReevesMP",
 "Number10gov"
]
[
 "Al\u00f4Cidade",
 "Cidade10",
 "NowPlaying",
 "thoughtprovoking",
 "thoughtprovoking"
]
[
 "RT",
 "@davethered55:",
 "To",
 "ALL",
 "E'ON"
]
```

Example 6: Frequency Analysis

In Python 2.7, a collections module is available and it provides a counter that makes computing a frequency distribution rather trivial. Here we show ow to use a Counter to compute frequency distributions as ranked lists of terms. Among the more compelling reasons for mining Twitter data is to try to answer the question of what people are talking about right now. One of the simplest techniques you could apply to answer this question is basic frequency analysis, just as

we are performing below

In [11]:
```
from collections import Counter

for item in [words, screen_names, hashtags]:
    c = Counter(item)
    print c.most_common()[:10] # top 10
    print
```

```
[(u'on', 73), (u'to', 26), (u'a', 24), (u'E', 17), (u'e', 17), (u'RT',
16), (u'the', 16), (u'my', 14), (u'by', 14), (u"I'm", 13)]

[(u'_TasteeMyLips', 6), (u'j_winfield', 5), (u'davethered55', 3),
(u'Polo_FdatHF', 3), (u'StarMovies_Ph', 3), (u'CHRISTandElle', 3),
(u'debbiedaywalker', 2), (u'YouTube', 2), (u'Number10gov', 2),
(u'RachelReevesMP', 2)]

[(u'SoundCloud', 5), (u'thoughtprovoking', 3), (u'NowPlaying', 2),
(u'DrFeelgood', 1), (u'BOOMONLINE', 1), (u'disabilin', 1),
(u'Cidade10', 1), (u'EUETS', 1), (u'EU2030', 1), (u'nowPlaying', 1)]
```

In [14]:
```
# Just in case --> !pip install prettytable

from prettytable import PrettyTable

for label, data in (('Word', words),
                    ('Screen Name', screen_names),
                    ('Hashtag', hashtags)):
    pt = PrettyTable(field_names=[label, 'Count'])
    c = Counter(data)
    [ pt.add_row(kv) for kv in c.most_common()[:10] ]
    pt.align[label], pt.align['Count'] = 'l', 'r' # Set column alignment
    print pt
```

```
+------+-------+
| Word | Count |
+------+-------+
| on   |    73 |
| to   |    26 |
| a    |    24 |
| E    |    17 |
| e    |    17 |
| RT   |    16 |
| the  |    16 |
| my   |    14 |
| by   |    14 |
| I'm  |    13 |
+------+-------+
+-----------------+-------+
| Screen Name     | Count |
+-----------------+-------+
| _TasteeMyLips   |     6 |
| j_winfield      |     5 |
| davethered55    |     3 |
| Polo_FdatHF     |     3 |
```

```
| StarMovies_Ph    |     3 |
| CHRISTandElle    |     3 |
| debbiedaywalker  |     2 |
| YouTube          |     2 |
| Number10gov      |     2 |
| RachelReevesMP   |     2 |
+------------------+-------+
+------------------+-------+
| Hashtag          | Count |
+------------------+-------+
| SoundCloud       |     5 |
| thoughtprovoking |     3 |
| NowPlaying       |     2 |
| DrFeelgood       |     1 |
| BOOMONLINE       |     1 |
| disabilin        |     1 |
| Cidade10         |     1 |
| EUETS            |     1 |
| EU2030           |     1 |
| nowPlaying       |     1 |
+------------------+-------+
```

Example 7: Lexical Diversity in the Tweets

A slightly more advanced measurement that involves *calculating simple frequencies and can be applied to unstructured text is a metric* called **lexical diversity**. Mathematically, this is an expression of the *number of unique tokens in the text divided by the total number of tokens in the text*, which are both elementary yet important metrics in and of themselves. Lexical diversity is an interesting concept in the area of interpersonal communications because it provides a quantitative measure for the diversity of an individual's or group's vocabulary. For example, suppose you are listening to someone who repeatedly says "and stuff" to broadly generalise information as opposed to providing specific examples to reinforce points with more detail or clarity. Now, contrast that speaker to someone else who seldom uses the word "stuff" to generalise and instead reinforces points with concrete examples. The speaker who repeatedly says "and stuff" would have a lower lexical diversity than the speaker who uses a more diverse vocabulary, and chances are reasonably good that you'd walk away from the conversation feeling as though the speaker with the higher lexical diversity understands the subject matter better.

As applied to tweets or similar online communications, lexical diversity can be worth considering as a primitive statistic for answering a number of questions, such as how broad or narrow the subject matter is that an individual or group discusses. Although an overall assessment could be interesting, breaking down the analysis to specific time periods could yield additional insight, as could comparing different groups or individuals. For example, it would be interesting to measure whether or not there is a significant difference between the lexical diversity of two soft drink companies such as Coca-Cola and Pepsi as an entry point for exploration if you were comparing the effectiveness of their social media marketing campaigns on Twitter.

With a basic understanding of how to use a statistic like lexical diversity to analyze textual content such as tweets, let's now compute the lexical diversity for statuses, screen names, and hashtags for our working data set:

In [15]:
```
# A function for computing lexical diversity
def lexical_diversity(tokens):
    return 1.0*len(set(tokens))/len(tokens)

# A function for computing the average number of words per tweet
def average_words(statuses):
    total_words = sum([ len(s.split()) for s in statuses ])
    return 1.0*total_words/len(statuses)
```

```
print lexical_diversity(words)
print lexical_diversity(screen_names)
print lexical_diversity(hashtags)
print average_words(status_texts)
```

```
0.559897501602
0.7375
0.708333333333
15.7676767677
```

Example 8: The most popular Retweets

Even though the user interface and many Twitter clients have long since adopted the native Retweet API used to populate status values such as retweet_count and retweeted_status, some Twitter users may prefer to quote a tweet, which entails a workflow involving copying and pasting the text and prepending "RT @username" or suffixing "/via @username" to provide attribution. A good exercise at this point would be to further analyse the data to determine if there was a particular tweet that was highly retweeted or if there were just lots of "one-off" retweets. The approach we'll take to find the most popular retweets is to simply iterate over each status update and store out the retweet count, originator of the retweet, and text of the retweet if the status update is a retweet. Next snipset of code shows how to capture these values with a list comprehension and sort by the retweet count to display the top few results.

In [17]:
```
retweets = [
            # Store out a tuple of these three values ...
            (status['retweet_count'],
             status['retweeted_status']['user']['screen_name'],
             status['text'])

            # ... for each status ...
            for status in statuses

            # ... so long as the status meets this condition.
                if status.has_key('retweeted_status')
           ]

# Slice off the first 5 from the sorted results
# and display each item in the tuple

pt = PrettyTable(field_names=['Count', 'Screen Name', 'Text'])
[ pt.add_row(row) for row in sorted(retweets, reverse=True)[:5] ]
pt.max_width['Text'] = 50
pt.align= 'l'
print pt
```

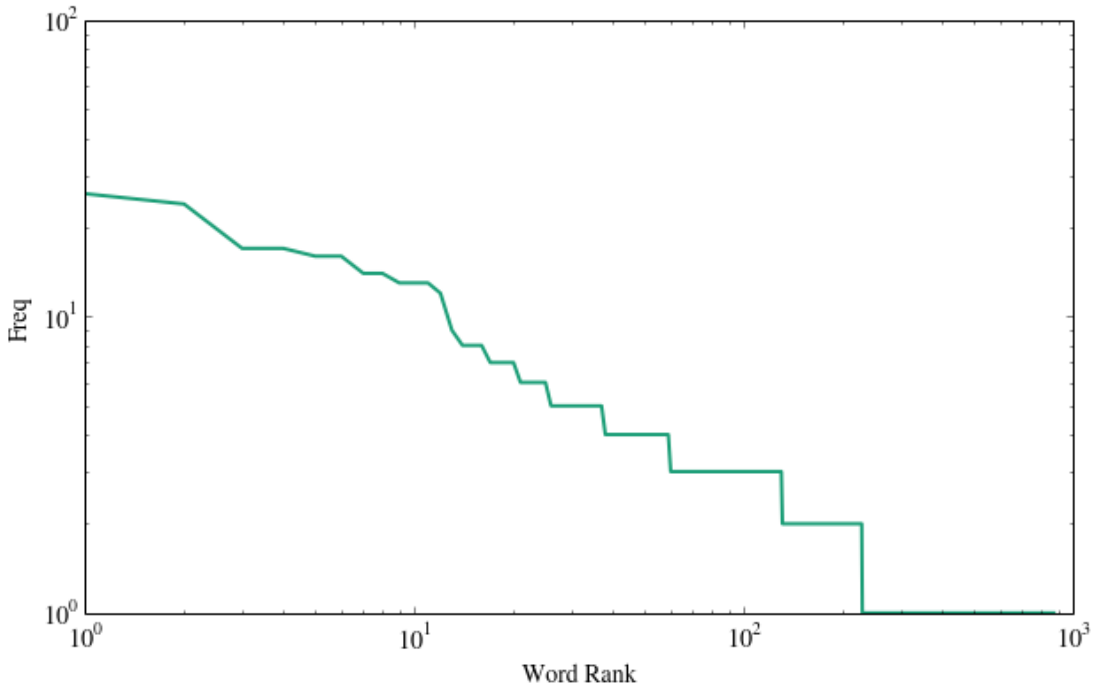
```
+-------+---------------+-----------------------------------------------
-------+
| Count | Screen Name   | Text
|
+-------+---------------+-----------------------------------------------
-------+
| 8     | MarketWatch   | RT @MarketWatch: E.ON looks to sell off
Spanish,   |
|       |               | Italian units http://t.co/0nsrbHoST8
|
| 3     | MarieLtdr     | RT @MarieLtdr: istouar trist:
|
|       |               |
|
```

```
|       |                | -on a à pène dormi set h é on doi ce -levé à
quel  |
|       |                | heur?
|
|       |                | - set heur trent
|
|       |                |
|
|       |                | rt si c tris (cc: @helenabrl @Hau...
|
| 3     | CHRISTandElle | RT @CHRISTandElle: While I'm watching Wall-E
on    |
|       |                | @starmovies_ph and surfing on my mobile,
this came |
|       |                | up in my news feed. #thoughtprovoking ht...
|
| 3     | CHRISTandElle | RT @CHRISTandElle: While I'm watching Wall-E
on    |
|       |                | @starmovies_ph and surfing on my mobile,
this came |
|       |                | up in my news feed. #thoughtprovoking ht...
|
| 3     | CHRISTandElle | RT @CHRISTandElle: While I'm watching Wall-E
on    |
|       |                | @starmovies_ph and surfing on my mobile,
this came |
|       |                | up in my news feed. #thoughtprovoking ht...
|
+-------+----------------+-----------------------------------------
-------+
```

Example 9: Plotting frequencies of words

In [24]:
```
word_counts = sorted(Counter(words).values(), reverse=True)

plt.loglog(word_counts)
plt.ylabel("Freq")
plt.xlabel("Word Rank")
```

Out [24]:
```
<matplotlib.text.Text at 0x76034b0>
```



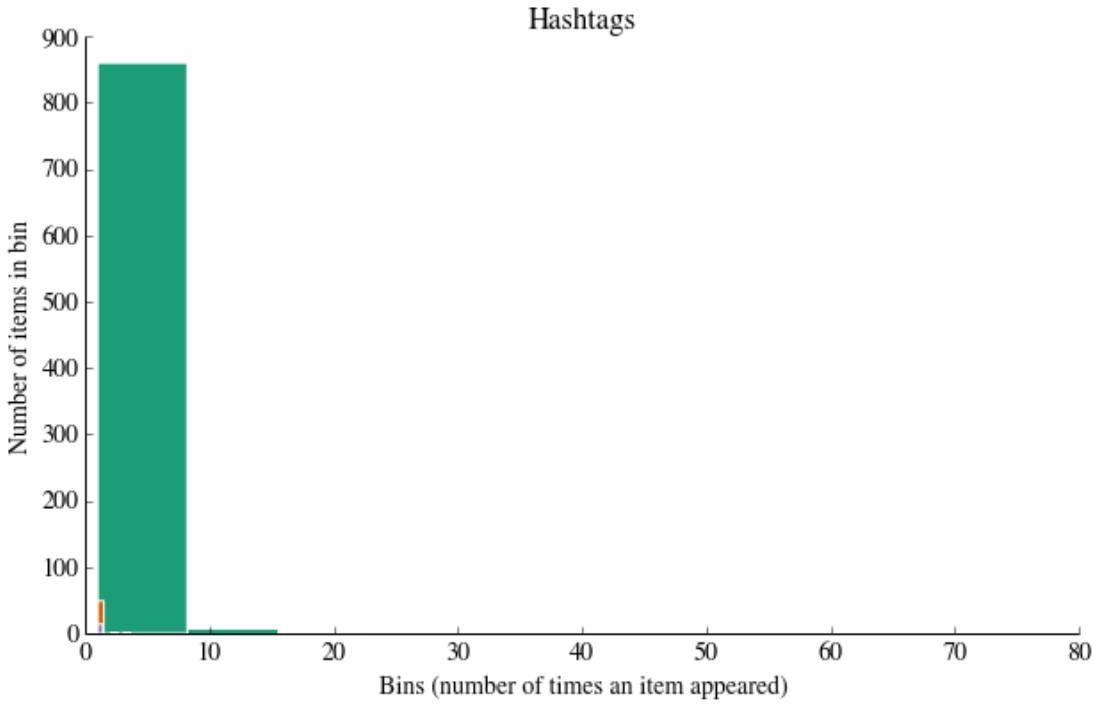
A plot of frequency values is intuitive and convenient, but it can also be useful to group together data values into bins that correspond to a range of frequencies. For example, how many words have a frequency between 1 and 5, between 5 and 10, between 10 and 15, and so forth? A histogram is designed for precisely this purpose and provides a convenient visualisation for displaying tabulated frequencies as adjacent rectangles, where the area of each rectangle is a measure of the data values that fall within that particular range of values.

Next histograms of tabulated frequency data for words, screen names, and hashtags, each displaying a particular kind of data that is grouped by frequency, look back to the corresponding tabular data and consider that there are a large number of words, screen names, or hashtags that have low frequencies and appear few times in the text; however, when we combine all of these low-frequency terms and bin them together into a range of "all words with frequency between 1 and 10," we see that the total number of these low-frequency words accounts for most of the text. More concretely, we see that there are approximately 10 words that account for almost all of the frequencies as rendered by the area of the large blue rectangle, while there are just a couple of words with much higher frequencies: "#MentionSomeoneImportantForYou" and "RT," with respective frequencies of 34 and 92 as given by our tabulated data.

Likewise, in the histogram of retweet frequencies, we see that there are a select few tweets that are retweeted with a much higher frequencies than the bulk of the tweets, which are retweeted only once and account for the majority of the volume given by the largest blue rectangle on the left side of the histogram.

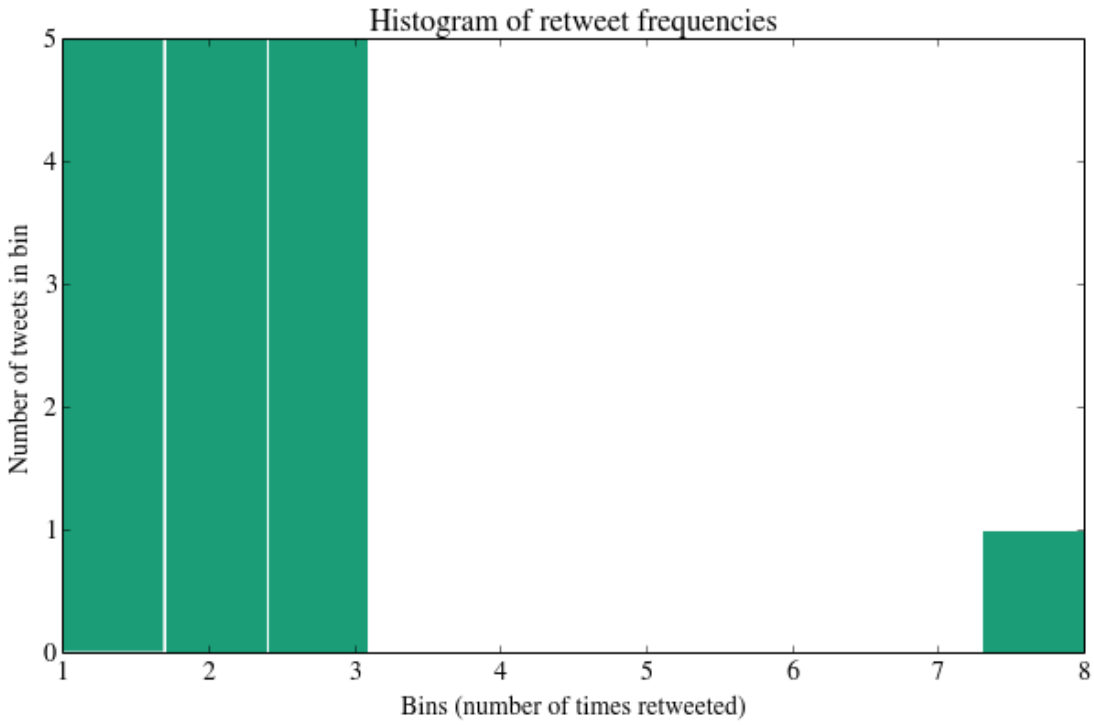
In [38]:
```
for label, data in (('Words', words),
                    ('Screen Names', screen_names),
                    ('Hashtags', hashtags)):

    # Build a frequency map for each set of data
    # and plot the values
    c = Counter(data)
    plt.hist(c.values())
    remove_border()
    # Add a title and y-label ...
    plt.title(label)
    plt.ylabel("Number of items in bin")
    plt.xlabel("Bins (number of times an item appeared)")
```

```
    # ... and display as a new figure
    #plt.figure()
    #remove_border()
```



In [29]:
```
# Using underscores while unpacking values in
# a tuple is idiomatic for discarding them

counts = [count for count, _, _ in retweets]

plt.hist(counts)
plt.title("Histogram of retweet frequencies")
plt.xlabel('Bins (number of times retweeted)')
plt.ylabel('Number of tweets in bin')

print counts
```

```
[1, 1, 1, 3, 3, 2, 3, 2, 1, 2, 2, 8, 3, 2, 1, 3]
```



7.3 Transferring data using the Clipboard

A handy way to grab data is to use the **read_clipboard method**, which takes the contents of the **clipboard buffer** and passes them to the read_table method. For instance, you can copy the following text to the clipboard (CTRL-C on many operating systems):

A B C
x 1 4 p
y 2 5 q
z 3 6 r

And then import the data directly to a **DataFrame** by calling:

```
In [25]: clipdf = pd.read_clipboard()
         clipdf
```

Out [25]:

	A	B	C
x	1	4	p
y	2	5	q
z	3	6	r

```
In [28]: df = pd.DataFrame(np.random.randn(10,10))
         df.to_clipboard()
         pd.read_clipboard()
```

Out [28]:

```
          0         1         2         3         4         5
6         7         8         9
0  0.445484 -1.726939 -0.803790 -0.352417 -0.979033 -1.337881
-0.867255 -0.523532  0.320852  0.843440
1 -0.247054  0.066216  0.264646 -0.528848 -0.399268  0.265917
1.886302 -1.560952 -0.431752 -0.642654
2  1.079251  0.801562  0.718280 -1.053267 -0.699632  0.688986
0.449936  1.048859 -0.395855  2.099105
3  0.477199 -0.331622  0.192500 -0.690828 -0.682078 -0.324446
0.337568  0.437147  0.623326 -0.302053
4 -0.102457  0.232483  0.613896  1.159289 -0.792233  0.726974
-0.775127 -1.914875 -0.033875 -0.133289
5  0.435930  0.986884  1.229685 -0.393746  1.812091 -1.049905
-0.514411  0.269162  1.986914  0.797323
6 -0.377658  0.885798  0.418364 -0.287936  0.584793 -1.976167
0.633389 -0.911546 -1.002134 -1.177509
7  0.127258 -0.861335  0.452441  1.437760 -0.558890  0.886835
1.407203 -0.467110  0.517097  0.496009
8  0.145878  1.430670  1.821520  1.515288 -2.437913 -1.568449
-2.389885  0.924258  0.391467  0.061432
9  1.001468 -0.929612 -0.979877  1.354407 -1.094605  0.564911
-1.622655  2.652130  0.958631  1.241469
```

7.4 Working with Data Bases: SQLite

Obviously, all the data retrived can be saved in a **Data Base** (SQL or NoSQL) using the appropiate package (e.g. `import MySQLdb`, `import sqlite3` or `import sqlalchemy`). About *data persistence* you can find more information here. Here we will work with he most common and extended SQL database: **SQLite**. It is is a relational database management system contained in a small (around 700 KB) C programming library. In contrast to other database management systems, SQLite is not a separate process that is accessed from the client application, but an integral part of it. SQLite is the most widely deployed SQL database engine, and the source code for SQLite is in the public domain.

The Basics of SQLite with Python

- **First**, `connect` to the database using the database library's `connect` method.
- **Second**, get a `cursor` which will allow us to `execute` SQL commands
- **Third**, We can now execute any SQL commands that we want in the database using the cursor's `execute` method. Querying the database simply involves writing the appropriate SQL and placing it inside a string in the `execute` method call.

Let's start importing the package, creating a new Data Base, a Connexion and a cursor... and checking the SQLite version:

In [27]:

```
import sqlite3 as sql3

# First, build the DataBase and the conexion via SQLite3
db = sql3.connect('test.db')

# Second, build the CURSOR which let us execute SQL commands
cur = db.cursor()
```

```
# Third, We can now execute any SQL commands that we want in the
# database using the cursor's execute method

cur.execute('SELECT SQLITE_VERSION()')
data = cur.fetchone()

print "SQLite version: %s" % data
```

```
SQLite version: 3.6.21
```

Let us begin creating a Table in test.db

In [28]:
```
db = sql3.connect('test.db')
cur = db.cursor()

cur.execute("DROP TABLE IF EXISTS Cars")
cur.execute("CREATE TABLE Cars(Id INT, Name TEXT, Price INT)")
```

Out [28]:
```
<sqlite3.Cursor at 0x7690fa0>
```

Inserting and Quering Data

In [33]:
```
db = sql3.connect('test.db')

with db:

    cur = db.cursor()
    cur.execute("INSERT INTO Cars VALUES(1,'Audi',52642)")
    cur.execute("INSERT INTO Cars VALUES(2,'Mercedes',57127)")
    cur.execute("INSERT INTO Cars VALUES(3,'Skoda',9000)")
    cur.execute("INSERT INTO Cars VALUES(4,'Volvo',29000)")
    cur.execute("INSERT INTO Cars VALUES(5,'Bentley',350000)")
    cur.execute("INSERT INTO Cars VALUES(6,'Citroen',21000)")
    cur.execute("INSERT INTO Cars VALUES(7,'Hummer',41400)")
    cur.execute("INSERT INTO Cars VALUES(8,'Volkswagen',21600)")
```

or also...

In [43]:
```
cars = (
    (1, 'Audi', 52642),
    (2, 'Mercedes', 57127),
    (3, 'Skoda', 9000),
    (4, 'Volvo', 29000),
    (5, 'Bentley', 350000),
    (6, 'Hummer', 41400),
    (7, 'Volkswagen', 21600)
)

con = sql3.connect('test.db')

with con:

    cur = con.cursor()

    # This script drops a Cars table if it exists and (re)creates it.
    cur.execute("DROP TABLE IF EXISTS Cars")
    cur.execute("CREATE TABLE Cars(Id INT, Name TEXT, Price INT)")
```

```
    # The first SQL statement drops the Cars table, if it exists.
    # The second SQL statement creates the Cars table.

    cur.executemany("INSERT INTO Cars VALUES(?, ?, ?)", cars)
```

In Python, we can use the `fetchall()` method to fetch all the records in the table:

In [24]:
```
import sqlite3 as sql3

db = sql3.connect('test.db')

cur = db.cursor()

cur.execute('SELECT * FROM Cars')

rows = cur.fetchall()

# We can print all ...
print rows

print '='*65

# Or one by one from rows...

for row in rows:
    print row
```

```
[(1, u'Audi', 62300), (2, u'Mercedes', 57127), (3, u'Skoda', 9000),
(4, u'Volvo', 29000), (5, u'Bentley', 350000), (6, u'Citroen', 21000),
(7, u'Hummer', 41400), (8, u'Volkswagen', 21600)]
=================================================================
(1, u'Audi', 62300)
(2, u'Mercedes', 57127)
(3, u'Skoda', 9000)
(4, u'Volvo', 29000)
(5, u'Bentley', 350000)
(6, u'Citroen', 21000)
(7, u'Hummer', 41400)
(8, u'Volkswagen', 21600)
```

Alternatively, to get the results into Python we then use either the `fetchone()` method to fetch one record at a time (it returns None when there are no more records to fetch so that you know when to stop)

In [40]:
```
con = sql3.connect('test.db')
cur = con.cursor()

cur.execute('SELECT * FROM Cars')
record = cur.fetchone()

while record:
    print record
    record = cur.fetchone()
```

```
(1, u'Audi', 52642)
(2, u'Mercedes', 57127)
(3, u'Skoda', 9000)
(4, u'Volvo', 29000)
(5, u'Bentley', 350000)
```

```
(6, u'Citroen', 21000)
(7, u'Hummer', 41400)
(8, u'Volkswagen', 21600)
```

Another possibility ...

In [41]:
```
con = sql3.connect('test.db')

with con:

    cur = con.cursor()
    cur.execute("SELECT * FROM Cars")

    rows = cur.fetchall()

    for row in rows:
        print row[0], row[1], row[2]
```

```
1 Audi 52642
2 Mercedes 57127
3 Skoda 9000
4 Volvo 29000
5 Bentley 350000
6 Citroen 21000
7 Hummer 41400
8 Volkswagen 21600
```

And a technically improved version of the previous code to retrieve data could be...

In [42]:
```
con = sql3.connect('test.db')

with con:

    cur = con.cursor()
    cur.execute("SELECT * FROM Cars")

    while True:

        row = cur.fetchone()

        if row == None:
            break

        print row[0], row[1], row[2]
```

```
1 Audi 52642
2 Mercedes 57127
3 Skoda 9000
4 Volvo 29000
5 Bentley 350000
6 Citroen 21000
7 Hummer 41400
8 Volkswagen 21600
```

Parameterized queries

When we use parameterized queries, we use placeholders instead of directly writing the values into the statements. Parameterized queries increase security and performance.

The Python **SQLite3** module supports two types of placeholders. Question marks and named placeholders.

In [44]:
```
uId = 1
uPrice = 62300

con = sql3.connect('test.db')

with con:

    cur = con.cursor()

    cur.execute("UPDATE Cars SET Price=? WHERE Id=?", (uPrice, uId))
    con.commit()

    print "Number of rows updated: %d" % cur.rowcount
```

```
Number of rows updated: 1
```

The second example uses parameterized statements with named placeholders:

In [45]:
```
uId = 4

con = sql3.connect('test.db')

with con:

    cur = con.cursor()

    cur.execute("SELECT Name, Price FROM Cars WHERE Id=:Id",
        {"Id": uId})
    con.commit()

    row = cur.fetchone()
    print row[0], row[1]
```

```
Volvo 29000
```

Metadata

Metadata is information about the data in the database. Metadata in a SQLite contains information about the tables and columns, in which we store data. Number of rows affected by an SQL statement is a metadata. Number of rows and columns returned in a result set belong to metadata as well.

Metadata in SQLite can be obtained using the PRAGMA command. SQLite objects may have attributes, which are metadata. Finally, we can also obtain specific metadata from querying the SQLite system sqlite_master table.

In [46]:
```
con = sql3.connect('test.db')

with con:

    cur = con.cursor()

    cur.execute('PRAGMA table_info(Cars)')

    data = cur.fetchall()

    for d in data:
        print d[0], d[1], d[2]
```

```
0 Id INT
1 Name TEXT
2 Price INT
```

In the next example, we want to print all rows from the *Cars table* with their column names.

In [47]:
```
con = sql3.connect('test.db')

with con:

    cur = con.cursor()
    cur.execute('SELECT * FROM Cars')

    col_names = [cn[0] for cn in cur.description]

    rows = cur.fetchall()

    print "%-5s %-15s %s" % (col_names[0], col_names[1], col_names[2])

    for row in rows:
        print "%-5s %-15s %s" % row
```

```
Id    Name            Price
1     Audi            62300
2     Mercedes        57127
3     Skoda           9000
4     Volvo           29000
5     Bentley         350000
6     Hummer          41400
7     Volkswagen      21600
```

Another example related to the metadata, we list all tables in the `test.db` database.

In [48]:
```
con = sql3.connect('test.db')

with con:

    cur = con.cursor()
    cur.execute("SELECT name FROM sqlite_master WHERE type='table'")

    rows = cur.fetchall()

    for row in rows:
        print row[0]
```

```
Friends
Cars
```

Export and Import Data

We can dump data in an SQL format to create a simple backup of our database tables:

In [49]:
```
# The data from the table is being written to the file:

def writeData(data):

    f = open('cars.sql', 'w')

    with f:
        f.write(data)

# We create a temporary table in the memory:

con = sql3.connect(':memory:')

# These lines create a Cars table, insert values and delete rows,
# where the Price is less than 30000 units.

with con:

    cur = con.cursor()

    cur.execute("DROP TABLE IF EXISTS Cars")
    cur.execute("CREATE TABLE Cars(Id INT, Name TEXT, Price INT)")
    cur.executemany("INSERT INTO Cars VALUES(?, ?, ?)", cars)
    cur.execute("DELETE FROM Cars WHERE Price < 30000")

    # The con.iterdump() returns an iterator to dump the database
    # in an SQL text format. The built-in join() function takes
    # the iterator and joins all the strings in the iterator separated
    # by a new line. This data is written to the cars.sql file in
    # the writeData() function.

    data = '\n'.join(con.iterdump())

    writeData(data)
```

In [50]:
```
print data
```

```
BEGIN TRANSACTION;
CREATE TABLE Cars(Id INT, Name TEXT, Price INT);
INSERT INTO "Cars" VALUES(1,'Audi',52642);
INSERT INTO "Cars" VALUES(2,'Mercedes',57127);
INSERT INTO "Cars" VALUES(5,'Bentley',350000);
INSERT INTO "Cars" VALUES(6,'Hummer',41400);
COMMIT;
```

Now, we are going to perform a *reverse* operation. We will import the dumped table back into memory.

In [51]:
```
def readData():

    f = open('cars.sql', 'r')
```

```
    with f:
        data = f.read()
        return data


con = sql3.connect(':memory:')

with con:

    cur = con.cursor()

    sql_query = readData()
    cur.executescript(sql_query)

    cur.execute("SELECT * FROM Cars")

    rows = cur.fetchall()

    for row in rows:
        print row
```

```
(1, u'Audi', 52642)
(2, u'Mercedes', 57127)
(5, u'Bentley', 350000)
(6, u'Hummer', 41400)
```

Transactions

A **transaction** is an atomic unit of database operations against the data in one or more databases. The effects of all the SQL statements in a transaction can be either all committed to the database or all rolled back. In SQLite, any command other than the `SELECT` will start an implicit transaction. Also, within a transaction a command like `CREATE TABLE ...`, `VACUUM`, `PRAGMA`, will commit previous changes before executing. Manual transactions are started with the `BEGIN TRANSACTION` statement and finished with the `COMMIT` or `ROLLBACK` statements. SQLite supports three non-standard transaction levels. `DEFERRED`, `IMMEDIATE` and `EXCLUSIVE`. SQLite Python module also supports an autocommit mode, where all changes to the tables are immediately effective.

In [52]:
```
# We create a friends table and try to fill it with data.
# However, the data is not commited...
# because the commit() menhod is commented.
# If we uncomment the line,
# the line will be written to the table:

#import sqlite3 as sql

try:
    con = sql3.connect('test.db')
    cur = con.cursor()
    cur.execute("DROP TABLE IF EXISTS Friends")
    cur.execute("CREATE TABLE Friends(Id INTEGER PRIMARY KEY, Name TEXT)")
    cur.execute("INSERT INTO Friends(Name) VALUES ('Tom')")
    cur.execute("INSERT INTO Friends(Name) VALUES ('Rebecca')")
    cur.execute("INSERT INTO Friends(Name) VALUES ('Jim')")
    cur.execute("INSERT INTO Friends(Name) VALUES ('Robert')")

#---> con.commit()

except sql3.error, e:

    if con:
        con.rollback()
```

```
    print "Error %s:" % e.args[0]
    sys.exit(1)

finally:

    if con:
        con.close()
```

7.5 Code Example 04: A Data Base of 10.000 Movies

The `pandas.io.sql` module provides a collection of query wrappers to both facilitate data retrieval and to reduce dependency on DB-specific API. These wrappers only support the Python database adapters which respect the Python DB-API.

Let us use, in the following example, a list of the 10.000 movies made since 1950 with the most IMDB user ratings. Download the data at http://bit.ly/cs109_imdb and save it as text file in your working directory.

In [25]:
```
# First, load the data:
names = ['imdbID', 'title', 'year', 'score', 'votes', 'runtime', 'genres']
movies = pd.read_csv('imdb_top_10000.txt', delimiter='\t', names = names)\
        .dropna()

# In order to load Dataframes into a SQl dataBase, we need
# a set of functions from pandas to link with the SQL

from pandas.io import sql


# third, let us create your connection.
cnx = sql3.connect('movies.db')
cnx.execute("DROP TABLE IF EXISTS movies")

# Load the DataFrame in SQLite3
sql.write_frame(movies, name='movies', con=cnx)

# Retrieving the data from SQLite3
# p1 = sql.read_frame('SELECT * FROM movies', cnx)

p3 = sql.read_frame('SELECT * FROM movies WHERE year=2001', cnx)

print "Size of the Table of the Database (rows,columns): ",  p3.shape
print '='*65
p3.head()
```

```
Size of the Table of the Database (rows,columns):  (353, 7)
=================================================================
```

Out [25]:

	imdbID	title	year	score	votes	runtime	genres
0	tt0120737	The Lord of the Rings: The Fellowship of the R...	2001	8.8	451263	178 mins.	Action|Adventure|Drama|Fantasy
1	tt0246578	Donnie Darko (2001)	2001	8.2	246756	113 mins.	Drama|Mystery|Sci-Fi
2	tt0211915	Am\xe9lie (2001)	2001	8.5	215732	122 mins.	Comedy|Fantasy|Romance
3	tt0126029	Shrek (2001)	2001	7.9	185372	90 mins.	Animation|Adventure|Comedy|Family|Fantasy
4	tt0241527	Harry Potter and the Sorcerer's Stone (2001)	2001				

```
7.2  159229  152 mins.          Adventure|Family|Fantasy|Mystery
```

In [66]:
```
cnx = sql3.connect('movies.db')

with cnx:

    cur = cnx.cursor()
    cur.execute('PRAGMA table_info(movies)')
    # cur.execute("DROP TABLE IF EXISTS movies")
    data = cur.fetchall()

    for d in data:
        print d[0], d[1], d[2]
```

```
0 imdbID TEXT
1 title TEXT
2 year INTEGER
3 score REAL
4 votes INTEGER
5 runtime TEXT
6 genres TEXT
```

7.6 Code Example 05: Price Time-Series and DataFramework Storage in a SQLite Data Base

Functions from `pandas.io.data` extract data from various Internet sources into a DataFrame. Currently the following sources are supported:

- **Yahoo! Finance** with `web.DataReader(ticker,'yahoo', start, end)`
- **Google Finance** with `web.DataReader(ticker,'google', start, end)`
- **St. Louis FED (FRED)** with `web.DataReader('GDP', 'fred', start, end)`
- **Kenneth French's** data library with `web.DataReader("5_Industry_Portfolios", "famafrench")`

It should be noted, that various sources support different kinds of data, so not all sources implement the same methods and the data elements returned might also differ.

In [7]:
```
# Download data from Yahoo! Finance:

import pandas.io.data as web

start = pd.datetime(2013, 1, 1)
end = pd.datetime(2013, 12, 1)

# f=web.DataReader("F", 'yahoo', start, end)

all_data = {}

for ticker in ['AAPL', 'GOOG', 'MSFT', 'DELL', 'GS', 'MS', 'BAC']:
    all_data[ticker] = web.DataReader(ticker,'yahoo', start, end)
```

In [18]:
```
# Create a pandas-DataFrame with the info
# coming from Yahoo! Finance

prices = pd.DataFrame({tic: data['Adj Close']
                       for tic, data in all_data.iteritems()})
```

```
print prices[['AAPL','GOOG','GS']].describe()
print '='*65
prices.head()
```

```
              AAPL         GOOG          GS
count   231.000000   231.000000  231.000000
mean    458.926234   866.051082  154.092944
std      39.097435    81.248638    8.998906
min     383.180000   702.870000  129.240000
25%     427.175000   806.525000  147.780000
50%     447.730000   871.980000  155.660000
75%     492.085000   896.380000  161.740000
max     556.070000  1063.110000  169.190000
=================================================================
```

Out [18]:

Date	AAPL	BAC	DELL	GOOG	GS	MS	MSFT
2013-01-02	535.58	11.99	10.50	723.25	129.95	19.46	26.81
2013-01-03	528.82	11.92	10.75	723.67	129.24	19.42	26.45
2013-01-04	514.09	12.07	10.78	737.97	132.76	20.03	25.95
2013-01-07	511.06	12.05	10.87	734.75	132.51	19.64	25.91
2013-01-08	512.44	11.94	10.59	733.30	131.32	19.49	25.77

In [34]:
```
# Notice that writing your DataFrame into
# a database works only with SQLite.
# Moreover, the index will currently be
# dropped, therefore first, we have
# to move it as column

prices['Dates']=prices.index[:]
prices.head()
```

Out [34]:

Dates	AAPL	BAC	DELL	GOOG	GS	MS	MSFT
Date							
2013-01-02	535.58	11.99	10.50	723.25	129.95	19.46	26.81
2013-01-02 00:00:00							
2013-01-03	528.82	11.92	10.75	723.67	129.24	19.42	26.45
2013-01-03 00:00:00							
2013-01-04	514.09	12.07	10.78	737.97	132.76	20.03	25.95
2013-01-04 00:00:00							
2013-01-07	511.06	12.05	10.87	734.75	132.51	19.64	25.91
2013-01-07 00:00:00							
2013-01-08	512.44	11.94	10.59	733.30	131.32	19.49	25.77
2013-01-08 00:00:00							

In [35]:
```
# To load DataFrames into a SQLite DataBase we need something from pandas
# that transforms DataFrames into tables and back

from pandas.io import sql

# Create your connection

cnx = sql3.connect('prices.db')
```

```
# Load the DataFrame in SQLite3
cur = cnx.cursor()
cur.execute("DROP TABLE IF EXISTS prices")

sql.write_frame(prices, name='prices', con = cnx)
```

In [36]:
```
# Let's retrive the prices from the DataBase:

cnx = sql3.connect('prices.db')

with cnx:

    cur = cnx.cursor()
    cur.execute('PRAGMA table_info(prices)')

    table = cur.fetchall()

    for d in table:
        print d[0], d[1]
```

```
0 AAPL
1 BAC
2 DELL
3 GOOG
4 GS
5 MS
6 MSFT
7 Dates
```

In [37]:
```
# And finally, let's build the pandas-DataFrame of prices
# from the SQLite3-database

from pandas.io import sql
from pandas.lib import Timestamp

cnx = sql3.connect('prices.db')

allp =  sql.read_frame("SELECT * FROM prices", cnx)
allp.Dates = allp.Dates.apply(Timestamp)
allp = allp.set_index('Dates')

apple = sql.read_frame("SELECT AAPL,Dates FROM prices", cnx)
apple.Dates = apple.Dates.apply(Timestamp)
apple = apple.set_index('Dates')


# apple.set_index(rng)
print '='*65
print allp[['AAPL','GOOG','GS']].describe()
print '='*65
apple.head()
```

```
============================================================
              AAPL         GOOG          GS
count  231.000000   231.000000  231.000000
mean   458.926234   866.051082  154.092944
std     39.097435    81.248638    8.998906
min    383.180000   702.870000  129.240000
25%    427.175000   806.525000  147.780000
50%    447.730000   871.980000  155.660000
75%    492.085000   896.380000  161.740000
```

```
max    556.070000  1063.110000  169.190000
==========================================================================
```

Out [37]:

```
                AAPL
Dates
2013-01-02  535.58
2013-01-03  528.82
2013-01-04  514.09
2013-01-07  511.06
2013-01-08  512.44
```

CHAPTER

EIGHT

DATA EXPLORATION: BASIC WORKFLOW IN DATA ANALYSIS WITH PANDAS

In this chapter , we are following the methodology and some of the examples of *Chris Beaumont*, from CS109-Harvard.

The basic workflow to work with data and pandas would be as follows:

1. **Build** a **DataFrame** from the data (ideally, put *all* data in this object)
2. **Clean** the **DataFrame**. It should have the following properties:

- Each row describes a single object or observation
- Each column describes a property of that object
- Columns are numeric whenever appropriate
- Columns contain atomic properties that cannot be further decomposed

3. Explore **global properties**. Use histograms, scatter-plots, and aggregation functions to summarise the data.
4. Explore **group properties**. Use `groupby` and small multiples to compare subsets of the data.

This process transforms your data into a format which is easier to work with, gives you a basic overview of the data's properties, and likely generates several questions for you to followup in subsequent analysis.

8.1 Build a DataFrame

A **DataFrame** is a 2-dimensional labeled data structure with columns of potentially different types. It is defined in **pandas** package. You can think of it *like a spreadsheet* or *SQL table*, or *a dict of Series objects*. It is generally the most commonly used **pandas** object. Like Series, **DataFrame** accepts many different kinds of input:

- Dict of 1D ndarrays, lists, dicts, or Series
- 2-D numpy.ndarray
- Structured or record ndarray
- A Series
- Another DataFrame

Along with the data, you can optionally pass **index** (row labels) and **columns** (column labels) arguments. If you pass an index and / or columns, you are guaranteeing the index and / or columns of the resulting DataFrame. Thus, a dict of Series plus a specific index will discard all data not matching up to the passed index.

Let us see that in a simple example: names and births

In [101]:
```
names = ['Vlad','Sam','Tomm','Teo','Liana']
WorkID = [102, 340, 560, 765, 304]

print names
print WorkID
# to merge this two lists we will use the zip function

pyclass = zip(names,WorkID)
print pyclass
```

```
['Vlad', 'Sam', 'Tomm', 'Teo', 'Liana']
[102, 340, 560, 765, 304]
[('Vlad', 102), ('Sam', 340), ('Tomm', 560), ('Teo', 765), ('Liana',
304)]
```

Let us use now a **DataFrame** from **pandas**

In [102]:
```
df = pd.DataFrame(data = pyclass, columns=['Names','WorkID'])
df
```

Out [102]:

	Names	WorkID
0	Vlad	102
1	Sam	340
2	Tomm	560
3	Teo	765
4	Liana	304

We can export the DataFrame to a `csv` file for storage or just because we want to use it in the next session

In [104]:
```
df.to_csv('WorkID_pyclass.csv', index=False, header=True)
del df
```

In [106]:
```
Location = 'C:\\Users\\Suso\\WorkID_pyclass.csv'
df = pd.read_csv (Location)
print df
print '-'*100
print df.Names
print '-'*100
print df.WorkID
```

```
   Names  WorkID
0   Vlad     102
1    Sam     340
2   Tomm     560
3    Teo     765
4  Liana     304
----------------------------------------------------------------------
------------------------------
0     Vlad
1      Sam
2     Tomm
3      Teo
4    Liana
Name: Names, dtype: object
----------------------------------------------------------------------
```

```
------------------------------
0    102
1    340
2    560
3    765
4    304
Name: WorkID, dtype: int64
```

Some basics about how to view the data

```
In [8]: df.head(2)
```

```
Out [8]:
  Names  Births
0  Vlad     102
1   Sam     340
```

```
In [9]: df.tail(-2)
```

```
Out [9]:
   Names  Births
2   Tomm     560
3    Teo     765
4  Liana     304
```

```
In [143]: df = pd.DataFrame({'int_col' : [1,2,6,8,-1],
                             'float_col' : [0.1, 0.2,0.2,10.1,None],
                             'str_col' : ['a','b',None,'c','a']})
          df
```

```
Out [143]:
   float_col  int_col str_col
0        0.1        1       a
1        0.2        2       b
2        0.2        6    None
3       10.1        8       c
4        NaN       -1       a
```

```
In [54]: df.index
```

```
Out [54]:
Int64Index([0, 1, 2, 3, 4], dtype=int64)
```

```
In [144]: df.T
```

```
Out [144]:
             0    1     2     3    4
float_col  0.1  0.2   0.2  10.1  NaN
int_col      1    2     6     8   -1
str_col      a    b  None     c    a
```

In [145]:
```
df.ix[1:,['float_col','int_col']]
```

Out [145]:

	float_col	int_col
1	0.2	2
2	0.2	6
3	10.1	8
4	NaN	-1

In [55]:
```
df.values
```

Out [55]:
```
array([[0.1, 1, 'a'],
       [0.2, 2, 'b'],
       [0.2, 6, None],
       [10.1, 8, 'c'],
       [nan, -1, 'a']], dtype=object)
```

In [56]:
```
df.describe()
```

Out [56]:

	float_col	int_col
count	4.00000	5.000000
mean	2.65000	3.200000
std	4.96689	3.701351
min	0.10000	-1.000000
25%	0.17500	1.000000
50%	0.20000	2.000000
75%	2.67500	6.000000
max	10.10000	8.000000

In [147]:
```
df.sort_index(axis=0, ascending=True)
```

Out [147]:

	float_col	int_col	str_col
0	0.1	1	a
1	0.2	2	b
2	0.2	6	None
3	10.1	8	c
4	NaN	-1	a

Another example, this time more statisticallly related...

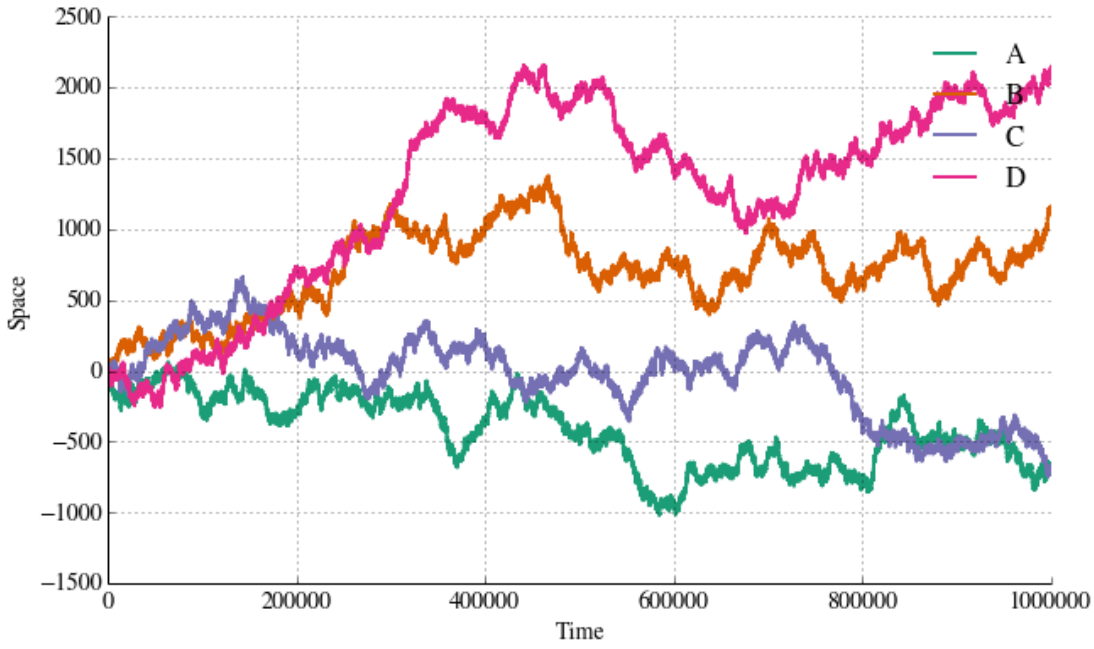
In [39]:
```
np.random.seed(12345)
#data = pd.DataFrame(np.random.randn(1000, 4))
df = pd.DataFrame(np.random.randn(1000000, 4).cumsum(0),
                  columns=['A', 'B', 'C', 'D'],
                  index=np.arange(0, 1000000, 1))

print df.head(15)
df.describe()
```

```
           A         B         C         D
0  -0.204708  0.478943 -0.519439 -0.555730
1   1.761073  1.872349 -0.426531 -0.273984
2   2.530095  3.118784  0.580659 -1.570205
3   2.805087  3.347697  1.933575 -0.683776
4   0.803450  2.975854  3.602601 -1.122346
5   0.263708  3.452839  6.851545 -2.143573
6  -0.313379  3.576961  7.154158 -1.619801
7  -0.312439  4.920770  6.440614 -2.450955
8  -2.682670  3.060010  5.579857 -1.890809
9  -3.948605  3.179837  4.516344 -1.557927
10 -6.308024  2.980294  2.974349 -2.528663
11 -7.615054  3.266643  3.352333 -3.282549
12 -7.283768  4.616386  3.422210 -3.035875
13 -7.295630  5.621197  4.749404 -3.955137
14 -8.844736  5.643382  5.507767 -4.615661
```

Out [39]:

	A	B	C	D
count	1000000.000000	1000000.000000	1000000.000000	1000000.000000
mean	-423.249054	702.366184	-18.853142	1264.969888
std	251.114913	287.158321	290.286721	678.587545
min	-1018.537576	0.478943	-744.922611	-259.002854
25%	-643.267952	519.927683	-148.238073	858.665973
50%	-428.466343	732.711656	35.948866	1473.969847
75%	-199.460833	900.041707	180.350337	1820.117473
max	63.375785	1377.444746	662.579391	2158.960227

In [40]:
```
df.plot()
plt.xlabel("Time")
plt.ylabel("Space")
plt.legend(frameon=False)
remove_border()
```



But let us play more *professionally*. Let us use a higher amount of data to do some serious Data Analysis: a list of the 10,000 movies made since 1950 with the most IMDB user ratings. Download the data at http://bit.ly/cs109_imdb and save it as text file in your working directory.

In [41]:
```
names = ['imdbID', 'title', 'year', 'score',
         'votes', 'runtime', 'genres']

data = pd.read_csv('imdb_top_10000.txt',
                   delimiter='\t', names = names).dropna()

print "Number of rows: %i" % data.shape[0]

data.head(10)  # print the first 10 rows
```

```
Number of rows: 9999
```

Out [41]:

```
      imdbID                                                title  year
score   votes    runtime                                genres
0  tt0111161                    The Shawshank Redemption (1994)  1994
9.2  619479  142 mins.                           Crime|Drama
1  tt0110912                              Pulp Fiction (1994)  1994
9.0  490065  154 mins.                       Crime|Thriller
2  tt0137523                                  Fight Club (1999)  1999
8.8  458173  139 mins.              Drama|Mystery|Thriller
3  tt0133093                                  The Matrix (1999)  1999
8.7  448114  136 mins.             Action|Adventure|Sci-Fi
4  tt1375666                                   Inception (2010)  2010
8.9  385149  148 mins.  Action|Adventure|Sci-Fi|Thriller
5  tt0109830                                 Forrest Gump (1994)  1994
8.7  368994  142 mins.                Comedy|Drama|Romance
6  tt0169547                               American Beauty (1999)  1999
8.6  338332  122 mins.                                 Drama
7  tt0499549                                     Avatar (2009)  2009
```

```
8.1  336855  162 mins.   Action|Adventure|Fantasy|Sci-Fi
8  tt0108052                          Schindler's List (1993)  1993
8.9  325888  195 mins.       Biography|Drama|History|War
9  tt0080684  Star Wars: Episode V - The Empire Strikes Back...  1980
8.8  320105  124 mins.    Action|Adventure|Family|Sci-Fi
```

8.2 Clean the DataFrame

There are several problems with the **DataFrame** at this point:

1. The runtime column describes a number, but is stored as a string
2. The genres column is not atomic – it aggregates several genres together. This makes it hard, for example, to extract which movies are Comedies.
3. The movie year is repeated in the title and year column

Fixing the runtime column

The following snipptet converts a string like '142 mins.' to the number 142:

In [42]:
```
dirty = '142 mins.'
number, text = dirty.split(' ')
clean = int(number)
print number
```

```
142
```

We can package this up into a list comprehension

In [43]:
```
clean_runtime = [float(r.split(' ')[0]) for r in data.runtime]
data['runtime'] = clean_runtime
data.head()
```

Out [43]:

	imdbID	title	year	score	votes	runtime	genres
0	tt0111161	The Shawshank Redemption (1994)	1994	9.2	619479	142	Crime|Drama
1	tt0110912	Pulp Fiction (1994)	1994	9.0	490065	154	Crime|Thriller
2	tt0137523	Fight Club (1999)	1999	8.8	458173	139	Drama|Mystery|Thriller
3	tt0133093	The Matrix (1999)	1999	8.7	448114	136	Action|Adventure|Sci-Fi
4	tt1375666	Inception (2010)	2010	8.9	385149	148	Action|Adventure|Sci-Fi|Thriller

In [44]:
```
data[['year','score']].head(15)
```

Out [44]:

	year	score
0	1994	9.2
1	1994	9.0

```
2   1999    8.8
3   1999    8.7
4   2010    8.9
5   1994    8.7
6   1999    8.6
7   2009    8.1
8   1993    8.9
9   1980    8.8
10  2005    8.3
11  1995    8.7
12  1991    8.7
13  1997    7.4
14  1995    8.4
```

In [45]:
```
data['score']
```

Out [45]:
```
0      9.2
1      9.0
2      8.8
3      8.7
4      8.9
5      8.7
6      8.6
7      8.1
8      8.9
9      8.8
10     8.3
11     8.7
12     8.7
13     7.4
14     8.4
...
9985    5.1
9986    7.3
9987    7.2
9988    5.0
9989    4.9
9990    6.3
9991    7.6
9992    5.8
9993    5.2
9994    6.7
9995    7.0
9996    5.2
9997    6.5
9998    6.5
9999    6.9
Name: score, Length: 9999, dtype: float64
```

In [46]:
```
print "type of df[['runtime']]:\t", type(data[['runtime']])
```

```
type of df[['runtime']]:        <class 'pandas.core.frame.DataFrame'>
```

In [47]:
```
data.votes.unique()
```

Out [47]:
```
array([619479, 490065, 458173, ...,   1358,   1357,   1356],
dtype=int64)
```

Splitting up the genres

We can use the concept of *indicator variables* to split the genres column into many columns. Each new column will correspond to a single genre, and each cell will be True or False.

In [48]:
```
#determine the unique genres
genres = set()
for m in data.genres:
    genres.update(g for g in m.split('|'))
genres = sorted(genres)

#make a column for each genre
for genre in genres:
    data[genre] = [genre in movie.split('|') for movie in data.genres]

data.head()
```

Out [48]:
```
      imdbID                          title  year  score   votes
runtime                          genres Action  Adult Adventure
Animation Biography Comedy  Crime  Drama Family Fantasy Film-Noir
History Horror  Music Musical Mystery   News Reality-TV Romance Sci-Fi
Sport Thriller    War Western
0  tt0111161  The Shawshank Redemption (1994)  1994    9.2  619479
142                       Crime|Drama  False  False     False
False     False  False   True   True  False   False     False   False
False  False   False   False  False      False   False  False  False
False  False   False
1  tt0110912             Pulp Fiction (1994)  1994    9.0  490065
154                    Crime|Thriller  False  False     False
False     False  False   True  False  False   False     False   False
False  False   False   False  False      False   False  False  False
True  False   False
2  tt0137523               Fight Club (1999)  1999    8.8  458173
139            Drama|Mystery|Thriller  False  False     False
False     False  False  False   True  False   False     False   False
False  False   False    True  False      False   False  False  False
True  False   False
3  tt0133093                The Matrix (1999)  1999    8.7  448114
136           Action|Adventure|Sci-Fi   True  False      True
False     False  False  False  False  False   False     False   False
False  False   False   False  False      False   False   True  False
False  False   False
4  tt1375666                 Inception (2010)  2010    8.9  385149
148  Action|Adventure|Sci-Fi|Thriller   True  False      True
```

```
False     False  False  False  False  False   False     False   False
False  False   False   False  False      False   False   True  False
True  False   False
```

Removing year from the title

We can fix each element by stripping off the last 7 characters

In [49]:
```
data['title'] = [t[0:-7] for t in data.title]
data.head()
```

Out [49]:

```
      imdbID                      title  year  score   votes  runtime
genres Action  Adult Adventure Animation Biography Comedy  Crime
Drama Family Fantasy Film-Noir History Horror  Music Musical Mystery
News Reality-TV Romance Sci-Fi  Sport Thriller    War Western
0  tt0111161  The Shawshank Redemption  1994    9.2  619479      142
Crime|Drama  False  False     False     False     False  False   True
True  False   False     False   False  False  False   False   False
False      False   False  False  False    False  False   False
1  tt0110912             Pulp Fiction  1994    9.0  490065      154
Crime|Thriller  False  False     False     False     False  False
True  False  False   False     False   False  False  False   False
False  False      False   False  False  False     True  False   False
2  tt0137523               Fight Club  1999    8.8  458173      139
Drama|Mystery|Thriller  False  False     False     False     False
False  False   True  False   False     False   False  False  False
False    True  False      False   False  False  False     True  False
False
3  tt0133093               The Matrix  1999    8.7  448114      136
Action|Adventure|Sci-Fi   True  False      True     False     False
False  False  False  False   False     False   False  False  False
False   False  False      False   False   True  False    False  False
False
4  tt1375666                Inception  2010    8.9  385149      148
Action|Adventure|Sci-Fi|Thriller   True  False      True     False
False  False  False  False  False   False     False   False  False
False   False   False  False      False   False   True  False     True
False   False
```

8.3 Explore Global Properties

Next, we get a handle on some basic, global summaries of the DataFrame.

Call `describe` on relevant columns

In [50]:
```
data[['score', 'runtime', 'votes']].describe()
```

Out [50]:

	score	runtime	votes
count	9999.000000	9999.000000	9999.000000
mean	6.385989	103.580358	16605.462946
std	1.189965	26.629310	34564.883945
min	1.500000	0.000000	1356.000000
25%	5.700000	93.000000	2334.500000
50%	6.600000	102.000000	4981.000000
75%	7.200000	115.000000	15278.500000
max	9.200000	450.000000	619479.000000

In [51]:
```
#Notice that a runtime of 0 looks suspicious. How many movies have that?
print len(data[data.runtime == 0])

#probably best to flag those bad data as NAN
data.runtime[data.runtime == 0] = np.nan
```

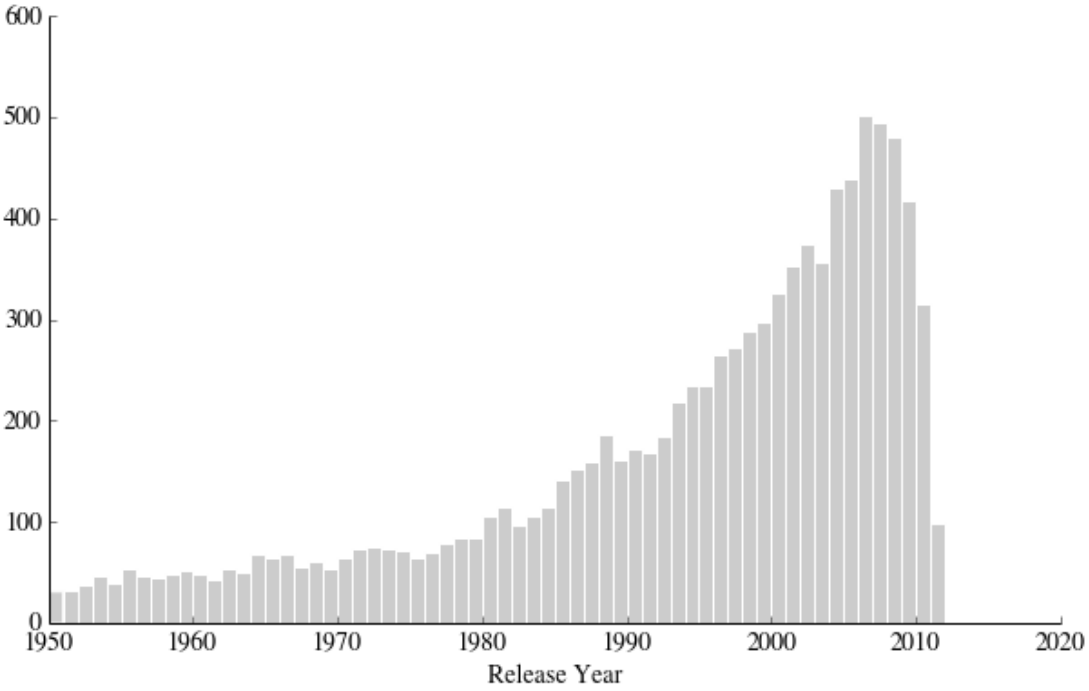
```
282
```

In [52]:
```
data.runtime.describe()
```

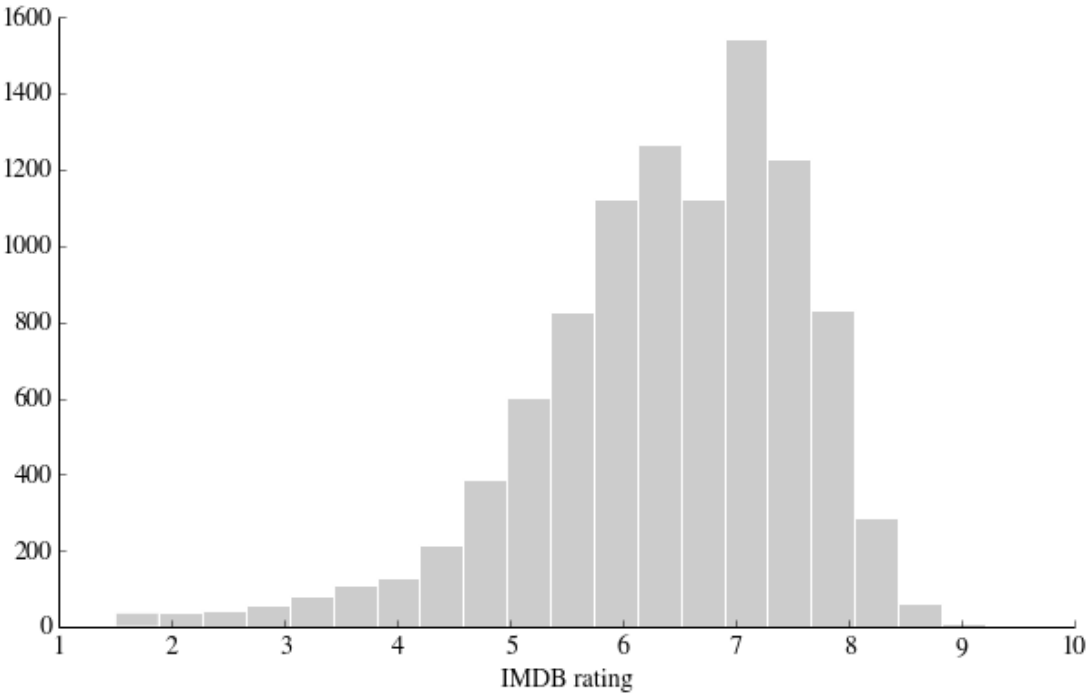
Out [52]:

```
count    9717.000000
mean      106.586395
std        20.230330
min        45.000000
25%        93.000000
50%       103.000000
75%       115.000000
max       450.000000
dtype: float64
```

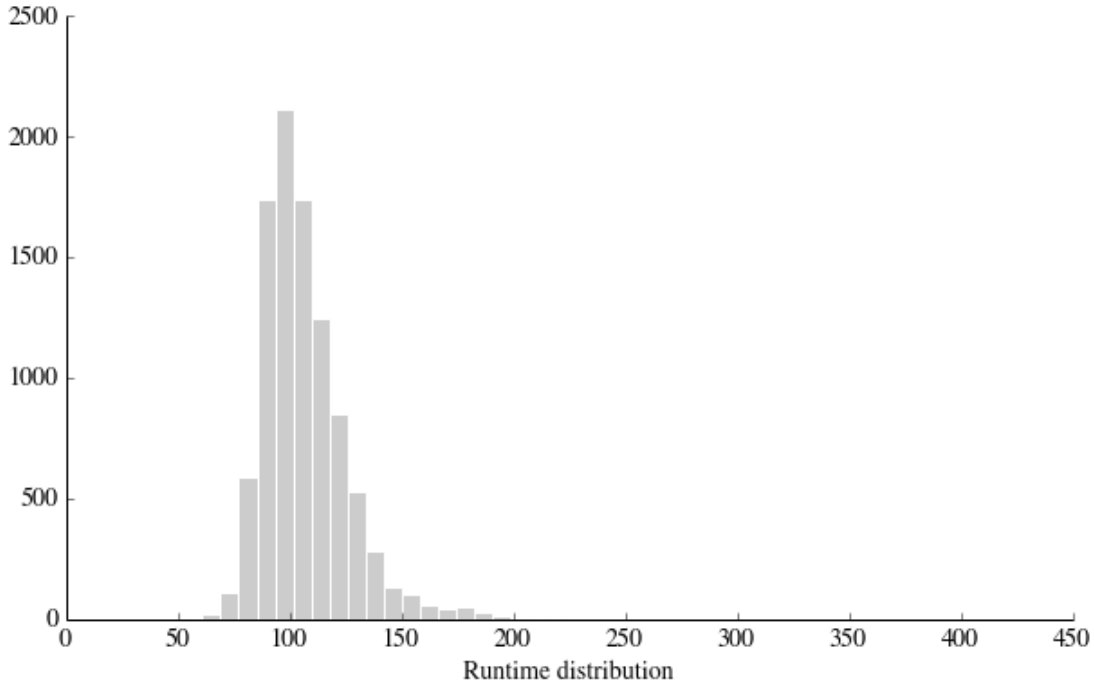
In [53]:
```
plt.hist(data.year, bins=np.arange(1950, 2013), color='#cccccc')
plt.xlabel("Release Year")
remove_border()
```



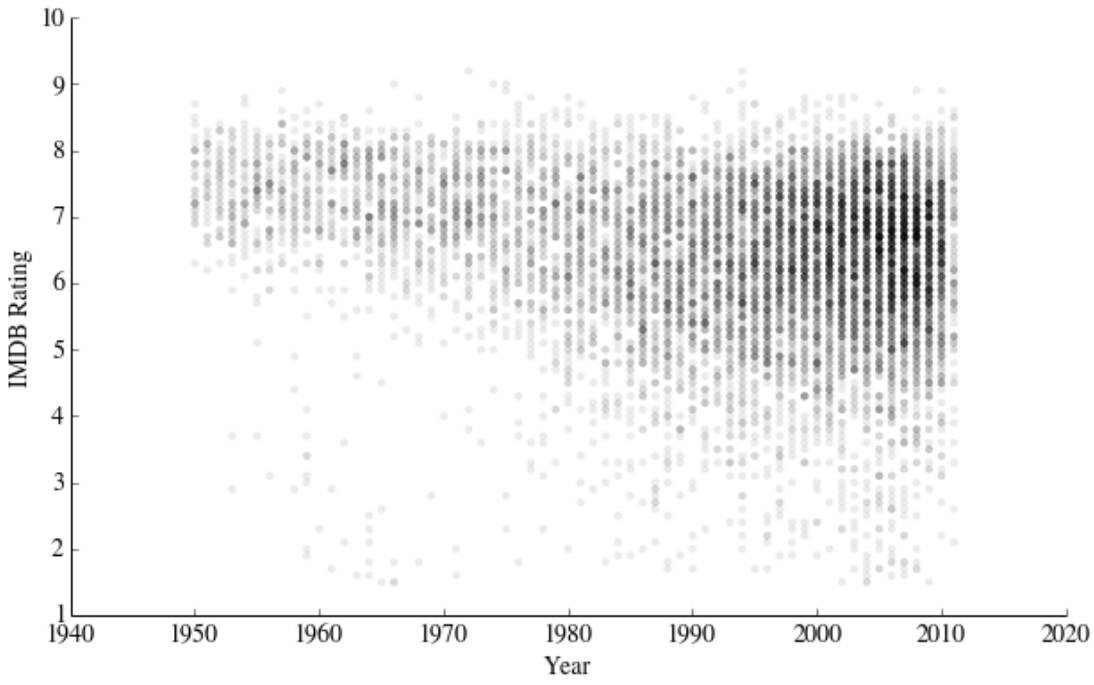
In [54]:
```
plt.hist(data.score, bins=20, color='#cccccc')
plt.xlabel("IMDB rating")
remove_border()
```



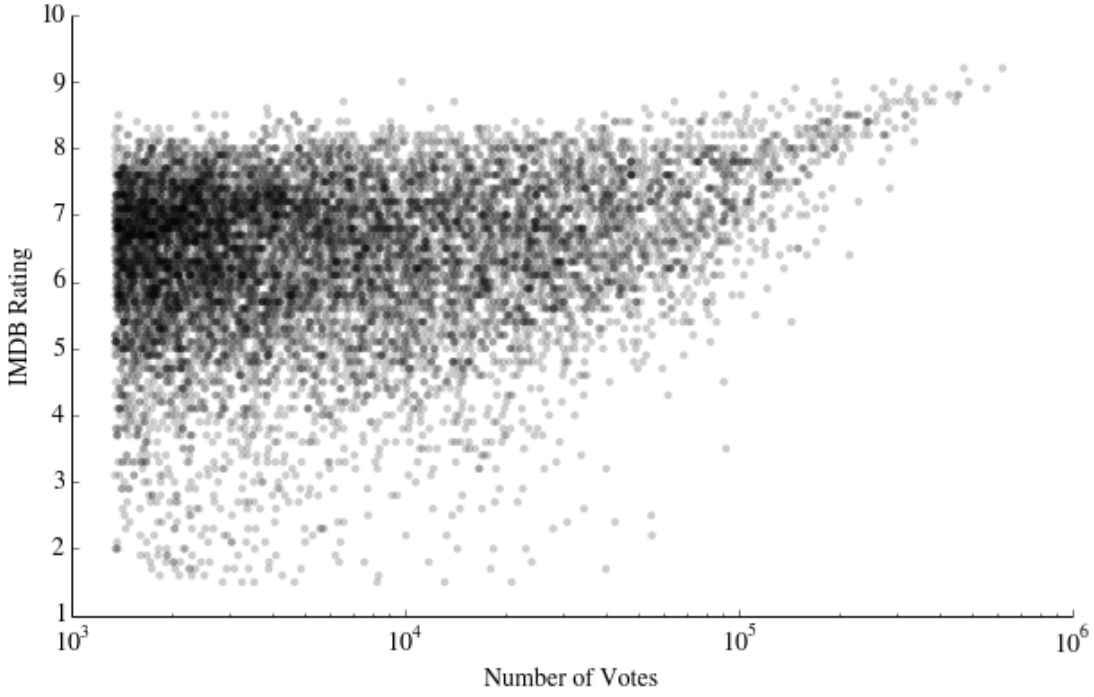
In [55]:
```
plt.hist(data.runtime.dropna(), bins=50, color='#cccccc')
plt.xlabel("Runtime distribution")
remove_border()
```



In [56]:
```
#hmm, more bad, recent movies. Real, or a selection bias?

plt.scatter(data.year, data.score, lw=0, alpha=.08, color='k')
plt.xlabel("Year")
plt.ylabel("IMDB Rating")
remove_border()
```



In [57]:
```
plt.scatter(data.votes, data.score, lw=0, alpha=.2, color='k')
plt.xlabel("Number of Votes")
plt.ylabel("IMDB Rating")
plt.xscale('log')
remove_border()
```



Identify some outliers

In [58]:
```
# low-score movies with lots of votes
data[(data.votes > 9e4) & (data.score < 5)]\
    [['title', 'year', 'score', 'votes', 'genres']]
```

Out [58]:
```
                title  year  score  votes
genres
317         New Moon  2009    4.5  90457
Adventure|Drama|Fantasy|Romance
334  Batman & Robin  1997    3.5  91875      Action|Crime|Fantasy|Sci-
Fi
```

In [59]:
```
# The lowest rated movies
data[data.score == data.score.min()]\
    [['title', 'year', 'score', 'votes', 'genres']]
```

Out [59]:
```
                                title  year  score  votes
genres
1982      Manos: The Hands of Fate  1966    1.5  20927
Horror
2793  Superbabies: Baby Geniuses 2  2004    1.5  13196
Comedy|Family
3746              Daniel the Wizard  2004    1.5   8271
Comedy|Crime|Family|Fantasy|Horror
5158                   Ben & Arthur  2002    1.5   4675
Drama|Romance
5993     Night Train to Mundo Fine  1966    1.5   3542
Action|Adventure|Crime|War
6257                Monster a-Go Go  1965    1.5   3255
Sci-Fi|Horror
6726                     Dream Well  2009    1.5   2848
Comedy|Romance|Sport
```

In [60]:
```
# The highest rated movies
data[data.score == data.score.max()]\
    [['title', 'year', 'score', 'votes', 'genres']]
```

Out [60]:
```
                        title  year  score   votes       genres
0   The Shawshank Redemption  1994    9.2  619479  Crime|Drama
26              The Godfather  1972    9.2  474189  Crime|Drama
```

Another interesing example: let us run aggregation functions like `sum` over several rows or columns... What genres are the most frequent?

In [61]:
```
#sum sums over rows by default
genre_count = np.sort(data[genres].sum())[::-1]
pd.DataFrame({'Genre Count': genre_count})
```

Out [61]:
```
              Genre Count
Drama                5697
Comedy               3922
```

```
Thriller            2832
Romance             2441
Action              1891
Crime               1867
Adventure           1313
Horror              1215
Mystery             1009
Fantasy              916
Sci-Fi               897
Family               754
War                  512
Biography            394
Music                371
History              358
Animation            314
Sport                288
Musical              260
Western              235
Film-Noir             40
Adult                  9
News                   1
Reality-TV             1
```

How many genres does a movie have, on average?

In [62]:
```
#axis=1 sums over columns instead
genre_count = data[genres].sum(axis=1)
print "Average movie has %0.2f genres" % genre_count.mean()
genre_count.describe()
```

```
Average movie has 2.75 genres
```

Out [62]:
```
count    9999.000000
mean        2.753975
std         1.168910
min         1.000000
25%         2.000000
50%         3.000000
75%         3.000000
max         8.000000
dtype: float64
```

8.4 Explore Group Properties

Let's split up movies by decade

In [63]:
```
decade =  (data.year // 10) * 10

tyd = data[['title', 'year']]
tyd['decade'] = decade

tyd.head(10)
```

Out [63]:

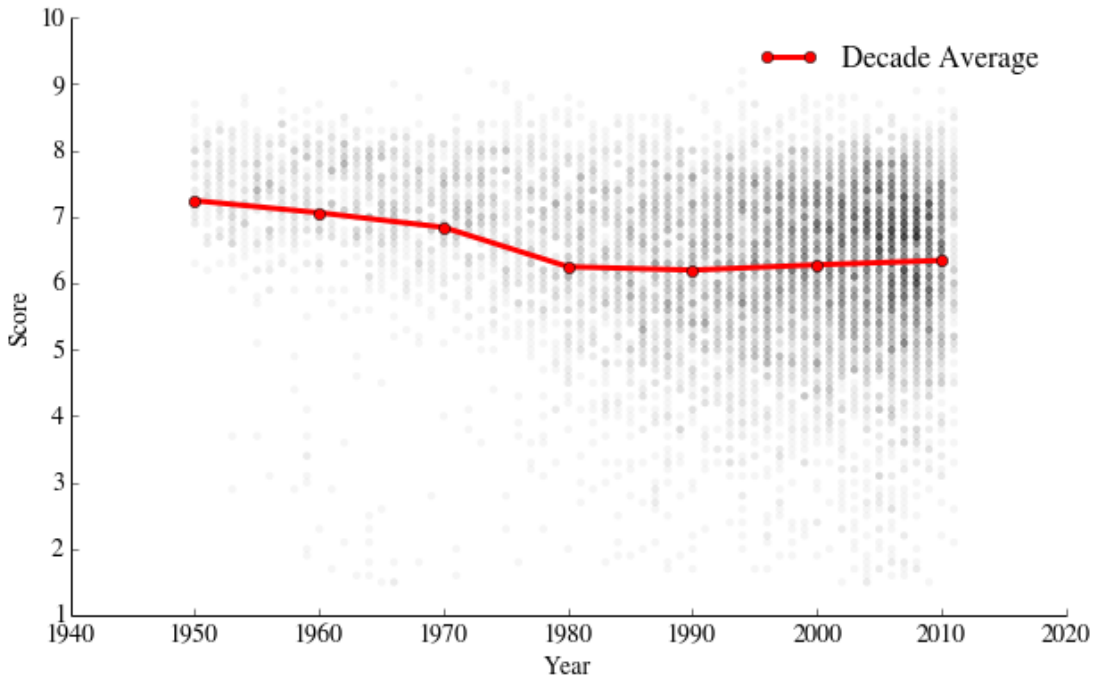
	title	year	decade
0	The Shawshank Redemption	1994	1990
1	Pulp Fiction	1994	1990
2	Fight Club	1999	1990
3	The Matrix	1999	1990
4	Inception	2010	2010
5	Forrest Gump	1994	1990
6	American Beauty	1999	1990
7	Avatar	2009	2000
8	Schindler's List	1993	1990
9	Star Wars: Episode V - The Empire Strikes Back	1980	1980

The first concept we deal with here is pandas `groupby`. The idea is to group a dataframe by the values of a particular factor variable. The documentation can be found here.

GroupBy will gather movies into groups with equal decade values

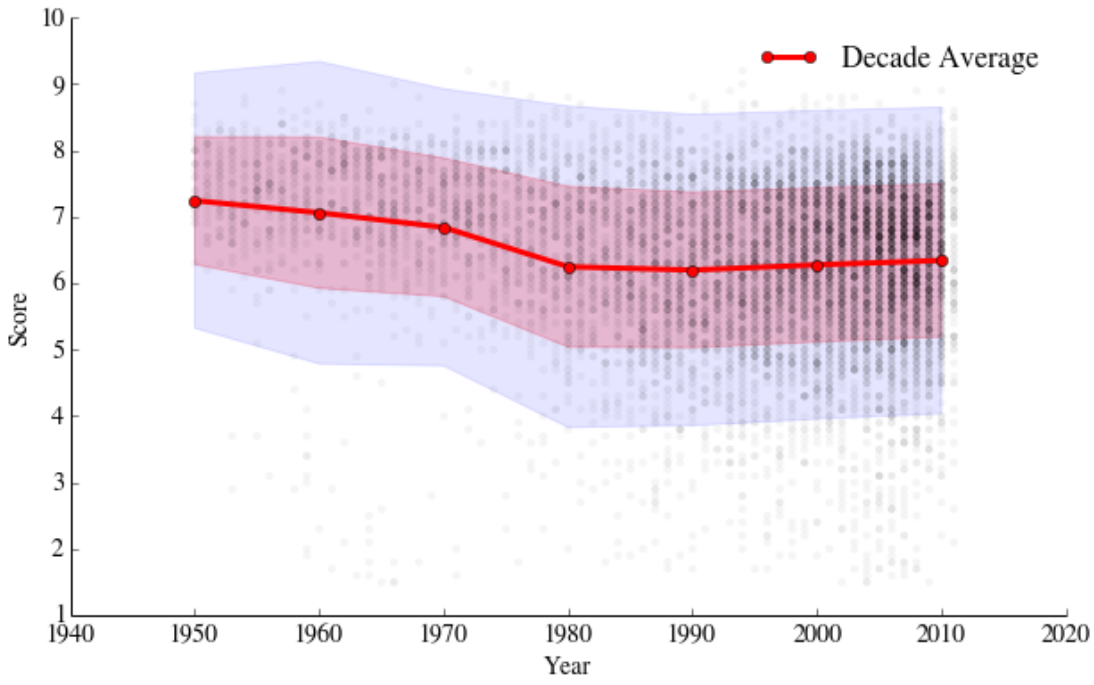
In [64]:
```
#mean score for all movies in each decade
decade_mean = data.groupby(decade).score.mean()
decade_mean.name = 'Decade Mean'
print decade_mean

plt.plot(decade_mean.index, decade_mean.values, 'o-',
        color='r', lw=3, label='Decade Average')
plt.scatter(data.year, data.score, alpha=.04, lw=0, color='k')
plt.xlabel("Year")
plt.ylabel("Score")
plt.legend(frameon=False)
remove_border()
```

```
year
1950    7.244522
1960    7.062367
1970    6.842297
1980    6.248693
1990    6.199316
2000    6.277858
2010    6.344552
Name: Decade Mean, dtype: float64
```



We can go one further, and compute the scatter in each year as well

In [65]:
```
grouped_scores = data.groupby(decade).score

mean = grouped_scores.mean()
std = grouped_scores.std()

plt.plot(decade_mean.index, decade_mean.values, 'o-',
         color='r', lw=3, label='Decade Average')
plt.fill_between(decade_mean.index, (decade_mean + std).values,
                 (decade_mean - std).values, color='r', alpha=.2)
plt.fill_between(decade_mean.index, (decade_mean + 2*std).values,
                 (decade_mean - 2*std).values, color='b', alpha=.1)
plt.scatter(data.year, data.score, alpha=.04, lw=0, color='k')
plt.xlabel("Year")
plt.ylabel("Score")
plt.legend(frameon=False)
remove_border()
```



You can also iterate over a GroupBy object. Each iteration yields two variables: one of the distinct values of the group key, and the subset of the dataframe where the key equals that value. To find the most popular movie each year:

```
In [66]: for year, subset in data.groupby('year'):
             print year, subset[subset.score == subset.score.max()].title.values
```

```
1950 ['Sunset Blvd.']
1951 ['Strangers on a Train']
1952 ["Singin' in the Rain"]
1953 ['The Wages of Fear' 'Tokyo Story']
1954 ['Seven Samurai']
1955 ['Diabolique']
1956 ['The Killing']
1957 ['12 Angry Men']
1958 ['Vertigo']
1959 ['North by Northwest']
1960 ['Psycho']
1961 ['Yojimbo']
1962 ['To Kill a Mockingbird' 'Lawrence of Arabia']
1963 ['The Great Escape' 'High and Low']
1964 ['Dr. Strangelove or: How I Learned to Stop Worrying and Love the
Bomb']
1965 ['For a Few Dollars More']
1966 ['The Good, the Bad and the Ugly']
1967 ['Cool Hand Luke']
1968 ['Once Upon a Time in the West']
1969 ['Butch Cassidy and the Sundance Kid' 'Army of Shadows']
1970 ['Patton' 'The Conformist' 'Le Cercle Rouge']
1971 ['A Clockwork Orange']
1972 ['The Godfather']
1973 ['The Sting' 'Scenes from a Marriage']
```

```
1974 ['The Godfather: Part II']
1975 ['Outrageous Class']
1976 ['Tosun Pasa']
1977 ['Star Wars: Episode IV - A New Hope']
1978 ['The Girl with the Red Scarf']
1979 ['Apocalypse Now']
1980 ['Star Wars: Episode V - The Empire Strikes Back']
1981 ['Raiders of the Lost Ark']
1982 ['The Marathon Family']
1983 ['Star Wars: Episode VI - Return of the Jedi']
1984 ['Balkan Spy']
1985 ['The Broken Landlord']
1986 ['Aliens']
1987 ['Mr. Muhsin']
1988 ['Cinema Paradiso']
1989 ['Indiana Jones and the Last Crusade' "Don't Let Them Shoot the
Kite"]
1990 ['Goodfellas']
1991 ['The Silence of the Lambs']
1992 ['Reservoir Dogs']
1993 ["Schindler's List"]
1994 ['The Shawshank Redemption']
1995 ['The Usual Suspects' 'Se7en']
1996 ['Fargo' 'The Bandit']
1997 ['Life Is Beautiful']
1998 ['American History X']
1999 ['Fight Club']
2000 ['Memento']
2001 ['The Lord of the Rings: The Fellowship of the Ring']
2002 ['City of God']
2003 ['The Lord of the Rings: The Return of the King']
2004 ['Eternal Sunshine of the Spotless Mind']
2005 ['My Father and My Son']
2006 ['The Departed' 'The Lives of Others']
2007 ['Like Stars on Earth']
2008 ['The Dark Knight']
2009 ['Inglourious Basterds']
2010 ['Inception']
2011 ['A Separation']
```

Small multiples

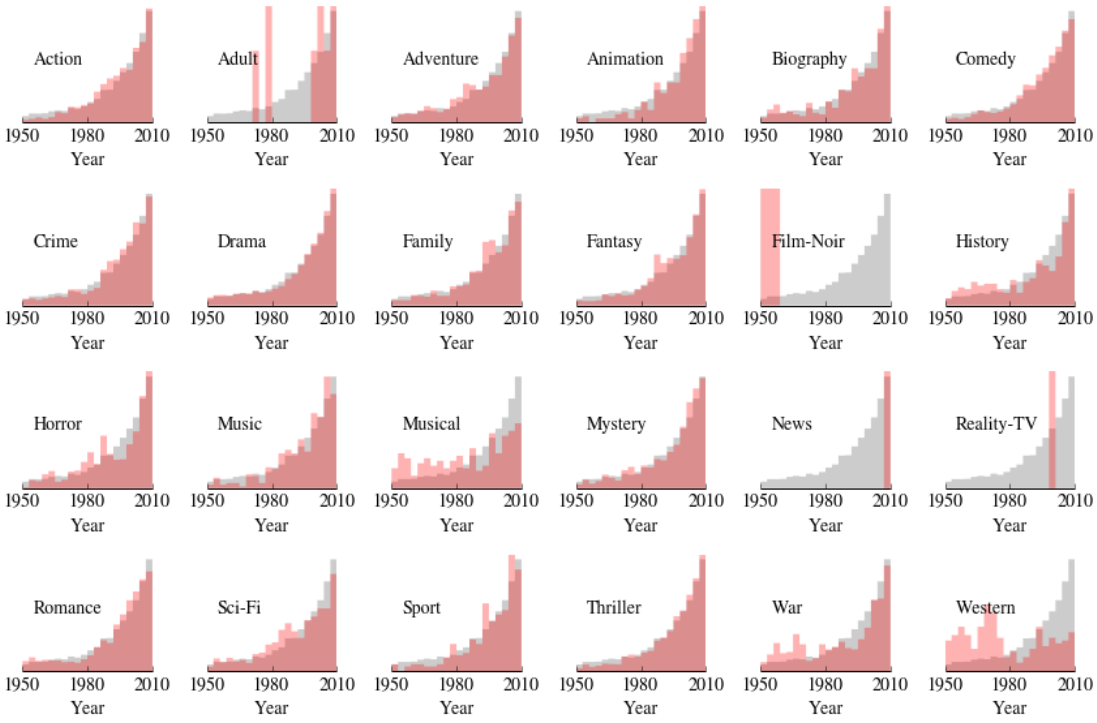
Let's split up the movies by genre, and look at how their release year/runtime/IMDB score vary. The distribution for all movies is shown as a grey background.

This isn't a standard groupby, so we can't use the `groupby` method here. A manual loop is needed

In [67]:
```
#create a 4x6 grid of plots.
fig, axes = plt.subplots(nrows=4, ncols=6, figsize=(12, 8),
                         tight_layout=True)

bins = np.arange(1950, 2013, 3)
for ax, genre in zip(axes.ravel(), genres):

    ax.hist(data[data[genre] == 1].year,
```

```
            bins=bins, histtype='stepfilled', normed=True,
            color='r', alpha=.3, ec='none')

    ax.hist(data.year, bins=bins, histtype='stepfilled',
            ec='None', normed=True, zorder=0, color='#cccccc')

    ax.annotate(genre, xy=(1955, 3e-2), fontsize=14)
    ax.xaxis.set_ticks(np.arange(1950, 2013, 30))
    ax.set_yticks([])
    remove_border(ax, left=False)
    ax.set_xlabel('Year')
```

```
C:\Anaconda\lib\site-packages\matplotlib\figure.py:1595: UserWarning:
This figure includes Axes that are not compatible with tight_layout,
so its results might be incorrect.
  warnings.warn("This figure includes Axes that are not "
```

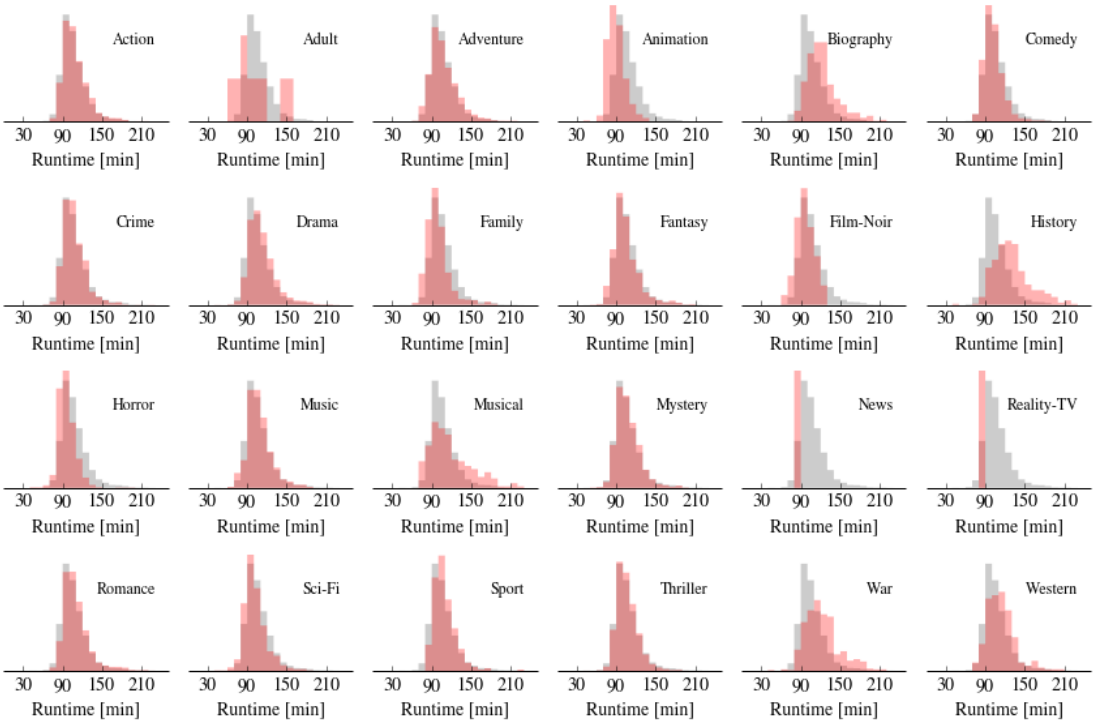


Some subtler patterns here:

1. Westerns and Musicals have a more level distribution
2. Film Noir movies were much more popular in the 50s and 60s

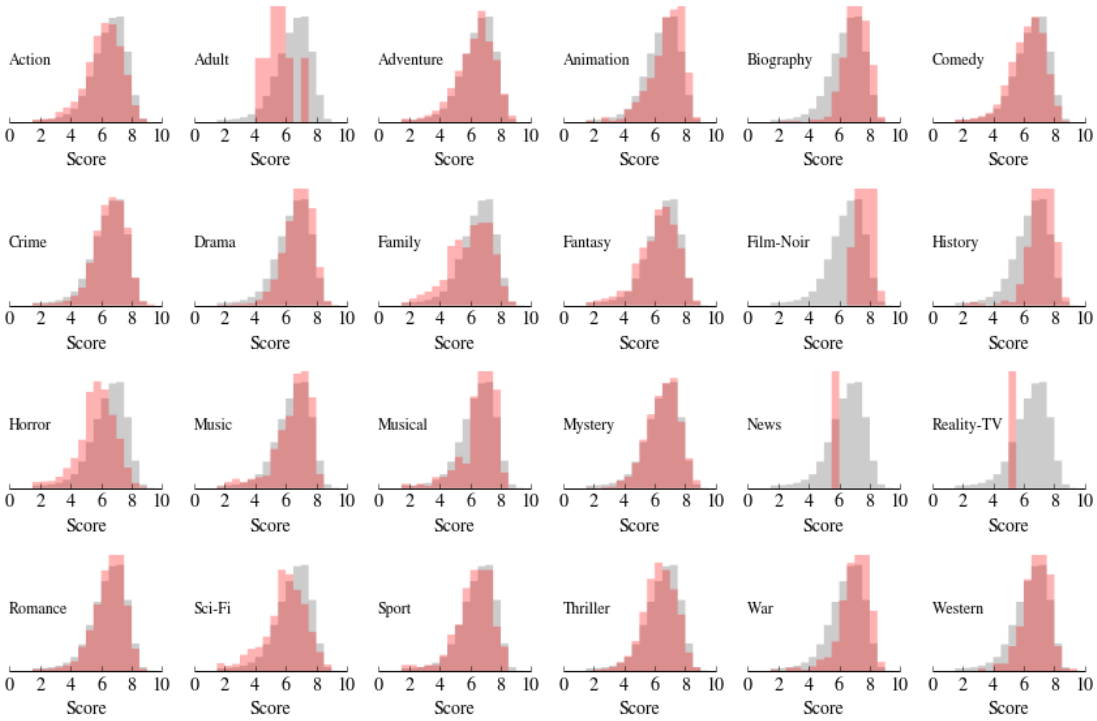
In [68]:
```
fig, axes = plt.subplots(nrows=4, ncols=6, figsize=(12, 8),
                         tight_layout=True)

bins = np.arange(30, 240, 10)

for ax, genre in zip(axes.ravel(), genres):
    ax.hist(data[data[genre] == 1].runtime,
            bins=bins, histtype='stepfilled',
            color='r', ec='none', alpha=.3, normed=True)

    ax.hist(data.runtime, bins=bins, normed=True,
            histtype='stepfilled', ec='none', color='#cccccc',
```

```
            zorder=0)

    ax.set_xticks(np.arange(30, 240, 60))
    ax.set_yticks([])
    ax.set_xlabel("Runtime [min]")
    remove_border(ax, left=False)
    ax.annotate(genre, xy=(230, .02), ha='right', fontsize=12)
```



1. Biographies and history movies are longer
2. Animated movies are shorter
3. Film-Noir movies have the same mean, but are more conentrated around a 100 minute runtime
4. Musicals have the same mean, but greater dispersion in runtimes

In [69]:
```
fig, axes = plt.subplots(nrows=4, ncols=6, figsize=(12, 8),
                         tight_layout=True)

bins = np.arange(0, 10, .5)

for ax, genre in zip(axes.ravel(), genres):
    ax.hist(data[data[genre] == 1].score,
            bins=bins, histtype='stepfilled',
            color='r', ec='none', alpha=.3, normed=True)

    ax.hist(data.score, bins=bins, normed=True,
            histtype='stepfilled', ec='none', color='#cccccc',
            zorder=0)

    ax.set_yticks([])
    ax.set_xlabel("Score")
    remove_border(ax, left=False)
    ax.set_ylim(0, .4)
    ax.annotate(genre, xy=(0, .2), ha='left', fontsize=12)
```



1. Film-noirs, histories, and biographies have higher ratings (a selection effect?)
2. Horror movies and adult films have lower ratings

CHAPTER

NINE

OPTIMIZATION IN PYTHON

9.1 Something about Optimization

Operational Research and Optimization is a field directly concerned with algorithms that compute a numerical solution for problems that can be formulated as follows:

$$\begin{array}{ll} \min_x & f(x) \\ \text{s.t.} & c(x) \geq 0 \end{array}$$

where $f(x)$ is a real function $f : \mathbb{R}^n \to \mathbb{R}$, while $c(x)$ can be a multivalued function $c : \mathbb{R}^n \to \mathbb{R}^m$ or not exist. We refer to f as the *objective function* and to c as the *constraint function(s)*. We are asked to obtain a point x^* that is a local minimizer for f.

It should be emphasised that we are looking for numerical solutions, as opposed to analytical solutions obtained by symbolic computation procedures, for example.

Operational Research and Optimization is the field directly concerned with providing the answers we wish to obtain; this field is divided into several areas, differentiated by the techniques used to compute the solutions. The main areas are:

- **Global optimization**: We wish to obtain the value that is the global minimum of our objective function, or perhaps a large number of local minimizers, with values close to that of the global minimum. A **global minimizer** is a point x^*, satisfying $c(x^*) \geq 0$, such that $f(x^*) \leq f(x)$ with $\forall x \in \{y : c(y) \geq 0\}$, while a **local minimizer** satisfies for some $\epsilon > 0$, $f(x^*) \leq f(x)$ with $\forall x \in \{y : c(y) \geq 0, ||y - x^*|| < \epsilon\}$
- **Stochastic programming**: Sometimes the data needed to formulate the problem given above (even if the model is well defined) is not known with certainty, but rather we only have a (total or partial) knowledge coming from their statistical or probabilistic distribution.
- **Integer programming**: Some of the variables may be limited to taking values in a discrete set.
- **Nonlinear programming**: The functions in the model are continuous, and we are only interested in finding a local solution.
- **Linear programming**: The functions in the problem are linear.

We will concern ourselves only with the last two cases, and so we will assume that the functions defining our problem are continuous, all parameters are known with certainty and it is enough to compute one local minimizer. As we mentioned, the solutions to the problems here described do not in general have a closed-form representation (a formula whose application yields the desired values for the variables). Furthermore, with few exceptions (linear and quadratic programming, for example) there is no finite procedure (a procedure giving the desired answer after a finite number of iterations) for computing a solution to our problems.

To solve any of these problems, we need to use an iterative procedure, that is, we compute a sequence of points x_k having the property that

$$x_k \to x^*$$

where x^* is the solution to out problem.

The $k+1$-st iteration of the algorithm would consist in obtaining a direction of movement p_k, having the property that the point $x_k + p_k$ is, on some measure, a better point than x_k. The value of p_k should correspond to a reasonable direction to follow in order to improve the last point, but its size may not be a good indication of the step to take along that direction. In these cases, the general procedure includes the computation of a step along the direction p_k, α_k, yielding an improved point x_{k+1}, computed as

$$x_{k+1} = x_k + \alpha_k p_k$$

In general, p_k will be a descent direction, that is, we will be able to decrease either f or some merit function M if we take a sufficiently small step from x_k along p_k. The value of p_k is normally obtained from a local model of the problem at x_k, that is, a simplified problem (easier to solve) that is similar to our problem for points that are sufficiently close to x_k. The value of α_k is added to correct for the discrepancies of the local model when we have to move away from x_k (to ensure convergence).

Therefore, the optimization model algorithm takes the form:

`repeat`

`Compute` p_k

`Compute` α_k

$x_{k+1} \leftarrow x_k + \alpha_k p_k$

`until converged`

9.2 Python packages for Optimization

There is multiple packages in Python in the field of optimization and operations research.

- **APM Python** - APM Python is free optimization software through a web service. Nonlinear Programming problem are sent to the APMonitor server and results are returned to the local Python script. A web-interface automatically loads to help visualize solutions, in particular dynamic optimization problems that include differential and algebraic equations. Default solvers include APOPT, BPOPT, and IPOPT. Pre-configured modes include optimization, parameter estimation, dynamic simulation, and nonlinear control.
- **Coopr** The Coopr software project integrates a variety of Python optimization-related packages.
- **CVOXPT** is a free software package for convex optimization based on the Python programming language. It can be used with the interactive Python interpreter, on the command line by executing Python scripts, or integrated in other software via Python extension modules. Its main purpose is to make the development of software for convex optimization applications straightforward by building on Python's extensive standard library and on the strengths of Python as a high-level programming language.
- **OpenOpt** contains connections to tens of solvers and has some own Python-written ones, e.g. nonlinear solver with specifiable accuracy: interalg, graphic output of convergence and some more numerical optimization features. Also OpenOpt can solve FuncDesigner problems with automatic differentiation, that usually work faster and gives more precise results than finite-differences derivatives approximation.

- **PuLP** - PuLP is an LP modeler written in python. PuLP can generate MPS or LP files and call GLPK, COIN CLP/CBC, CPLEX, and GUROBI to solve linear problems.
- **Pyomo** - The Python Optimization Modeling Objects (Pyomo) package is an open source tool for modeling optimization applications in Python. Pyomo can be used to define symbolic problems, create concrete problem instances, and solve these instances with standard solvers. Pyomo provides a capability that is commonly associated with algebraic modeling languages such as AMPL, AIMMS, and GAMS, but Pyomo's modeling objects are embedded within a full-featured high-level programming language with a rich set of supporting libraries. Pyomo leverages the capabilities of the Coopr software library, which integrates Python packages for defining optimizers, modeling optimization applications, and managing computational experiments.
- **scipy.optimize** - some solvers written or connected by SciPy developers.
- **pyOpt** - pyOpt is a package for formulating and solving nonlinear constrained optimization problems in an efficient, reusable and portable manner (license: LGPL).

Here we will briefly review two of these packages: **Scipy.Optimize** for non-linear optimization and **CVXOPT** for convex optimization.

9.3 Scipy.Optimize

In [3]:
```
import numpy as np
import scipy.optimize
```

The **Scipy.Optimize** package provides several commonly used optimization algorithms. A detailed listing is available: scipy.optimize? (can also be found by help(scipy.optimize)). The module contains:

- **Unconstrained** and **constrained minimization** of multivariate scalar functions (minimize) using a variety of algorithms (e.g. BFGS, Nelder-Mead simplex, Newton Conjugate Gradient, COBYLA or SLSQP)
- **Global (brute-force) optimization** routines (e.g., anneal, basinhopping)
- **Least-squares minimization** (leastsq) and **curve fitting** (curve_fit) algorithms
- **Scalar univariate functions minimizers** (minimize_scalar) and root finders (newton)
- **Multivariate equation system solvers** (root) using a variety of algorithms (e.g. hybrid Powell, Levenberg-Marquardt or large-scale methods such as Newton-Krylov)

Below, we show several examples demonstrate their basic usage

In [5]:
```
# Documentation to scipy.optimize in IPython
scipy.optimize?
```

9.4 Unsconstrained problems with Scipy.Optimize

We start the study of algorithms for the solution of the nonlinear problem with the case in which we do not have constraints on the value of the variables. In this case, we are given a function $f : \mathbb{R}^n \to \mathbb{R}$, and we are asked to obtain a point x^* that is a local minimizer for f.

As we mentioned in the previous section, the standard procedure in this case is to obtain an improvement for the best point computed from the analysis of a local approximation of f, constructed from its derivatives at the point. The following notation will be used throughout the rest of this chapter:

- The **gradient vector** of f at x, $\nabla f(x)$, will be denoted by $g(x)$ or g. We will take it to be a column vector

$$g(x) = \begin{pmatrix} \frac{\partial f(x)}{\partial x_1} \\ \vdots \\ \frac{\partial f(x)}{\partial x_n} \end{pmatrix}$$

- The **Hessian matrix** of f at x, will be denoted by $H(x)$ or H. It is a symmetric matrix (at least for twice-continuously differentiable functions)

$$H(x) = \begin{pmatrix} \frac{\partial^2 f(x)}{\partial x_1^2} & \cdots & \frac{\partial^2 f(x)}{\partial x_1 \partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f(x)}{\partial x_n \partial x_1} & \cdots & \frac{\partial^2 f(x)}{\partial x_n^2} \end{pmatrix}$$

  Although it is advantageous to use as much derivative information as is available (at least up to second derivatives) in our algorithms, there may be cases in which either the function f is not differentiable on a significant set of points, or the derivatives are not available for some reason. In these cases we have to limit ourselves to using information exclusively about function values, as in the algorithm presented in the preceding section.It is important to emphasize that in many of the cases where derivatives are not available at all points, we still have information about special structure in our function (convex functions, or Lipschitz continuous functions), that can be exploited to generate more efficient algorithms (see Fletcher *"Practical Methods for Optimization"*, chap. 14). From now on we assume that our function is twice-continuously differentiable.

The minimize function provides a common interface to unconstrained and constrained minimization algorithms for multivariate scalar functions in scipy.optimize. To demonstrate the minimization function consider the problem of minimizing the **Rosenbrock function** of N variables:

$$f(\mathbf{x}) = \sum_{i=1}^{N-1} 100 \left(x_i - x_{i-1}^2\right)^2 + (1 - x_{i-1})^2$$

The minimum value of this function is 0 which is achieved when $x_i = 1$.

Note that the Rosenbrock function and its derivatives are included in **scipy.optimize**. The implementations shown in the following sections provide examples of how to define an objective function as well as its jacobian and hessian functions.

9.4.1 Nelder-Mead Simplex algorithm

We now present an example of an algorithm that follows the lines indicated at the beginning of the chapter, although it is not based on a local approximation, being based solely on function comparisons. We will see later on that this will make the algorithm very inefficient, but there may be cases when there is no other option available.

The **Nelder–Mead method** or **Simplex method** is a commonly used nonlinear optimization technique, which is a well-defined numerical method for problems for which derivatives may not be known. However, the Nelder–Mead technique is a heuristic search method that can converge to non-stationary points on problems that can be solved by alternative methods.

The algorithm starts with $n+1$ points $x_0, x_1, \ldots, x_n$ (supposed to be an approximation in some sense to the solution of the problem). We will assume that they are ranked by their objective function values, so that $f_n > f_{n-1} > \ > f_1 > f_0$.

In any iteration we keep the $n+1$ best points (assume the same notation as above), ranked according to their function values. To determine the search direction p_n and the step-length α_n, we take the following steps:

1. Compute c (the center of gravity of the best n points) as

$$c = \frac{1}{n}\sum_{i=0}^{n-1} x_i$$

and define the search direction as

$$p_n = c - x_n$$

2. Determine α_n as follows:

(a). If $f_0 \leq f(x_n + (1+\beta)p_n) \leq f_{n-1}$, compute $x_{n+1} = x_n + (1+\beta)p_n$ where $\beta > 0$ (the reflection coefficient) is a parameter whose value is fixed in advance.

(b). If $f(x_n + (1+\beta)p_n) < f_0$ we have found the best point so far, and it may be worth continuing the search by trying a larger value for α_n. For $\gamma > \beta$ compute

$$x_{n+1} = \begin{cases} x_{n+1} = x_n + (1+\beta)p_n & \text{if} f(x_n + (1+\beta)p_n) < f(x_n + (1+\gamma)p_n), \\ x_{n+1} = x_n + (1+\gamma)p_n & \text{otherwise} \end{cases}$$

(c). If $f_0 \leq f(x_n + (1+\beta)p_n) > f_{n-1}$ we have moved too far along p_n. We take a shorter step and select

$$x_{n+1} = \begin{cases} x_{n+1} = x_n + (1+\delta)p_n & \text{if} f(x_n + (1+\beta)p_n) < f_n, \\ x_{n+1} = x_n + (1-\delta)p_n & \text{otherwise} \end{cases}$$

Again, $\delta < \beta$ is another parameter to be fixed in the algorithm.

3. Replace x_n with x_{n+1} and relabel the $n+1$ points. Repeat the process until sufficiently close to the solution (until the size of the polytope is sufficiently small).

Just emphasize that this sort of algorithms should only be used when all else has failed. Basically, **Nelder–Mead** generates a new test position by extrapolating the behaviour of the objective function measured at each test point arranged as a simplex. The algorithm then chooses to replace one of these test points with the new test point and so the technique progresses. The simplest step is to replace the worst point with a point reflected through the centroid of the remaining N points. If this point is better than the best current point, then we can try stretching exponentially out along this line. On the other hand, if this new point isn't much better than the previous value, then we are stepping across a valley, so we shrink the simplex towards a better point.

In the example below, the minimize routine is used with the **Nelder-Mead simplex algorithm** (selected through the method parameter):

In [1]:
```
def rosen(x):
    """The Rosenbrock function"""
    return sum(100.0*(x[1:]-x[:-1]**2.0)**2.0 + (1-x[:-1])**2.0)
```

In [4]:
```
from scipy.optimize import minimize
x0 = np.array([1.3, 0.7, 0.8, 1.9, 1.2])
res = minimize(rosen, x0, method='nelder-mead',
               options={'xtol': 1e-8, 'disp': True})
```

```
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 339
         Function evaluations: 571
```

```
In [5]: print(res.x)
```

```
[ 1.  1.  1.  1.  1.]
```

The simplex algorithm is probably the simplest way to minimize a fairly well-behaved function. It requires only function evaluations and is a good choice for simple minimization problems. However, because it does not use any gradient evaluations, it may take longer to find the minimum.

Another optimization algorithm that needs only function calls to find the minimum is Powell's method available by setting `method='powell'` in minimize.

9.4.2 Optimality conditions for unconstraint problems

Before we gave a characterization of a local minimizer as a point x^* satisfying $f(x^*) \leq f(x)\ \forall x\ y : ||y - x^*|| < \epsilon$, for some $\epsilon > 0$.

This is not a very useful definition, as in order to check that a point is a minimizer from this condition, we would need to compute the value of f at an infinite number of points near x^*. Fortunately, for local solutions there are better characterisations, requiring information only about the function and its derivatives at x^*.

Necessary Conditions

If x^* is a local minimizer of f, then

1. $g(x^*) = 0$, and
2. $H(x^*)$ is positive semidefinite

Sufficient Conditions

If the following conditions hold at a point x^*,

1. $g(x^*) = 0$, and
2. $H(x^*)$ is positive definite,

then x^* is a local minimizer for f

The proof of both conditions are based in the Taylor series expansion around the point x^* and can be checked in any basic optimization book.

In order to compute points that satisfy the optimality conditions, we need to have available the first (and perhaps the second) derivatives of the objective function at the different points generated by the algorithm.

In many cases we would be able to obtain these derivatives by differentiating the objective function, either manually or with the help of some symbolic computation package. This situation is obviously the most favourable one.

Unfortunately, it is often the case that we do not have an expression for the objective function that we can differentiate (for example, consider the situation where f is one of the eigenvectors of a general matrix, whose coefficients are functions of the variables). This means that we cannot obtain formulas for the derivatives, but we can still try to obtain approximations to their values by numerical procedures.

- **Forward-difference formula**: From the definition of the partial derivatives of f as limits

$$\frac{\partial f(x)}{\partial x_i} = \lim_{h \to 0} \frac{f(x + he_i) - f(x)}{h}$$

where e_i denotes the i-th unit vector, we can construct an approximation as

$$\frac{\partial f(x)}{\partial x_i} \approx \nabla_i f(x) = \frac{f(x + he_i) - f(x)}{h}$$

for some for some sufficiently small $h > 0$.

Of course, $\nabla_i f$ is not equal to the derivative, and that is so because of two reasons:

1. **Truncation errors**, due to neglected terms in the Taylor series expansions, the truncation error is

$$\nabla_i f(x) = \frac{\partial f(x)}{\partial x_i} - \frac{1}{2}h\frac{\partial^2 f(\xi)}{\partial x_i^2}$$

2. **Cancellation errors**. The values of $f(x)$ and $f(x + he_i)$ will be very close if h is small, and we will have cancellation in finite arithmetic when computing their difference.

- **Central-difference formula** The previous formula has the advantage that only $n + 1$ evaluations of f are required to compute the gradient, but it also has the disadvantage that near the solution of the problem we lose accuracy due to the fact that we are working with values of the derivative that are becoming close to zero (cancellation). An alternative formula for approximating the partial derivatives is

$$\frac{\partial f(x)}{\partial x_i} \approx \delta_i f(x) = \frac{f(x + he_i) - f(x - he_i)}{2h}$$

having the advantage that the truncation error is now given by

$$\delta_i f(x) = \frac{\partial f(x)}{\partial x_i} - \frac{1}{6}h^2\frac{\partial^3 f(\xi)}{\partial x_i^3}$$

and for small values of h this value is smaller than the one for the previous case. The disadvantage of this approach is that it requires $2n$ function evaluations to compute the gradient. It is advisable to use the first formula ($\nabla_i f$) until we detect a significant loss of precision due to cancellation, and then switch to the second formula ($\delta_i f$); as we should then be near the solution, the cost of the additional function evaluations should not be too important.

- **Second-order differences**: If we need to approximate the elements of the Hessian matrix, the formula used most frequently is

$$\delta_{ij} f(x) = \frac{f(x + he_i + he_j) - f(x + he_i - he_j) - f(x - he_i + he_j) + f(x - he_i - he_j)}{4h^2}$$

9.4.3 Newton-Conjugate-Gradient algorithm

Consider the general algorithm presented in the preceding section. We start by selecting reasonable values for the search direction p_k. In order to do that, we study an approximation to our problem. The natural one comes from the Taylor expansion around x_k,

$$f(x_k + p_k) = f(x_k) + g(x_k)^T p_k + \frac{1}{2}p_k^T H(x_k)p_k + \dots$$

We are interested in minimizing the left-hand side, and to do that we may try to minimize a certain number of terms from the right-hand side of the equality. The simplest non-trivial choice consists in keeping the first two terms, giving

$$f(x_k + p_k) \approx f(x_k) + g(x_k)^T p_k + \frac{1}{2} p_k^T H(x_k) p_k$$

and in order to find a minimizer for f, we now minimize the right-hand side,

$$min_p g_k^T p + \frac{1}{2} p^T H_k p$$

To find a solution for this problem we impose the condition

$$\nabla_p (g(x_k)^T p_k + \frac{1}{2} p_k^T H(x_k) p_k) = 0$$

giving a point p_k satisfying

$$H_k p_k = -g_k$$

This point will be a solution of the problem if H_k is positive definite; otherwise, if H_k is indefinite we will have a maximum or a saddle point, implying that the value of p_k computed in the manner described above may not be a very good approximation for a step to a minimizer of f. Another complication (related to the one discussed before) in the case when H_k is indefinite is due to the descent properties of p_k. Remember that we are interested in computing a point x_{k+1} that is "better" than x_k, meaning that it has a lower value of the objective function.

In many practical situations we may not be able to compute explicitly the second derivatives of our objective function (in fact, we may not be able to compute even the first derivatives analytically). In these cases we need to generate some approximation to the second derivatives in order to use Newton-type methods.

One alternative (already discussed) is to approximate the elements of the Hessian by finite differences. Concerning the choice of h, the theoretical results indicate that the quadratic convergence rate will be preserved if h goes to zero as $||g_k||$ does. The main limitation of discrete Newton methods is due to the number of function evaluations needed to compute the Hessian matrix; in general we need on the order of n^2 function evaluations (or the equivalent to n extra evaluations of the gradient), making the cost prohibitive if the values of the function are not cheap to compute. Of course, if the matrix is sparse and has special structure, the total number of function evaluations required may be significantly reduced.

The method which requires the fewest function calls and is therefore often the fastest method to minimize functions of many variables uses the **Newton-Conjugate Gradient algorithm**. This method is a modified Newton's method and uses a conjugate gradient algorithm to (approximately) invert the local Hessian. Notice that if the Hessian is positive definite then the local minimum of this function can be found by setting the gradient of the quadratic form to zero, resulting in

$$\mathbf{x}_{\text{opt}} = \mathbf{x}_0 - \mathbf{H}^{-1} \nabla f$$

The inverse of the Hessian is evaluated using the conjugate-gradient method. Conjugate gradient methods were introduced in 1952 for the solution of large systems of linear equations with positive definite coefficient matrix. With no rounding errors the method would require at most n iterations to find the solution, but due to rounding errors the number of iterations may be much larger.

In 1963 Fletcher and Reeves suggested that the method could be used for optimization. However, in practice the method has been shown not to be competitive with Newton-type methods, except in those cases where it is impossible to perform solves with the Hessian matrix (large non-sparse problems).

In summary, the application of these methods has been restricted to either the solution of very large, not very sparse, optimization problems, or to the solution of large linear systems of equations as part of an optimization routine, specially when it is sufficient to obtain an approximate solution.

An example of employing this method to minimizing the Rosenbrock function is given below. To take full advantage of the Newton-CG method, a function which computes the Hessian must be provided. The Hessian matrix itself does not need to be constructed, only a vector which is the product of the Hessian with an arbitrary vector needs to be available to the minimization routine. As a result, the user can provide either a function to compute the Hessian matrix, or a function to compute the product of the Hessian with an arbitrary vector.

Here we show a full hessian example: The Hessian of the Rosenbrock function is

$$\begin{aligned} H_{ij} &= \frac{\partial^2 f}{\partial x_i \partial x_j} \\ &= 200\left(\delta_{i,j} - 2x_{i-1}\delta_{i-1,j}\right) - 400x_i\left(\delta_{i+1,j} - 2x_i\delta_{i,j}\right) - 400\delta_{i,j}\left(x_{i+1} - x_i^2\right) + 2\delta_{i,j}, \\ &= \left(202 + 1200x_i^2 - 400x_{i+1}\right)\delta_{i,j} - 400x_i\delta_{i+1,j} - 400x_{i-1}\delta_{i-1,j}, \end{aligned}$$

if $i, j \in [1, N-2]$ with $i, j \in [0, N-1]$ defining the $N \times N$ matrix. Other non-zero entries of the matrix are

$$\begin{aligned} \frac{\partial^2 f}{\partial x_0^2} &= 1200x_0^2 - 400x_1 + 2, \\ \frac{\partial^2 f}{\partial x_0 \partial x_1} &= \frac{\partial^2 f}{\partial x_1 \partial x_0} = -400x_0, \\ \frac{\partial^2 f}{\partial x_{N-1} \partial x_{N-2}} &= \frac{\partial^2 f}{\partial x_{N-2} \partial x_{N-1}} = -400x_{N-2}, \\ \frac{\partial^2 f}{\partial x_{N-1}^2} &= 200. \end{aligned}$$

For example, the Hessian when $N = 5$ is

$$\mathbf{H} = \begin{bmatrix} 1200x_0^2 - 400x_1 + 2 & -400x_0 & 0 & 0 & 0 \\ -400x_0 & 202 + 1200x_1^2 - 400x_2 & -400x_1 & 0 & 0 \\ 0 & -400x_1 & 202 + 1200x_2^2 - 400x_3 & -400x_2 & 0 \\ 0 & & -400x_2 & 202 + 1200x_3^2 - 400x_4 & -400x_3 \\ 0 & 0 & 0 & -400x_3 & 200 \end{bmatrix}$$

The code which computes this Hessian along with the code to minimize the function using Newton-CG method is shown in the following example:

In [20]:
```
def rosen_hess(x):
    x = np.asarray(x)
    H = np.diag(-400*x[:-1],1) - np.diag(400*x[:-1],-1)
    diagonal = np.zeros_like(x)
    diagonal[0] = 1200*x[0]**2-400*x[1]+2
    diagonal[-1] = 200
    diagonal[1:-1] = 202 + 1200*x[1:-1]**2 - 400*x[2:]
    H = H + np.diag(diagonal)
    return H
```

In [22]:
```
res = minimize(rosen, x0, method='Newton-CG',
               jac=rosen_der, hess=rosen_hess,
               options={'avextol': 1e-8, 'disp': True})
```

```
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 20
         Function evaluations: 23
```

```
                Gradient evaluations: 20
                Hessian evaluations: 20
```

```
In [23]: print(res.x)
```

```
         [ 0.99999352  0.99998455  0.99996844  0.99993646  0.99987251]
```

9.4.4 Broyden-Fletcher-Goldfarb-Shanno method or BFGS-algorithm

In **Quasi-Newton methods**, the Hessian matrix of second derivatives doesn't need to be evaluated directly. The idea behind **Quasi-Newton methods** is to try to use the values of the gradient computed along the solution process, $g_1, \ldots, g_k, \ldots$, to construct an approximation to the Hessian matrix, without requiring any additional gradient estimation. In order to explain the **Broyden–Fletcher–Goldfarletb–Shanno (BFGS) algorithm** we start considering the univariate Newton method where

$$x_{k+1} = x_k - \frac{f'_k}{f''_k}$$

and, assuming that we cannot compute f''_k , we can approximate it using

$$f'_{k-1} \approx f'_k + f''_k(x_{k-1} - x_k)$$

to obtain

$$x_{k+1} = x_k - \frac{f'_k}{\frac{f'_{k-1} - f'_k}{x_{k-1} - x_k}}$$

This procedure obtains the step from a second-order model of the problem in which the Hessian at x_k has been approximated by

$$f''_k \approx \frac{f'_{k-1} - f'_k}{x_{k-1} - x_k}$$

Analogously, the theory of **Quasi-Newton methods** is based on the fact that an approximation to the curvature of a nonlinear function along a line can be computed from two values of the gradient on the line, without explicitly forming the Hessian matrix. Let s_k be the step taken from x_k, $s_k \equiv \alpha_k p_k$ then

$$g(x_k + s_k) = g(x_k) + H_k s_k + \cdots$$

and we could compute an approximation to the Hessian, B_k, from

$$B_k s_k = g_{k+1} - g_k$$

The main difference with the univariate case is that now we have $n(n+1)/2$ unknowns and n equations, and we cannot determine all the elements of B_k from this relationship.

This condition can be related to the secant method by noting that if $\phi(\alpha) = f(x_k + \alpha p_k)$, then

$$\phi''(\alpha_k) = g(x_k + \alpha_k p_k)^T p_k \approx \phi'(0) + \alpha_k \phi''(0) = g_k^T p_k + \alpha_k p_k^T H_k p_k$$

or

$$\phi''(0) \approx \frac{1}{\alpha_k}(\phi'(\alpha_k) - \phi'(0))$$

Or in other words, we approximate the curvature of f (the Hessian) along p_k by using the directional derivatives at x_k and x_{k+1}, to get a reasonable estimate along that direction, but this procedure does not give us any information along other directions.

Another important property of this approximation is that for quadratic functions $f(x) = \frac{1}{2}x^T Hx + c^T x$ the condition

$$Hs_k = g_{k+1} - g_k$$

is satisfied exactly.

Assume that at the start of iteration k of a quasi-Newton algorithm we have a matrix B_k which is an approximation to the Hessian matrix H_k; then, as in the derivation of Newton's method, B_k is taken as the Hessian matrix of the quadratic model function

$$f(x_k + p) \approx f_k + g_k^T p + \frac{1}{2}p^T B_k p$$

and the search direction is obtained as the solution of the linear system

$$B_k p_k = -g_k$$

The initial Hessian approximation B_0 is usually taken to be the identity matrix. With this choice, the initial direction is the steepest descent direction $p_0 = -g_0$.

After x_{k+1} has been computed, a new Hessian approximation B_{k+1} is obtained to take into account the new curvature information. Given that we are obtaining information about the Hessian only along one direction, we should not discard all previous information; what we do is to update it using the formula

$$B_{k+1} = B_k + U_k$$

where U_k is the updating matrix at iteration k. During any single iteration we only obtain new information about f in the direction of s_k. It would be reasonable to have B_{k+1} differing from B_k by a matrix of low rank (that is, B_{k+1} should not be very different from B_k). An obvious idea is to modify B_k by a rank-one matrix to satisfy the quasi-Newton condition

$$B_{k+1}s_k = y_k$$

at the new point. Let

$$B_{k+1} = B_k + uv^T$$

for some vectors u and v (the matrix uv^T is of rank one). These vectors should satisfy the quasi-Newton condition, so we should have

$$B_{k+1}s_k = (B_k + uv^T)s_k = y_k \Rightarrow v^T s_k u = y_k - B_k s_k$$

Therefore, the vector u must be parallel to $y_k - B_k s_k$, and u becomes proportional to the error in the quasi-Newton condition using the last approximation B_k.

Note that v only enters the formula as a scale factor, so we are free to select any v (such that $v^T s_k = 0$), and the rank one modification from B_k to B_{k+1} is given by

$$B_{k+1} = B_k + \frac{1}{v^T s_k}(y_k - B_k s_k)v^T$$

If v is chosen to be equal to s_k, the update reduces to the form

$$B_{k+1} = B_k + \frac{1}{||s_k||^2}(y_k - B_k s_k)s_k^T$$

and this formula is known as **Broyden's update** which is the solution of the minimum norm problem

$$\begin{array}{ll} \min & ||U_k||_F \\ s.t. & (B_k + U_k)s_k = y_k \end{array}$$

where $||U_k||_F$ denotes the Frobenius norm, $||U_k||_F^2 = \sum_{i,j} u_{ij}^2$

The **Broyden's update formula** does not possess two important properties: **symmetry** and **positive definiteness**.

1. Symmetry condition on BFGS

Since the Hessian matrix is symmetric, we would like to have an update that has the property of hereditary symmetry, that is, if B_k is symmetric, then B_{k+1} should be symmetric. In particular, to preserve symmetry in the rank-one update we must have u = v. We then get the formula

$$B_{k+1} = B_k + \frac{1}{(y_k - B_k s_k)^T s_k}(y_k - B_k s_k)(y_k - B_k s_k)^T$$

where we require $y_k - B_k s_k \neq 0$ and $(y_k - B_k s_k)^T s_k \neq 0$. This update is known as the **symetric rank-one update** $(SR1)$

2. Positive Definite condition on BFGS

On the other hand, since we use the Hessian approximation to obtain the search direction, and we need it to be a descent direction, it seems reasonable to require that every Hessian approximation be **positive definite** as well as symmetric. If B_k is a positive definite matrix, the resulting search direction will be a descent direction, and the step length along p_k can be chosen based on one of the line-search termination criteria mentioned before.

We have already seen that the quasi-Newton condition forces the underlying model function to have curvature $y_k^T s_k$ along s_k at x_k (recall that $y_k^T s_k = s_k^T B_{k+1} s_k$). Therefore, if $B_{k+1}s_k$. Therefore, if B_{k+1} is to be positive definite, we would want $y_k^T s_k$ to be positive. Note that anystep lenght α_k satisfying $|g_{k+1}p_k| \leq -\eta g_k^T p_k$ will be such that $y_k^T s_k > 0$, since

$$y_k^T s_k = \alpha_k y_k^T p_k = \alpha_k (g_{k+1} - g_k)^T p_k \geq \alpha_k (1 - \eta) g_k^T p_k$$

and this gives us an important reason to use that particular linesearch criterion with quasi-Newton methods. As we want to preserve symmetry, in order to impose hereditary positive-definiteness we will need to use at least a rank-two update. Consider the formula

$$B_{k+1} = B_k + \alpha u u^T + \beta v v^T$$

for some scalars α, β (not necessarily positive), and vectors u, v. From the quasi-Newton condition

$$y_k - B_k s_k = \alpha u^T s_k u + \beta v^T s_k v$$

and given that we have two vectors to approximate, y_k and $B_k s_k$, we could define u and v as proportional to these two vectors, to obtain

$$B_{k+1} = B_k - \frac{1}{s_k^T B_k s_k} B_k s_k s_k^T B_k + \frac{1}{y_k^T s_k} y_k y_k^T$$

This update is called the **Broyden-Fletcher-Goldfarb-Shanno update (BFGS)**, and it is the most successful one developed to date. It can be shown that it has the property of hereditary positive-definiteness, so that if B_0 is chosen to be positive definite, all matrices B_k will also be positive definite.

In summary, we can say that **Quasi-Newton methods** are a *generalization of the secant method* to find the root of the first derivative for multidimensional problems. In multi-dimensions the secant equation does not specify a unique solution, and quasi-Newton methods differ in how they constrain the solution. The **Broyden–Fletcher–Goldfarb–Shanno (BFGS) algorithm** showed before, is one of the most popular members of this class. Also in common use is **L-BFGS**, which is a limited-memory version of BFGS that is particularly suited to problems with very large numbers of variables (like >1000). The **BFGS-B** variant handles simple box constraints.

In SciPy, the **scipy.optimize.minimize** function implements **BFGS method** (`method='BFGS'`). It is also possible to run BFGS using any of the L-BFGS algorithms by setting the parameter L to a very large number.

A high-precision arithmetic version of BFGS (pBFGS), implemented in C++ and integrated with the high-precision arithmetic package ARPREC is robust against numerical instability (e.g. round-off errors).Therefore, this routine uses the gradient of the objective function. If the gradient is not given by the user, then it is estimated using first-differences. The BFGS method typically requires fewer function calls than the simplex algorithm even when the gradient must be estimated.

To demonstrate this algorithm, the Rosenbrock function is again used here. The *gradient* of the Rosenbrock function is the vector:

$$\begin{aligned}\frac{\partial f}{\partial x_j} &= \sum_{i=1}^{N} 200\left(x_i - x_{i-1}^2\right)\left(\delta_{i,j} - 2x_{i-1}\delta_{i-1,j}\right) - 2\left(1 - x_{i-1}\right)\delta_{i-1,j} \\ &= 200\left(x_j - x_{j-1}^2\right) - 400x_j\left(x_{j+1} - x_j^2\right) - 2\left(1 - x_j\right)\end{aligned}$$

This expression is valid for the interior derivatives. Special cases are

$$\begin{aligned}\frac{\partial f}{\partial x_0} &= -400x_0\left(x_1 - x_0^2\right) - 2\left(1 - x_0\right) \\ \frac{\partial f}{\partial x_{N-1}} &= 200\left(x_{N-1} - x_{N-2}^2\right)\end{aligned}$$

A Python function which computes this gradient is constructed by the code-segment:

In [5]:
```
def rosen_der(x):
    xm = x[1:-1]
    xm_m1 = x[:-2]
    xm_p1 = x[2:]
    der = np.zeros_like(x)
    der[1:-1] = 200*(xm-xm_m1**2) - 400*(xm_p1 - xm**2)*xm - 2*(1-xm)
    der[0] = -400*x[0]*(x[1]-x[0]**2) - 2*(1-x[0])
    der[-1] = 200*(x[-1]-x[-2]**2)
    return der
```

This gradient information is specified in the minimize function through the jac parameter as illustrated below.

```
In [13]: res = minimize(rosen, x0, method='BFGS', jac=rosen_der,
                        options={'disp': True})

         Optimization terminated successfully.
                 Current function value: 0.000000
                 Iterations: 51
                 Function evaluations: 63
                 Gradient evaluations: 63
```

```
In [1]: print(res.x)
```

9.4.5 Least-squares methods

Among the most common particular cases for the unconstrained optimization problem we have been studying is that of minimizing a sum of squares of functions. In this section we examine solution techniques for this problem.

The Linear Least Squares Problem

The linear least squares problem can be stated as

$$\min ||Ax - b||_2^2$$

where A is an $m \times n$ matrix, and usually $m \gg n$. When A is of full rank, the solution of this problem can be written as

$$x = (A^T A)^{-1} A^T b$$

Computing x by this method could be numerically unstable, since we are effectively squaring the condition number of A in forming the product $A^T A$. Thus, this approach should not be used in practice.

A numerically stable approach is one that uses the **QR factorization** of A, that is, the fact that we can find an orthogonal matrix Q, an upper triangular matrix R and a permutation matrix P such that

$$QAP = \begin{pmatrix} R & C \\ 0 & 0 \end{pmatrix}$$

The QR factorization can be obtained either by plane rotations or **Householder transformations**; we shall look at Householder transformations here. For any nonzero vector u, we define the corresponding **Householder transformation** as a symmetric elementary matrix of the form

$$H = I - \frac{1}{\beta} u u^T$$

where $\beta = \frac{1}{2} u^T u$. Note that the Householder transformation is an orthogonal matrix. Furthermore, because of its special structure the product Ha for any vector a is easy to form without matrix multiplications,

$$Ha = a - \frac{u^T a}{\beta} u$$

The above relation also implies that the transformed vector Ha will be identical to the original vector in all positions where the Householder vector u has zeros. We can use the Householder transformation to annihilate all elements but one of any vector a as follows:

Define

$$\big(\alpha = -sign(a_1)||a||_2, u = (a_1 - \alpha, a_2, a_3, \ldots, a_n)^T, \beta = \alpha(\alpha - a_1)\big)$$

then

$$Ha = (\alpha, 0, 0, \ldots, 0)^T$$

Thus, given an $m \times n$ matrix A, we can apply a sequence of Householder transformations, so that in each step we have a matrix A^i with zeros in the subdiagonal of the first i-1 columns, and we make zeros below the diagonal of the i-th column. We define H_i as

$$H_i = I - \frac{1}{\beta_i} u_i u_i^T$$

where $\bar{a}^i = (0, \ldots, 0, a_i^i i, \ldots, a_m^i i)^T$, $u_i = \bar{a}^i - \alpha_i e_i$ and $alpha_i = -sign(a_{ii}^i)||\bar{a}^i||_2$. For a 5×5 symbolic example with $i = 3$ we would have

$$\begin{pmatrix} 1 & & & & \\ & 1 & & & \\ & & \times & \times & \times \\ & & \times & \times & \times \\ & & \times & \times & \times \end{pmatrix} \begin{pmatrix} \times & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ & & \times & \times & \times \\ & & \times & \times & \times \\ & & \times & \times & \times \end{pmatrix} = \begin{pmatrix} \times & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ & & m & m & m \\ & & 0 & m & m \\ & & 0 & m & m \end{pmatrix}$$

and finally we get

$$H_n \ldots H_2 H_1 A = QA = \begin{pmatrix} R \\ 0 \end{pmatrix} = \hat{R}$$

where R is an upper triangular $n \times n$ matrix, and the orthogonal matrix Q is the product of the Householder transformations.

Once we have obtained the QR factorization of A, we can solve the least squares problem easily, as

$$||Ax - b||_2 = ||Q(Ax - b)||_2 = ||\hat{R}x - Qb||_2$$

Thus, the residual $||Ax - b||$ will be minimized when Rx is equal to the first n components of Qb.

The Nonlinear Least Squares Problem

The nonlinear least squares problem is the problem of minimizing the sum of squares of nonlinear functions, i.e.:

$$\min_x f(x) = \frac{1}{2}\sum_{i=1}^{m}(f_i(x))^2 = \frac{1}{2}||F(x)||_2^2$$

where $F(x)$ is a vector function whose i-th component is $f_i(x)$ and $||F(x)||$ is termed the residual at x. Problems of this type occur when fitting model functions to data, for example, in nonlinear parameter estimation. We can solve this problem by using any of the unconstrained minimization methods discussed so far; however, the special structure of the nonlinear least squares problem deserves special attention. In particular the gradient and the Hessian have a special form.

We define the Jacobian matrix of $F(x)$ as the $m \times n$ matrix $J(x)$ whose i-th row is the vector of partial derivatives corresponding to the i-th component of F, that is

$$(J(x))_{ij} = \frac{\partial f_i(x)}{\partial x_j}$$

If we let $H_i(x)$ denote the Hessian matrix of the function $f_i(x)$, then the gradient and the Hessian of $f(x)$ are given by

$$g(x) = J(x)^T F(x)$$

and

$$H(x) = J(x)^T J(x) + Q(x), \quad Q(x) = \textstyle\sum_{i=1}^{m} f_i(X)\nabla^2 f_i(x)$$

For example, suppose it is desired to fit a set of data $\{\mathbf{x}_i, \mathbf{y}_i\}$ to a known model, $\mathbf{y} = \mathbf{f}(\mathbf{x}, \mathbf{p})$ where $\mathbf{p}$ is a vector of parameters for the model that need to be found. A common method for determining which parameter vector gives the best fit to the data is to minimize the sum of squares of the residuals. The residual is usually defined for each observed data-point as

$$e_i(\mathbf{p}, \mathbf{y}_i, \mathbf{x}_i) = \|\mathbf{y}_i - \mathbf{f}(\mathbf{x}_i, \mathbf{p})\|.$$

An objective function to pass to any of the previous minization algorithms to obtain a least-squares fit is.

$$J(\mathbf{p}) = \sum_{i=0}^{N-1} e_i^2(\mathbf{p}).$$

The leastsq algorithm performs this squaring and summing of the residuals automatically. It takes as an input argument the vector function $\mathbf{e}(\mathbf{p})$ and returns the value of $\mathbf{p}$ which minimizes $J(\mathbf{p}) = \mathbf{e}^T\mathbf{e}$ directly. The user is also encouraged to provide the Jacobian matrix of the function (with derivatives down the columns or across the rows). If the Jacobian is not provided, it is estimated.

An example should clarify the usage. Suppose it is believed some measured data follow a sinusoidal pattern

$$y_i = A\sin(2\pi k x_i + \theta)$$

where the parameters A, k , and θ are unknown. The residual vector is

$$e_i = |y_i - A\sin(2\pi k x_i + \theta)|.$$

By defining a function to compute the residuals and (selecting an appropriate starting position), the least-squares fit routine can be used to find the best-fit parameters $\hat{A}$, $\hat{k}$, $\hat{\theta}$. This is shown in the following example:

In [1]:
```
x = arange(0,6e-2,6e-2/30)
A,k,theta = 10, 1.0/3e-2, pi/6
y_true = A*sin(2*pi*k*x+theta)
y_meas = y_true + 2*random.randn(len(x))
```

In [2]:
```
def residuals(p, y, x):
    A,k,theta = p
    err = y-A*sin(2*pi*k*x+theta)
    return err
```

In [3]:
```
def peval(x, p):
    return p[0]*sin(2*pi*p[1]*x+p[2])
```

In [4]:
```
p0 = [8, 1/2.3e-2, pi/3]
print(array(p0))
```

```
[  8.          43.47826087   1.04719755]
```

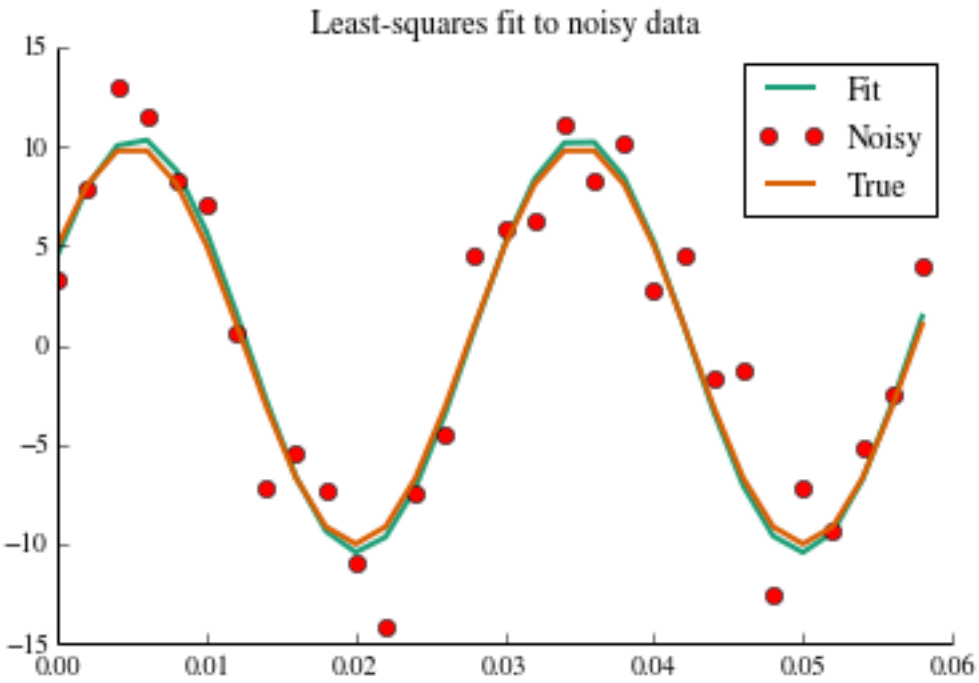
In [5]:
```
from scipy.optimize import leastsq
plsq = leastsq(residuals, p0, args=(y_meas, x))
print(plsq[0])
```

```
[ 10.42911267  33.62523969   0.45233951]
```

In [6]:
```
print(array([A, k, theta]))
```

```
[ 10.          33.33333333   0.52359878]
```

In [9]:
```
%matplotlib inline
import matplotlib.pyplot as plt
plt.plot(x,peval(x,plsq[0]),x,y_meas,'ro',x,y_true)
plt.title('Least-squares fit to noisy data')
plt.legend(['Fit', 'Noisy', 'True'])
plt.show()
```



9.5 Constrained Optimization with Scipy.Optimize

The rest of the chapter will be concerned with the problem of finding a (local) solution for the constrained program

$$\begin{array}{ll} \min_x & f(x) \\ \text{s.t.} & c(x) \geq 0 \end{array}$$

where the functions $f : \mathbb{R}^n \to \mathbb{R}$ and $c : \mathbb{R}^n \to \mathbb{R}^m$ are supposed to be twice-continuously differentiable. We start by introducing some definitions:

- We have already presented and used the concepts of **gradient vector** and **Hessian matrix** of a real function; in what follows we will also need the concept of the **Jacobian matrix** of a multivalued function. In particular, we define the **Jacobian matrix** of c at x, $A(x)$ (or A for short) as the $m \times n$ matrix
- Also, the **Lagrangian function** will play an important role in all that follows. For the problem under consideration, the Lagrangian function $L : \mathbb{R}^{n+m} \to \mathbb{R}$ is defined as

$$L(x, \alpha) = f(x) - \lambda^T c(x)$$

  where λ is some vector in $\mathbb{R}^m$ we will refer to this vector as the **Lagrange multipliers** of the problem. We will see that this function combines in a natural way the two goals of minimizing the objective function and computing feasible points.
- Given a point x, and the constraint vector $c(x)$, we will say that constraint i is **active** if $c_i(x) = 0$, that it is violated if $c_i(x) < 0$, and that it is inactive if $c_i(x) > 0$. Finally, a point x for which $c(x) \leq 0$ is called a **feasible point**, while if $c_i(x) < 0$ for some i, then we say that x is **infeasible**.

It is important to keep in mind that the main complication introduced into our problem is that we now have two goals to attain: we have to reduce the value of f as much as possible, as we had before, but at the same time we have to find points that satisfy the constraints $c(x) \geq 0$.

9.5.1 Optimality conditions

We already presented a characterization of a minimizer for our problem at the beginning of the chapter, namely x^* was a local minimizer if and only if $c(x^*) \geq 0$ (x^* was feasible) and

$$\exists \epsilon > 0 \; f(x^*) \leq f(x) \; \forall x \in \{y : ||y - x^*|| \leq \epsilon, c(y) \geq 0\}$$

As in the case of unconstrained optimization, this characterization is not very useful, given that it involves comparisons of $f(x^*)$ with the value of f at an infinite number of points.

Fortunately, we again have other characterizations for the minimizers that depend only on the values of f and c at x^*. The conditions that follow depend on the constraints satisfying certain regularity conditions known as constraint qualifications. Although we will not deal with that issue here, one possible example of these conditions (although a very strong one) is that at the solution the Jacobian of the active constraints should have full rank. We assume in all that follows that some constraint qualification holds.

Necessary Conditions

Denote by $\hat{A}(x)$ the Jacobian matrix for the subset of constraints active at the point x (the rows of the A corresponding to the active constraints).

if $x^*$is a local minimizer for (NLP) then
$g(x^*) = A(x^*)^T \lambda$
$\lambda^T c(x^*) = 0$
$\lambda \geq 0, c(x^*) \geq 0$
$s^T \nabla^2 L(x^*, \lambda) s \geq 0, \; \forall s \in \{t : \hat{A}(x^*)t = 0\}$

Note that the conditions involve only the active constraints ($\hat{A}$) at the solution, as the two conditions $g^* = A^{*T}\lambda$, $\lambda^T c^* = 0$ are equivalent to $\hat{A}^{*T}\lambda = g^*$. Also, the conditions can be expressed in terms of the Lagrangian function as requiring that at (x^*, λ) the gradient ∇L should vanish, and the point should be a minimum with respect to the x variable; note also that this does not imply a similar result for λ, in fact, L has a saddle point at (x^*, λ).

Of course, finding a solution to the problem will in general require finding not just the point x^*, but also the Lagrange multiplier vector λ.

Sufficient conditions

Let $A_+(x)$ denote the Jacobian of the constraints that are active at x and also have a positive multiplier $\lambda_i > 0$ associated to them.

if the following conditions hold at x^*
$g(x^*) = A(x^*)^T \lambda$
$\lambda^T c(x^*) = 0$
$\lambda \geq 0,$
$c(x^*) \geq 0$
$s^T \nabla^2 L(x^*, \lambda) s > 0, \; \forall s \in \{t : \hat{A}_+(x^*)t = 0, t \neq 0\}$
then x^* is a local minimizer for (NPL).

As we did in the unconstrained case, we require for the sufficient conditions that a certain matrix should be positive definite. Observe the special role played by the zero multipliers. For a justification of these conditions you are referred to Luenberger book.

Lagrange Multipliers: Sensitivity Analysis

In the optimality conditions for the constrained case the vector λ of Lagrange multipliers plays a very important role. We now offer an interpretation of their meaning. Assume that we alter the problem to

$$\begin{array}{ll} \min_x & f(x) \\ \text{s.t.} & c(x) + \epsilon e_i \geq 0 \end{array}$$

where e_i is the i-th unit vector. We denote by $x^*(\epsilon)$ the solution for the perturbed problem, and by $\lambda + \delta\lambda$ the multiplier vector. We have

$$f(x^*(\epsilon)) - f(x^*) = \epsilon g(x^*)^T \frac{dx^*}{d\epsilon} + o(\epsilon)$$

$$c(x^*(\epsilon)) + \epsilon e_i - c(x^*) = \epsilon e_i + \epsilon A(x^*) \frac{dx^*}{d\epsilon} + o(\epsilon)$$

Multiplying the second equation by $\lambda + \delta\lambda$ we have

$$0 = \delta\lambda^T c(x^*) + \epsilon(\lambda_i + \delta\lambda_i) + \epsilon g(x^*)^T \frac{dx^*}{d\epsilon} + o(\epsilon)$$

noting that $\delta\lambda = o(1)$ as $\epsilon \to 0$. Replacing this result in the first equation,

$$\lim_{\epsilon \to 0} \frac{f(x^*(\epsilon) - f(x^*)}{\epsilon} = -\lambda_i$$

where we have used the fact that either $c_i(x^*) = 0$, or otherwise the i-th constraint cannot be active at $x^*(\epsilon)$ for all sufficiently small values of ϵ.

This means is that if constraint i is slightly perturbed (its right-hand side is changed from 0 to $-\epsilon$), then the value of the objective function will change by approximately $-\epsilon\lambda_i$ (this is a first-order approximation) when moving to the modified constraint. In particular, if $\lambda_i < 0$ this tells us we can improve the value of the objective function by remaining feasible but moving away from the constraint.

This result is useful because it gives us information about what constraints would be interesting to slacken to get a better value for the objective function, and also because it tells us how much we should be willing to pay to modify the constraint.

CHAPTER

TEN

STATISTICAL MODELLING WITH SCIKIT-LEARN

10.1 Some Basics before to start

Something about Statistical Learning

Statistical Learning concerns the construction and study of statistical methodologies, models or systems that learn from data. For example, a statistical learning system could be trained on email messages to learn to distinguish between spam and non-spam messages. After learning, it can then be used to classify new email messages into spam and non-spam folders. "Optimizing a performance criterion using example data and past experience", said by E. Alpaydin, gives an easy but faithful description about statistical learning. In machine learning, data plays an indispensable role, and the learning algorithm is used to discover and learn knowledge or properties from the data.

In general, a **Statistical Learning** problem considers a set of n samples of data and then tries to predict properties of unknown data. If each sample is more than a single number and, for instance, a multi-dimensional entry (aka multivariate data), is it said to have several attributes or features. In most Statistical Learning applications, the data is in a 2D array of shape `[n_samples x n_features]`, where the number of features is the same for each object, and each feature column refers to a related piece of information about each sample.

Statistical Learning can be divided into two broad regimes: **Supervised learning** and **Unsupervised learning**.

- **Supervised Learning**, in which the data comes with additional attributes that we want to predict. This problem can be either:
  - **Classification**: samples belong to two or more classes and we want to learn from already labeled data how to predict the class of unlabeled data. An example of classification problem would be the handwritten digit recognition example, in which the aim is to assign each input vector to one of a finite number of discrete categories. Another way to think of classification is as a discrete (as opposed to continuous) form of supervised learning where one has a limited number of categories and for each of the n samples provided, one is to try to label them with the correct category or class.
  - **Regression**: if the desired output consists of one or more continuous variables, then the task is called regression. An example of a regression problem would be the prediction of the length of a salmon as a function of its age and weight.
- **Unsupervised learning**, in which the training data consists of a set of input vectors x without any corresponding target values. The goal in such problems may be to discover groups of similar examples within the data, where it is called clustering, or to determine the distribution of data within the input space, known as density estimation, or to project the data from a high-dimensional space down to two or three dimensions for the purpose of visualization

Something about Scikit-Learn

Scikit-Learn project provides an open source machine learning library for the Python programming language. The main goal of the project is to provide efficient and well-established machine learning tools within a programming environment that is accessible to non-machine learning experts and re-usable in various scientific areas. The project is not a novel domain-specific language, but this library provides statistical learning idioms to a general-purpose high-level language. Among other things, it includes classical learning algorithms, model evaluation and selection tools, as well as preprocessing procedures. The library is distributed under the simplified BSD license, encouraging its use in both academic and commercial settings.

The library has been designed to tie in with the set of numeric and scientific packages centered around the **NumPy** and **SciPy** libraries. **NumPy** (Van der Walt et al.,2011) augments Python with a contiguous numeric array datatype and fast array computing primitives, while **SciPy** (Haenel et al., 2013) extends it further with common numerical operations, either by implementing these in Python/NumPy or by wrapping existing C/C++/Fortran implementations. Building upon this stack, a series of libraries called **Scikits** were created, to complement SciPy with domain specific toolkits. Currently, the two most popular and feature-complete ones are by far Scikit-Learn and Scikit-Image, which does image processing.

Scikit-Learn as a project started in 2007 as a Google Summer of Code project leaded by David Cournapeau. Later that year, Matthieu Brucher started work on this project as part of his thesis. In 2010 Fabian Pedregosa, Gael Varoquaux, Alexandre Gramfort and Vincent Michel of INRIA took leadership of the project and made the first public release, February the 1st 2010. Since then, several releases have appeared following a ~3 month cycle, and a striving international community has been leading the development. This project is mostlyly financed by INRIA and Google, and it has been always an open-source project build as a community effort, and as such more than 300 scientist and developers have contributed to it over the years.

The structure of **Scikit-Learn**: Some of the following text is taken from the Scikit-Learn API paper

> All objects within **Scikit-Learn** share *a uniform common basic API consisting of three complementary interfaces*:
>
> 1. An **Estimator interface** for *building and fitting models*,
> 2. A **Predictor interface** for *making predictions* and
> 3. A **Transformer interface** for *converting data*
>
> The **Estimator interface** is at the core of the library. It defines instantiation mechanisms of objects and exposes a **fit** method for learning a model from training data. All supervised and unsupervised learning algorithms (e.g., for classification, regression or clustering) are defined as objects implementing this interface. Machine learning tasks like feature extraction, feature selection or dimensionality reduction are also provided as estimators. Actual learning is performed by the fit method. This method is called with **training data** (e.g., supplied as two arrays `X_train` and `y_train` in supervised learning estimators). Its task is to run a learning algorithm and to determine model-specific parameters from the training data and set these as attributes on the estimator object.

As example of **Estimator Interface** see these lines:

```
clf = LogisticRegression()
clf.fit(X_train, y_train)
```

If one changes classifiers, say, to a Random Forest classifier, one would simply replace `LogisticRegression()` in the snippet above by `RandomForestClassifier()`.

> The **Predictor Interface** extends the notion of an estimator by adding a predict method that takes an array `X_test` and produces predictions for `X_test`, based on the learned parameters of the estimator. In the case of supervised learning estimators, this method typically returns the predicted labels or values computed by the model. Some unsupervised learning estimators may also implement the predict interface, such as k-means, where the predicted values are the cluster labels.

```
clf.predict(X_test)
```

Since it is common to modify or [U+FB01]lter data before feeding it to a learning algorithm, some estimators in the library implement a **transformer** interface which de[U+FB01]nes a transform method. It takes as input some new data `X_test` and yields as output a transformed version of `X_test`. Preprocessing, feature selection, feature extraction and dimensionality reduction algorithms are all provided as transformers within the library.

This is usually done via the `fit_transform` method. For example, to do a PCA:

```
pca = RandomizedPCA(n_components=2)
train_x = pca.fit_transform(train_x)
test_x = pca.transform(test_x)
```

The training set here is "fit" to find the PC components, and then then transformed. Since `pca.fit()` by itself changes the `pca` object, if we want to transform other data using the same transformation we simply call `transform` subsequently.Finally, for now, there is the concept of a meta-estimator, which behaves quite similarly to standard estimators, but allows us to wrap, for example, cross-validation, or methods that build and combine simpler models or schemes. For example:

```
from sklearn.multiclass import OneVsOneClassifier
clf=OneVsOneClassifier(LogisticRegression())
```

In scikit-learn, model selection is supported in two distinct meta-estimators, GridSearchCV and RandomizedSearchCV. They take as input an estimator (basic or composite), whose hyper-parameters must be optimized, and a set of hyperparameter settings to search through.

Most machine learning algorithms implemented in scikit-learn expect data to be stored in a two-dimensional array or matrix. The arrays can be either numpy arrays, or in some cases scipy.sparse matrices. The size of the array is expected to be $[n\_samples, n\_features]$

To get a grip on how to do machine learning with scikit-learn, it is worth working through the entire set of notebooks at: https://github.com/jakevdp/sklearn_pycon2013 . These go relatively fast, are fun to read. The repository at https://github.com/jakevdp/sklearn_scipy2013 has more advanced notebooks. The `rendered_notebooks` folder here is useful with worked-out examples.

10.2 Three Machine Learning Cases with Scikit-learn

10.3 Linear Regression: Is profitable to study?

We'll see an example of the concepts mentioned above by considering a linear regression problem. Let us load the **Census Data** set.

In [4]:
```
census_data = pd.read_csv("./data/census_demographics.csv")

census_data.head(10)
```

Out [4]:

```
                    state  per_black  per_hisp  per_white  educ_hs
educ_coll  average_income  median_income  pop_density      vote_pop
older_pop  per_older  per_vote
0                ALABAMA       26.5       4.0       66.8     81.4
21.7           22984          42081         94.4   3001712.500
672383.600      0.140     0.625
1                 ALASKA        3.6       5.8       63.7     90.7
27.0           30726          66521          1.2    475548.444
58540.158      0.081     0.658
```

```
2                ARIZONA         4.5       30.1        57.4      85.0
26.3            25680           50448          56.3   3934880.535
920515.710       0.142      0.607
3               ARKANSAS        15.6        6.6        74.2      81.9
19.1            21274           39267          56.0   1798043.148
428944.934       0.146      0.612
4             CALIFORNIA         6.6       38.1        39.7      80.7
30.1            29188           60883         239.1  24009747.944
4409953.704      0.117      0.637
5               COLORADO         4.3       20.9        69.7      89.3
35.9            30151           56456          48.5   3310567.012
578197.948       0.113      0.647
6           CONNECTICUT         11.1       13.8        70.9      88.4
35.2            36775           67740         738.1   2263008.088
515622.096       0.144      0.632
7               DELAWARE         21.9        8.4        65.1      87.0
27.7            29007           57599         460.8    568773.645
133348.845       0.147      0.627
8  DISTRICT OF COLUMBIA         50.7        9.5        35.3      86.5
49.2            42078           58526        9856.5    442485.136
70451.544       0.114      0.716
9                FLORIDA         16.5       22.9        57.5      85.3
25.9            26551           47661         350.6  11701330.788
3354127.392      0.176      0.614
```

Clean the data set, and have it indexed by the state abbrev.

In [5]:
```
states_abbrev_dict = {
        'AK': 'Alaska',
        'AL': 'Alabama',
        'AR': 'Arkansas',
        'AS': 'American Samoa',
        'AZ': 'Arizona',
        'CA': 'California',
        'CO': 'Colorado',
        'CT': 'Connecticut',
        'DC': 'District of Columbia',
        'DE': 'Delaware',
        'FL': 'Florida',
        'GA': 'Georgia',
        'GU': 'Guam',
        'HI': 'Hawaii',
        'IA': 'Iowa',
        'ID': 'Idaho',
        'IL': 'Illinois',
        'IN': 'Indiana',
        'KS': 'Kansas',
        'KY': 'Kentucky',
        'LA': 'Louisiana',
        'MA': 'Massachusetts',
        'MD': 'Maryland',
        'ME': 'Maine',
        'MI': 'Michigan',
        'MN': 'Minnesota',
        'MO': 'Missouri',
        'MP': 'Northern Mariana Islands',
        'MS': 'Mississippi',
        'MT': 'Montana',
        'NA': 'National',
        'NC': 'North Carolina',
        'ND': 'North Dakota',
```

```
        'NE': 'Nebraska',
        'NH': 'New Hampshire',
        'NJ': 'New Jersey',
        'NM': 'New Mexico',
        'NV': 'Nevada',
        'NY': 'New York',
        'OH': 'Ohio',
        'OK': 'Oklahoma',
        'OR': 'Oregon',
        'PA': 'Pennsylvania',
        'PR': 'Puerto Rico',
        'RI': 'Rhode Island',
        'SC': 'South Carolina',
        'SD': 'South Dakota',
        'TN': 'Tennessee',
        'TX': 'Texas',
        'UT': 'Utah',
        'VA': 'Virginia',
        'VI': 'Virgin Islands',
        'VT': 'Vermont',
        'WA': 'Washington',
        'WI': 'Wisconsin',
        'WV': 'West Virginia',
        'WY': 'Wyoming'
}
abbrev_states_dict = {v: k for k, v in states_abbrev_dict.items()}
```

In [6]:
```
def capitalize(s):
    s = s.title()
    s = s.replace("Of", "of")
    return s
census_data["State"] = census_data.state.map(capitalize)
del census_data["state"]
census_data['State']=census_data['State'].replace(abbrev_states_dict)
census_data.set_index("State", inplace=True)
census_data.head(10)
```

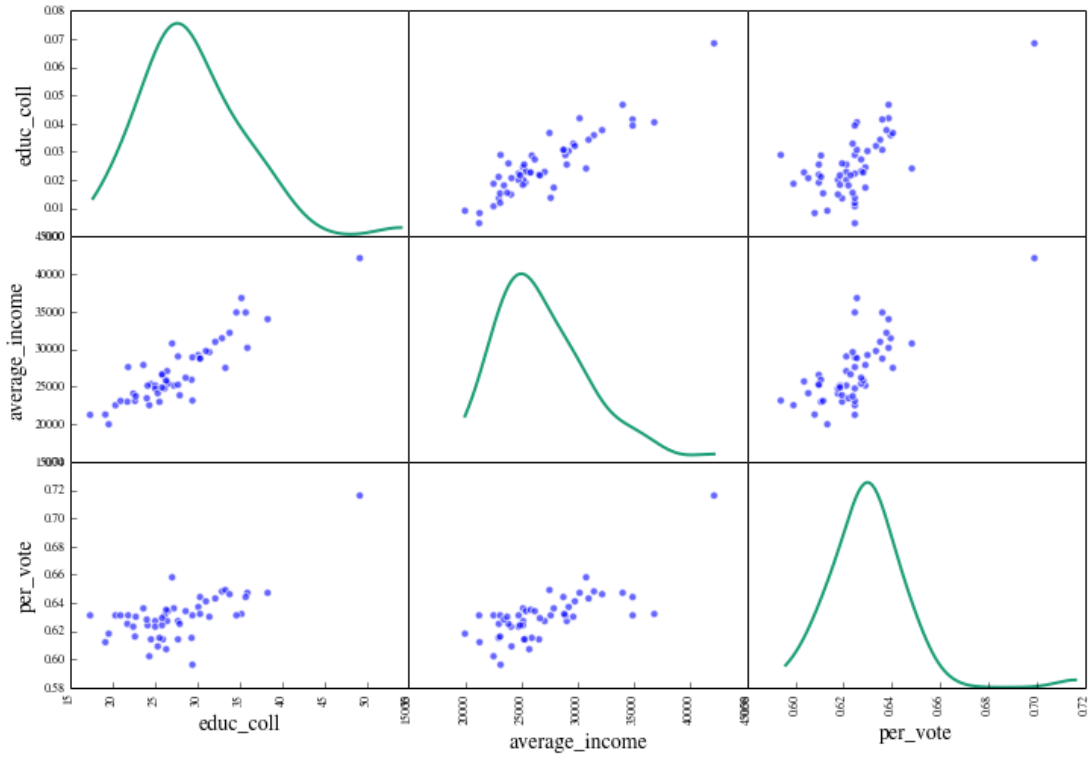
Out [6]:

State	per_black	per_hisp	per_white	educ_hs	educ_coll	average_income	median_income	pop_density	vote_pop	older_pop	per_older	per_vote
AL	26.5	4.0	66.8	81.4	21.7	22984	42081	94.4	3001712.500	672383.600	0.140	0.625
AK	3.6	5.8	63.7	90.7	27.0	30726	66521	1.2	475548.444	58540.158	0.081	0.658
AZ	4.5	30.1	57.4	85.0	26.3	25680	50448	56.3	3934880.535	920515.710	0.142	0.607
AR	15.6	6.6	74.2	81.9	19.1	21274	39267	56.0	1798043.148	428944.934	0.146	0.612
CA	6.6	38.1	39.7	80.7	30.1	29188	60883	239.1	24009747.944	4409953.704	0.117	0.637
CO	4.3	20.9	69.7	89.3	35.9	30151	56456	48.5	3310567.012	578197.948	0.113	0.647

```
CT           11.1       13.8        70.9      88.4        35.2
36775           67740         738.1    2263008.088    515622.096
0.144      0.632
DE           21.9        8.4        65.1      87.0        27.7
29007           57599         460.8     568773.645    133348.845
0.147      0.627
DC           50.7        9.5        35.3      86.5        49.2
42078           58526        9856.5     442485.136     70451.544
0.114      0.716
FL           16.5       22.9        57.5      85.3        25.9
26551           47661         350.6   11701330.788   3354127.392
0.176      0.614
```

We use a SPLOM to visualize some columns of this dataset. In Panda's the SPLOM is a one-liner.

In [7]:
```
smaller_frame=census_data[['educ_coll', 'average_income', 'per_vote']]
from pandas.tools.plotting import scatter_matrix
axeslist=scatter_matrix(smaller_frame, s=100, alpha=0.6, figsize=(12, 8), diagonal="kd
```



Notice how `average_income` seems to have a strong correlation with `educ_coll`. Lets try and regress the former against the latter. One might expect that the average income is higher in states which have "better" education systems and send more students to college. First lets confirm our intuition by seeing the co-relations.

In [8]:
```
smaller_frame.corr()
```

Out [8]:

	educ_coll	average_income	per_vote
educ_coll	1.000000	0.894066	0.670977
average_income	0.894066	1.000000	0.732703

```
per_vote          0.670977        0.732703  1.000000
```

We carry out the regression, first **standardizing our variables**. This is strictly not necessary, but we are doing it as we wish to play around with PCA. Since `scikit-learn` wants a `n_sample` rows times `n_features` matrix, we need to reshape the `x` variable. We store both an `_vec` variable, which is easier to plot with, as well as the reshaped variable.

In [9]:
```
from sklearn.linear_model import LinearRegression

X_HD = smaller_frame[['educ_coll', 'average_income']].values

# Standardizing the sample
X_HDn = (X_HD - X_HD.mean(axis=0))/X_HD.std(axis=0)

educ_coll_std_vec = X_HDn[:,0]
educ_coll_std = educ_coll_std_vec.reshape(-1,1)
average_income_std_vec = X_HDn[:,1]
average_income_std = average_income_std_vec.reshape(-1,1)
```

We split the data into a **training set** and a **testing set**. By default, *25% of the data is reserved for testing*. This is the first of multiple ways that we will see to do this.

In [10]:
```
from sklearn.cross_validation import train_test_split

X_train, X_test, y_train, y_test = train_test_split(educ_coll_std, average_income_std_
```

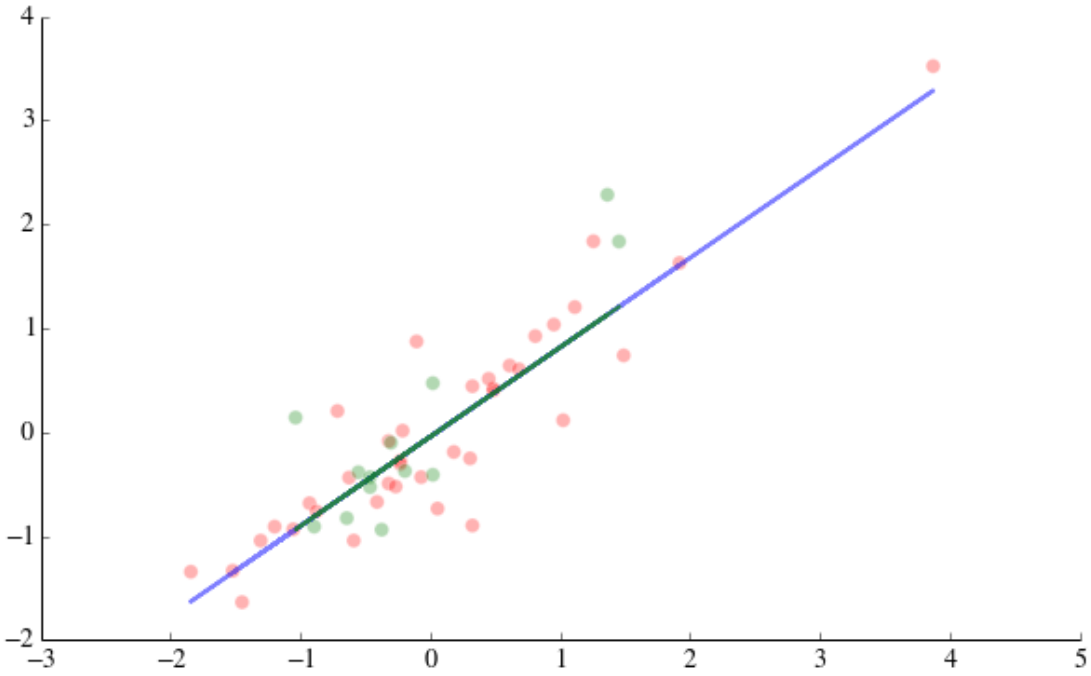
We **use the training set for the fit**, and find what our **predictions** ought to be on both the **training** and **test set**.

In [11]:
```
clf1 = LinearRegression()
clf1.fit(X_train, y_train)
predicted_train = clf1.predict(X_train)
predicted_test = clf1.predict(X_test)
trains=X_train.reshape(1,-1).flatten()
tests=X_test.reshape(1,-1).flatten()
print clf1.coef_, clf1.intercept_
```

```
[ 0.85965684] -0.0425101136644
```

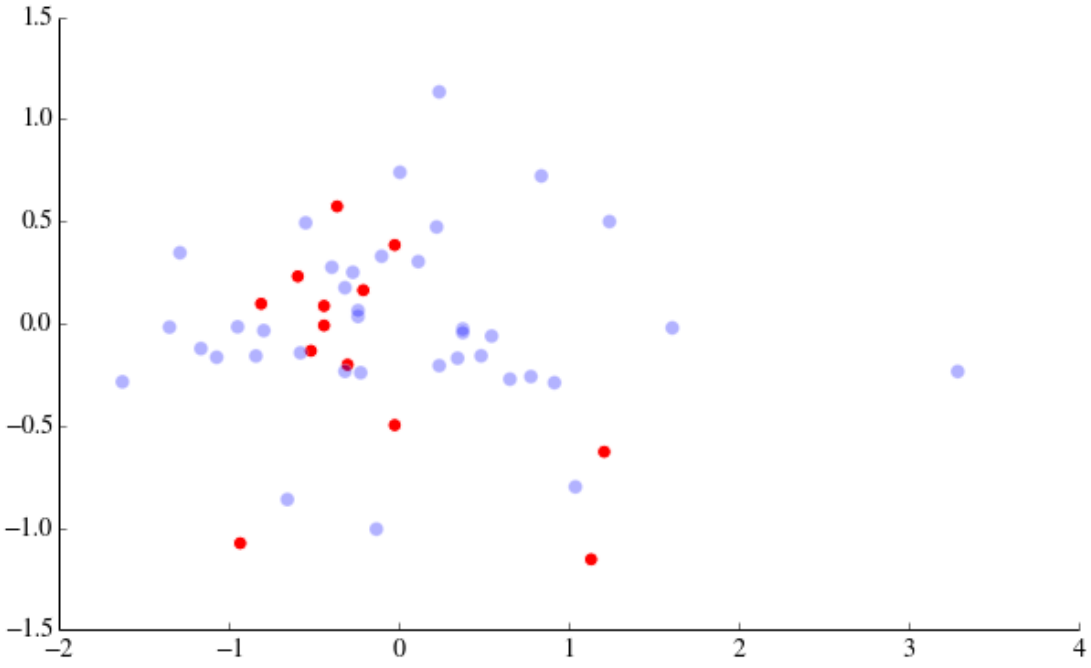
We plot the scatter against the fit for both training and test data.

In [12]:
```
plt.scatter(X_train, y_train,c='r', s=60, alpha=0.3)
plt.scatter(X_test, y_test,c='g', s=60, alpha=0.3)

plt.plot(trains, predicted_train, c='b', alpha=0.5)
plt.plot(tests, predicted_test, c='g', alpha=0.7)
remove_border()
```



We then look at the **residuals**, again *on both sets*.

```
In [13]: plt.scatter(predicted_test, predicted_test- y_test, c='r', s=60)
         plt.scatter(predicted_train, predicted_train- y_train, c='b', s=60, alpha=0.3)
         remove_border()
```



We ask **Scikit-Learn** to spit out the R^2. If you'd like R-style detailed information about your regression, use `statsmodels` instead.

In [14]:
```
clf1.score(X_train, y_train), clf1.score(X_test, y_test)
```

Out [14]:
```
(0.82954117098448632, 0.69008661326460008)
```

10.4 Logistic Regression: The case of the Space-shuttle Challenger

This case is inspired in **Edward Tufte** book title "Visual and Statistical Thinking: Displays of Evidence for Making Decisions" (see in http://www.edwardtufte.com/tufte/ebooks):

> On January 28, 1986, the space shuttle Challenger exploded and seven astronauts died because two rubber O-rings leaked. These rings had lost their resiliency because the shuttle was launched on a very cold day. Ambient temperatures were in the low 30s and the O-rings themselves were much colder, less than 20F.
>
> One day before the flight, the predicted temperature for the launch was 26F to 29F. Concerned that the rings would not seal at such a cold temperature, the engineers who designed the rocket opposed launching Challenger the next day.

But they did not make their case persuasively, and were over-ruled by NASA.

In [15]:
```
from IPython.display import Image as Im
from IPython.display import display
Im('./data/shuttle.png')
```

Out [15]:

The image above shows the leak, where the O-ring failed.

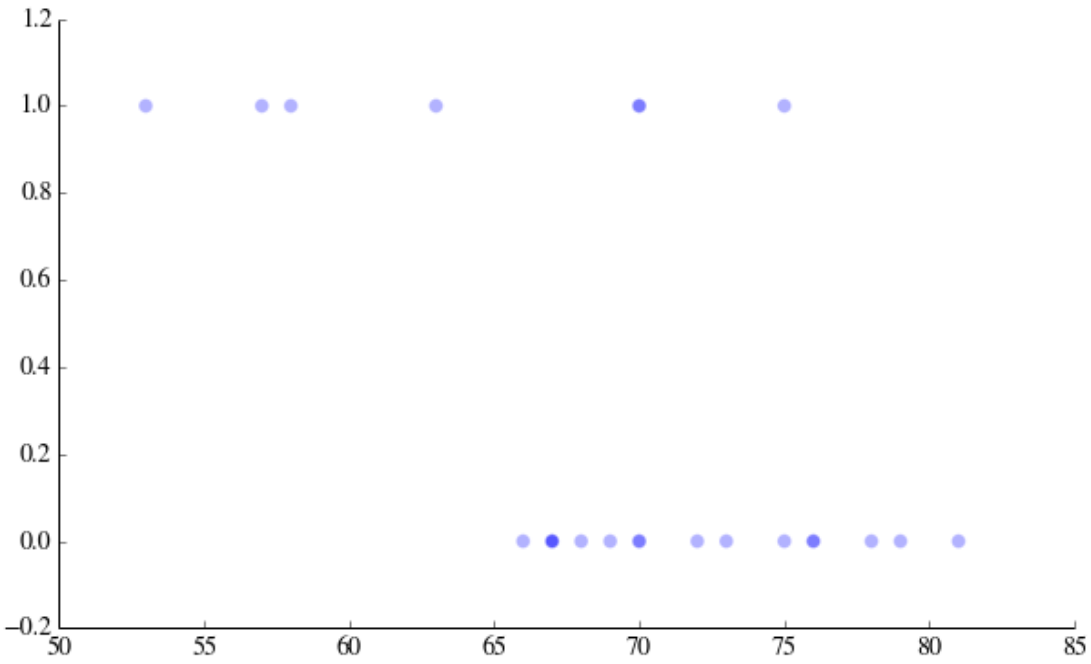
We have here data on previous failures of the O-rings at various temperatures.

In [16]:
```
data=np.array([[float(j) for j in e.strip().split()] for e in open("./data/chall.txt")
data
```

Out [16]:
```
array([[ 66.,   0.],
       [ 70.,   1.],
       [ 69.,   0.],
       [ 68.,   0.],
       [ 67.,   0.],
       [ 72.,   0.],
       [ 73.,   0.],
       [ 70.,   0.],
       [ 57.,   1.],
       [ 63.,   1.],
       [ 70.,   1.],
       [ 78.,   0.],
       [ 67.,   0.],
       [ 53.,   1.],
       [ 67.,   0.],
       [ 75.,   0.],
       [ 70.,   0.],
       [ 81.,   0.],
       [ 76.,   0.],
       [ 79.,   0.],
       [ 75.,   1.],
       [ 76.,   0.],
       [ 58.,   1.]])
```

Lets plot this data

In [17]:
```
temps, pfail = data[:,0], data[:,1]
plt.scatter(temps, pfail, c='b', s=60, alpha=0.3)
axes=plt.gca()
axes.grid(False)
remove_border(axes)
```



Notice that **1** represents **failure**. This graph has a classic **sigmoid shape**, so one might expect **logistic regression** to work (for technicalities related with the logistic regression check chapter 4.4. in the book from *Hastie et al. (2009) "The Elements of Statistical Learning"*). Furthermore, we do want to find the **probability of failure** and make **predictions** from there as:

$$\log \frac{\Pr(y=1|X=x)}{\Pr(y=0|X=x)} = \beta_0 + \beta^T x$$

Logistic regression is carried out in the same way as linear. However, there is the "pesky" matter of setting the **regularization co-efficient** `C`. The default `C` in sklearn is 1. The meaning of `C` is simple: the larger the `C`, the lesser the regularization (in fact, it is defined as the inverse of the regularization strength). The smaller the `C` the higher the regularization, or in other words, we include and L^1 penalty function:

$$\max_{\beta_0,\beta} \sum_{i=1}^{N} \{[y_i(\beta_0 + \beta^T x_i) - log(1 + \exp^{\beta_0+\beta^T x_i})] - \frac{1}{C}\sum_{j=1}^{p} |\beta_j|\}$$

or with an L^2 penalty function:

$$\max_{\beta_0,\beta} \sum_{i=1}^{N} \{[y_i(\beta_0 + \beta^T x_i) - log(1 + \exp^{\beta_0+\beta^T x_i})] - \frac{1}{C}\sum_{j=1}^{p} (\beta_j)^2\}$$

What does regularization do? Larger regularizations (lower `C`) penalize the values of regression coefficients. Smaller ones (bigger `C`) let the co-efficients range widely. Thus, larger `C` let the regression coefficients range widely. Scikit-learn bakes in two penalties: a L^2 penalty which penalizes the sum of the squares of the coefficients, and a L^1 penalty which penalizes the sum of the absolute values.

There is a reason for doing this is *variable selection* and *shrinks the regression coeficients* by imposing a penaly on their size. When there are many correlated variables in a linear regression model, their coefficients can become poorly determined and exhibit high variance. A wildly large positive coefficient on one variable can be canceled by a similar large negative coefficient on its correlated cousin. We want to make sure we can get away with the

simplest model that describes our data, even if that might increase the bias side of the bias-variance tradeoff a bit. Both optimization problems are concave and a solution can be found using nonlinear programming methods (Koh et al. 2007 for example).

Remember here, though, that we have just two co-efficents: an **intercept**, and the outside **temperature**. So we do not expect to need regularization much. Indeed lets set `C=1000`.

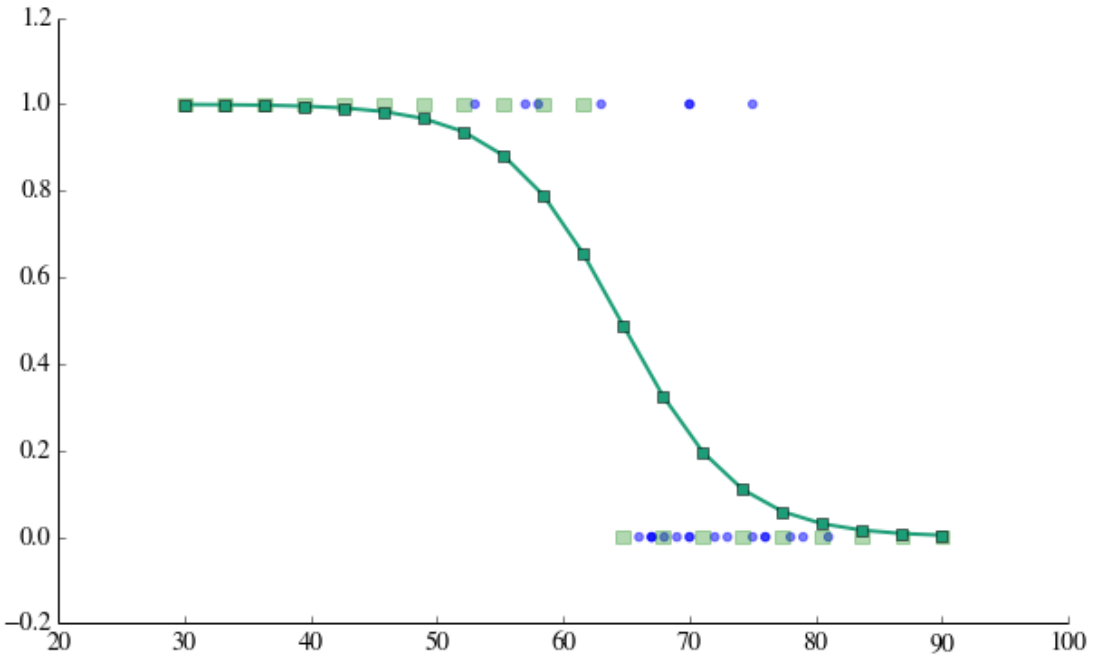
In [18]:
```
from sklearn.linear_model import LogisticRegression
reg = 1000.
clf4 = LogisticRegression(C = reg)
clf4.fit(temps.reshape(-1,1), pfail)
```

Out [18]:
```
LogisticRegression(C=1000.0, class_weight=None, dual=False,
          fit_intercept=True, intercept_scaling=1, penalty='l2',
          random_state=None, tol=0.0001)
```

Lets make predictions, get the associated probabilities, and plot them. We create a new grid of temperatures to evaluate our predictions at. Note that we do not do a test/train split: we have only 23 data points, but need to shut down the launch if there is any doubt. (One wishes...)

In [78]:
```
tempsnew = np.linspace(30., 90., 20)
probs = clf4.predict_proba(tempsnew.reshape(-1,1))[:, 1]
predicts = clf4.predict(tempsnew.reshape(-1,1))
```

In [79]:
```
plt.scatter(temps, pfail, color="blue", s=20, alpha=0.5)
axes=plt.gca()
axes.grid(False)

plt.plot(tempsnew, probs, marker='s')
plt.scatter(tempsnew, predicts, marker='s', color="green", s=60, alpha=0.3)

remove_border(axes)
```



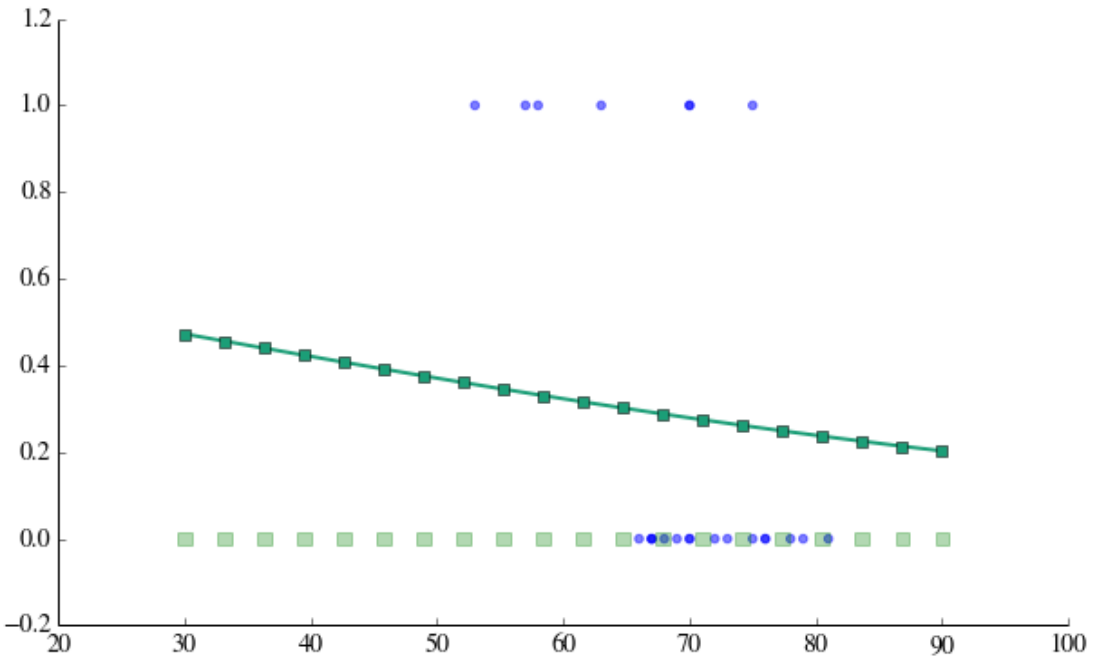
We use **pandas `crosstab`** to write a **table of prediction vs failure** on the "training" set. As one might expect, the mislabellings come at the higher temperatures.

```
In [25]: pd.crosstab(pfail, clf4.predict(temps.reshape(-1,1)), rownames=["Actual"], colnames=["
```

```
Out [25]:
Predicted   0  1
Actual
0          16  0
1           3  4
```

Now, let's carry out a **Logistic Regression** with Scikit-learn's **default value for `C`**. Let us do a plot similar to the scatterplot above, and carry out the cross-tabulation. What happens?

```
In [80]: clf4w = LogisticRegression()
         clf4w.fit(temps.reshape(-1,1), pfail)

         probsw = clf4w.predict_proba(tempsnew.reshape(-1,1))[:, 1]
         predictsw = clf4w.predict(tempsnew.reshape(-1,1))

         plt.scatter(temps, pfail, color="blue", s=20, alpha=0.5)
         axes = plt.gca()
         axes.grid(False)

         plt.plot(tempsnew, probsw, marker='s')
         plt.scatter(tempsnew, predictsw, marker='s', color="green", s=60, alpha=0.3)
         remove_border(axes)
```




```
In [81]: pd.crosstab(pfail, clf4w.predict(temps.reshape(-1,1)), rownames=["Actual"], colnames=[
```

Out [81]:

```
Predicted   0
Actual
0          16
1           7
```

Logistic Regression with cross-validation

We now actually go ahead and to the **train/test split**. Not once but multiple times, on a grid search, **for different values of C**.

For each C, we:

1. **create `n_folds` folds.** Since the data size is 23 here, and we have 5 folds, we roughly split the data into 5 folds of 4-5 values each, randomly.
2. We then **train on 4 of these folds**, test on the 5th
3. We **average the results** of all such combinations
4. We move on **to the next value of C**, and find the **optimal value** that minimizes mean square error.
5. We **finally use that value to make the final fit**.

Notice the structure of the `GridSearchCV` estimator in `cv_optimize`.

In [28]:

```
from sklearn.linear_model import LogisticRegression

def fit_logistic(X_train, y_train, reg=0.0001, penalty="l2"):
    clf = LogisticRegression(C=reg, penalty=penalty)
    clf.fit(X_train, y_train)
    return clf

from sklearn.grid_search import GridSearchCV

def cv_optimize(X_train, y_train, paramslist, penalty="l2", n_folds=10):
    clf = LogisticRegression(penalty=penalty)
    parameters = {"C": paramslist}
    gs = GridSearchCV(clf, param_grid = parameters, cv = n_folds)
    gs.fit(X_train, y_train)
    return gs.best_params_, gs.best_score_

def cv_and_fit(X_train, y_train, paramslist, penalty = "l2", n_folds = 5):
    bp, bs = cv_optimize(X_train, y_train, paramslist, penalty = penalty,
    n_folds = n_folds)
    print "BP,BS", bp, bs
    clf = fit_logistic(X_train, y_train,
    penalty = penalty, reg = bp['C'])
    return clf
```

In [29]:

```
clf = cv_and_fit(temps.reshape(-1,1), pfail, np.logspace(-4, 3, num=100))
```

```
BP,BS {'C': 521.40082879996839} 0.869565217391
```

In [30]:

```
pd.crosstab(pfail, clf.predict(temps.reshape(-1,1)), rownames=["Actual"],
 colnames=["Predicted"])
```

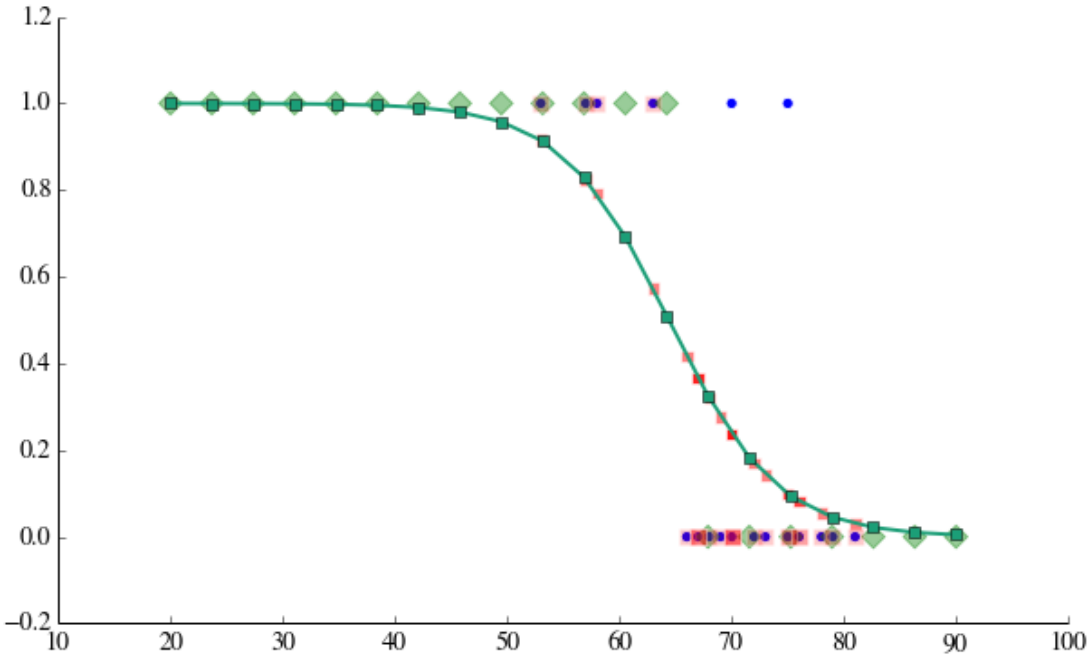
Out [30]:

```
Predicted   0  1
Actual
```

```
0           16   0
1            3   4
```

We plot our results, this time also marking in red the predictions on the "training" set.

In [31]:
```
plt.scatter(temps, pfail, s=40)
axes = plt.gca()
axes.grid(False)

probs = clf.predict_proba(tempsnew.reshape(-1,1))[:,1]
predicts = clf.predict(tempsnew.reshape(-1,1))

plt.plot(tempsnew, probs, marker='s')
plt.scatter(tempsnew, predicts, marker='D', color="green", s=80, alpha=0.4)

train_probs = clf.predict_proba(temps.reshape(-1,1))[:,1]
plt.scatter(temps, train_probs, marker='s', c='r', alpha=0.5, s=40)

train_predicts = clf.predict(temps.reshape(-1,1))
plt.scatter(temps, train_predicts, marker='s', c='r', alpha=0.2, s=80)

remove_border(axes)
```



The failures in prediction are, exactly where you might have expected them to be, as before.

In [32]:
```
zip(temps,pfail, clf.predict(temps.reshape(-1,1)))
```

Out [32]:
```
[(66.0, 0.0, 0.0),
 (70.0, 1.0, 0.0),
 (69.0, 0.0, 0.0),
 (68.0, 0.0, 0.0),
 (67.0, 0.0, 0.0),
 (72.0, 0.0, 0.0),
 (73.0, 0.0, 0.0),
```

```
(70.0, 0.0, 0.0),
(57.0, 1.0, 1.0),
(63.0, 1.0, 1.0),
(70.0, 1.0, 0.0),
(78.0, 0.0, 0.0),
(67.0, 0.0, 0.0),
(53.0, 1.0, 1.0),
(67.0, 0.0, 0.0),
(75.0, 0.0, 0.0),
(70.0, 0.0, 0.0),
(81.0, 0.0, 0.0),
(76.0, 0.0, 0.0),
(79.0, 0.0, 0.0),
(75.0, 1.0, 0.0),
(76.0, 0.0, 0.0),
(58.0, 1.0, 1.0)]
```

We note that the true story was even worse than our data made it out to be! We did not take the severity of the incidents into account. How could we have incorporated this severity into our analysis? (these images are taken from Tufte's booklet).

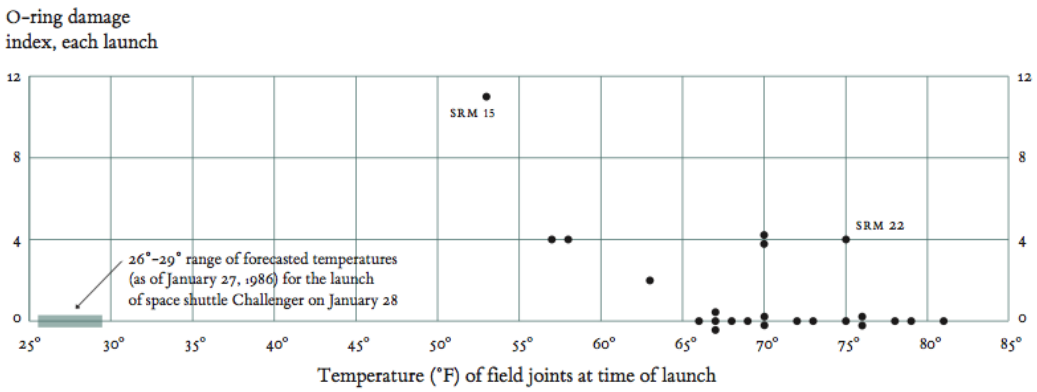
In [33]:
```
Im('./data/chall-table.png')
```

Out [33]:

Flight	Date	Temperature °F	Erosion incidents	Blow-by incidents	Damage index	Comments
51-C	01.24.85	53°	3	2	11	Most erosion any flight; blow-by; back-up rings heated.
41-B	02.03.84	57°	1		4	Deep, extensive erosion.
61-C	01.12.86	58°	1		4	O-ring erosion on launch two weeks before Challenger.
41-C	04.06.84	63°	1		2	O-rings showed signs of heating, but no damage.
1	04.12.81	66°			0	Coolest (66°) launch without O-ring problems.
6	04.04.83	67°			0	
51-A	11.08.84	67°			0	
51-D	04.12.85	67°			0	
5	11.11.82	68°			0	
3	03.22.82	69°			0	
2	11.12.81	70°	1		4	Extent of erosion not fully known.
9	11.28.83	70°			0	
41-D	08.30.84	70°	1		4	
51-G	06.17.85	70°			0	
7	06.18.83	72°			0	
8	08.30.83	73°			0	
51-B	04.29.85	75°			0	
61-A	10.30.85	75°		2	4	No erosion. Soot found behind two primary O-rings.
51-I	08.27.85	76°			0	
61-B	11.26.85	76°			0	
41-G	10.05.84	78°			0	
51-J	10.03.85	79°			0	
	06.27.82	80°			?	O-ring condition unknown; rocket casing lost at sea.
51-F	07.29.85	81°			0	

In [34]:
```
Im('./data/chall-damage.png')
```

Out [34]:



10.5 Principal Components Analysis (PCA): Image recognition and clasification

In [35]:
```
from PIL import Image
```

You are an ATM and have to distinguish between cash and check. Its based on a check/drivers license separator at the yhat blog (http://blog.yhathq.com/posts/image-classification-in-Python.html), and a fair bit of code is obtained from there. This problem is a bit more interesting as there is more structure in the images.

We standardize the size of the images:

In [36]:
```
#setup a standard image size; this will distort some images
but will get everything into the same shape
STANDARD_SIZE = (322, 137)
def img_to_matrix(filename, verbose=False):
    """
    takes a filename and turns it into a numpy array of RGB pixels
    """
    img = Image.open(filename)
    if verbose==True:
        print "changing size from %s to %s" % (str(img.size),
        str(STANDARD_SIZE))
    img = img.resize(STANDARD_SIZE)
    img = list(img.getdata())
    img = map(list, img)
    img = np.array(img)
    return img

def flatten_image(img):
    """
    takes in an (m, n) numpy array and flattens it
    into an array of shape (1, m * n)
    """
    s = img.shape[0] * img.shape[1]
    img_wide = img.reshape(1, s)
    return img_wide[0]
```

In [37]:
```
import os
checks_dir = "./data/images/images/checks/"
dollars_dir = "./data/images/images/dollars/"
def images(img_dir):
    return [img_dir+f for f in os.listdir(img_dir)]
```

```
checks=images(checks_dir)
dollars=images(dollars_dir)
images=checks+dollars
labels = ["check" for i in range(len(checks))] +
["dollar" for i in range(len(dollars))]
len(labels), len(images)
```

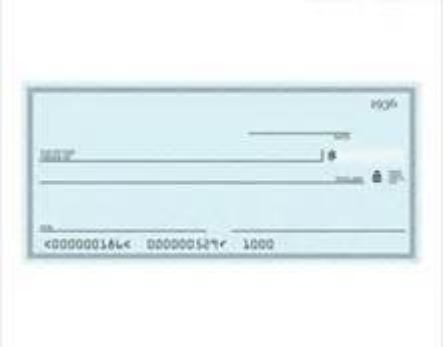
Out [37]:

```
(87, 87)
```

Lets see what some of these images look like:

In [38]:

```
for i in range(5):
    display(Im(checks[i]))
for i in range(5):
    display(Im(dollars[i]))
```




Alexander Popov
123 Main Street
Anywhere USA 98765
1198
Date
Pay to the Order of
$
Dollars
Main Street Bank
321 Main Street
Anywhere USA 98765
MEMO


Your Name
Your Address
City, State ZIP
Today's date 20 08
Payable to University of Florida
$ 26.00
Twenty-six and 00/00
dollars
Memo Subtropics
Sign Here


FF 95594731 A
F6
THE UNITED STATES OF AMERICA
100
ONE HUNDRED DOLLARS

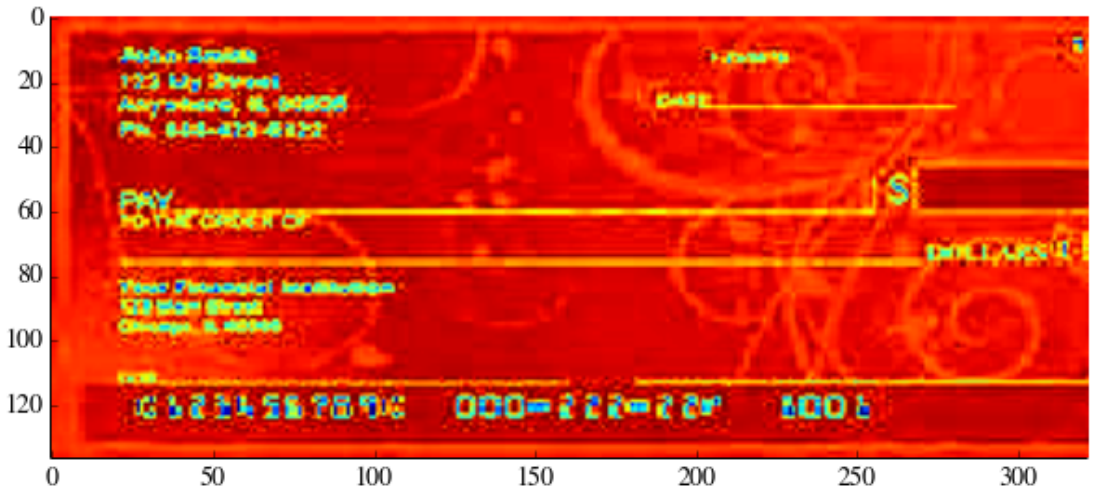

THE UNITED STATES OF AMERICA
THE CITIZENS NATIONAL BANK OF
WATERTOWN
SOUTH DAKOTA
FIFTY DOLLARS
E000065A
3349
50
FIFTY DOLLARS


THE UNITED STATES OF AMERICA
F48773146J
F
ONE DOLLAR

What features might you use to istinguish the notes from the checks?Here is an example of transforming an image into its R channel.

In [39]:
```
i0=images[20]
display(Im(i0))
i0m=img_to_matrix(i0)
print i0m.shape
plt.imshow(i0m[:,1].reshape(137,322))
remove_border()
```


```
(44114, 3)
```



We do this for every image, flattening each image into 3 channels of 44114 pixels, for a total of 132342 features per image!

In [40]:
```
data = []
for image in images:
    img = img_to_matrix(image)
    img = flatten_image(img)
    data.append(img)

data = np.array(data)
data.shape
```

Out [40]:
```
(87, 132342)
```

In [41]:
```
y = np.where(np.array(labels)=="check", 1, 0)
y.shape
```

Out [41]:
```
(87,)
```

We now carryout a 20D PCA, which captures almost the 73% of the variance.

In [42]:
```
def do_pca(d,n):
    pca = PCA(n_components=n)
    X = pca.fit_transform(d)
    print pca.explained_variance_ratio_
    return X, pca
```

In [43]:
```
from sklearn.decomposition import PCA
X20, pca20=do_pca(data,20)
```

```
[ 0.35926306  0.06293312  0.0410784   0.03119548  0.02816986
0.02288336
  0.02101291  0.01874053  0.01732656  0.0153024   0.014216
0.01318392
  0.01247011  0.01163811  0.01099578  0.01060718  0.0100742
0.00980232
```

```
           0.00960552  0.00915352]
```

In [44]:
```
np.sum(pca20.explained_variance_ratio_)
```

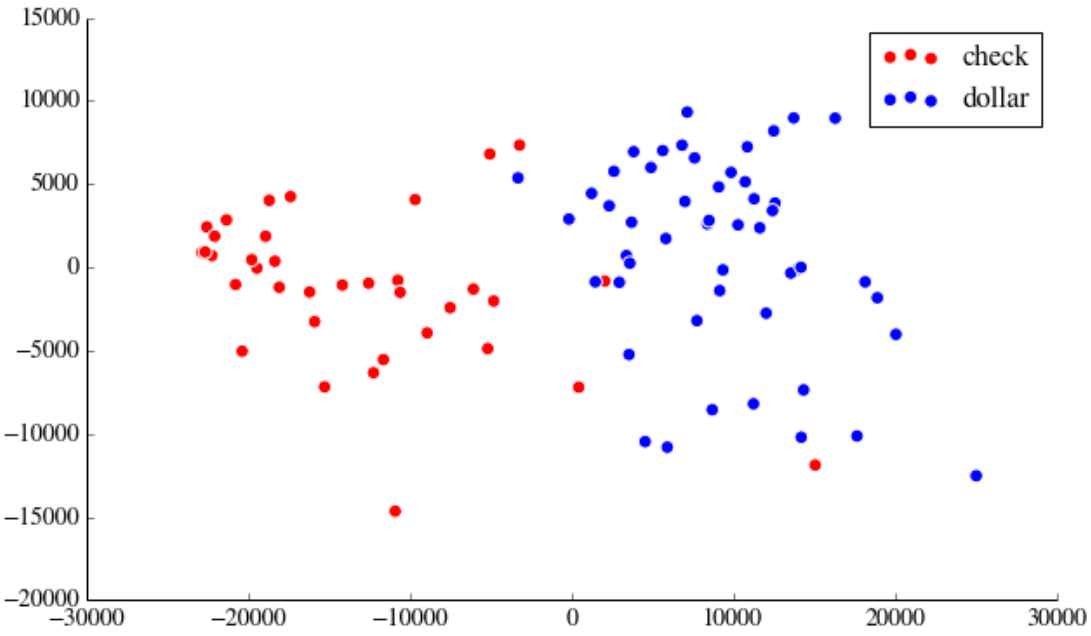
Out [44]:
```
0.72965252
```

Just for kicks, because it is easy to plot, we'll do the 2D PCA

In [45]:
```
X2, pca2 = do_pca(data,2)
```

```
[ 0.35926306  0.06293312]
```

In [82]:
```
df = pd.DataFrame({"x": X2[:, 0], "y": X2[:, 1],
 "label":np.where(y==1,
 "check", "dollar")})
colors = ["red", "blue"]
for label, color in zip(df['label'].unique(), colors):
    mask = df['label']==label
    plt.scatter(df[mask]['x'], df[mask]['y'], c=color, label=label, s=60)
plt.legend()
remove_border()
```



A quick visual shows that 2Dims may be enough to allow for linear separation of checks from dollars, with 42% of the variance accounted for. A higher number of dimensions would be usefull for face recognition, but all we want to do is to split images into two classes, so its not actually that surpring.

(For a notebook on face recognition, see:

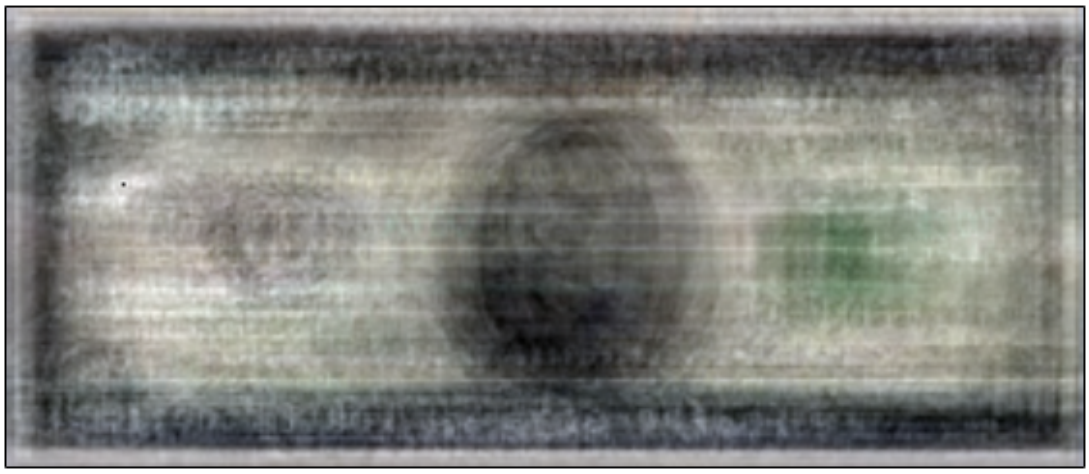
http://nbviewer.ipython.org/urls/raw.github.com/jakevdp/sklearn_scipy2013/master/rendered_notebooks/05.1 _application_to_face_recognition.ipynb.) Here below you have some code to re-construct, from the principal components, the images corresponding to them.

In [47]:
```
def normit(a):
    a=(a - a.min())/(a.max() -a.min())
    a=a*256
    return np.round(a)

def getRGB(o):
    size=322*137*3
    r=o[0:size:3]
    g=o[1:size:3]
    b=o[2:size:3]
    r=normit(r)
    g=normit(g)
    b=normit(b)
    return r,g,b

def getNC(pc, j):
    return getRGB(pc.components_[j])

def getMean(pc):
    m=pc.mean_
    return getRGB(m)

def display_from_RGB(r, g, b):
    rgbArray = np.zeros((137,322,3), 'uint8')
    rgbArray[..., 0] = r.reshape(137,322)
    rgbArray[..., 1] = g.reshape(137,322)
    rgbArray[..., 2] = b.reshape(137,322)
    img = Image.fromarray(rgbArray)
    plt.imshow(np.asarray(img))
    ax=plt.gca()
    ax.set_xticks([])
    ax.set_yticks([])
    return ax

def display_component(pc, j):
    r,g,b = getNC(pc,j)
    return display_from_RGB(r,g,b)
```

And use these to see the first two PC's. It looks like the contrast difference between the presidential head and the surroundings is the main key to doing the classifying. The second PC seems to capture general darkness.

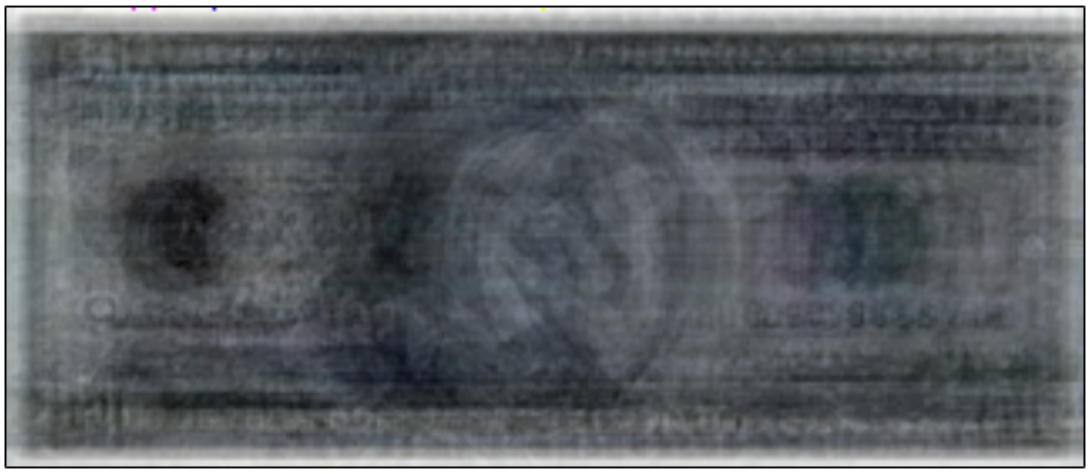
In [48]:
```
display_component(pca2,0)
```

Out [48]:
```
<matplotlib.axes.AxesSubplot at 0xcdf5810>
```



In [49]:
```
display_component(pca2,1)
```

Out [49]:
```
<matplotlib.axes.AxesSubplot at 0xcf2fb10>
```



Let us compare it with a 5 dimensional PCA, get the variance explanation, and display the components.

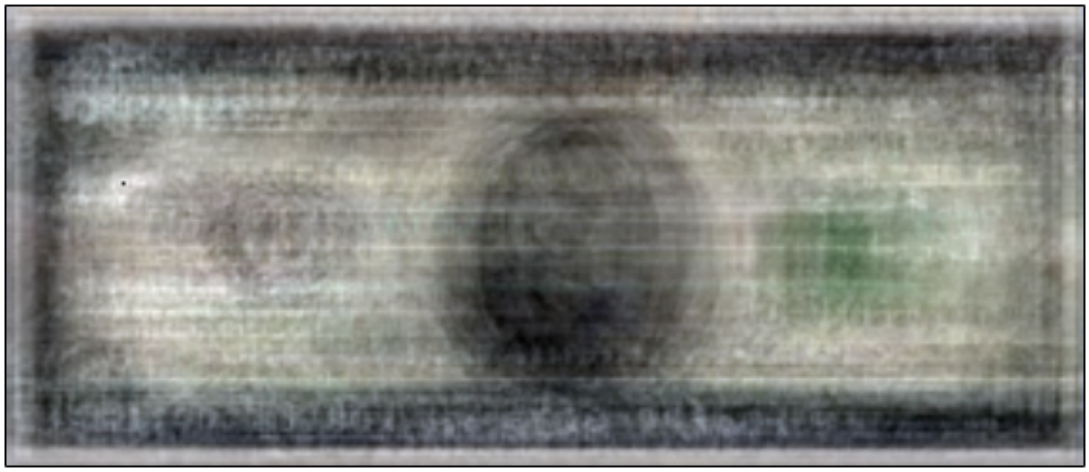
In [50]:
```
X5, pca5=do_pca(data, 5)
```

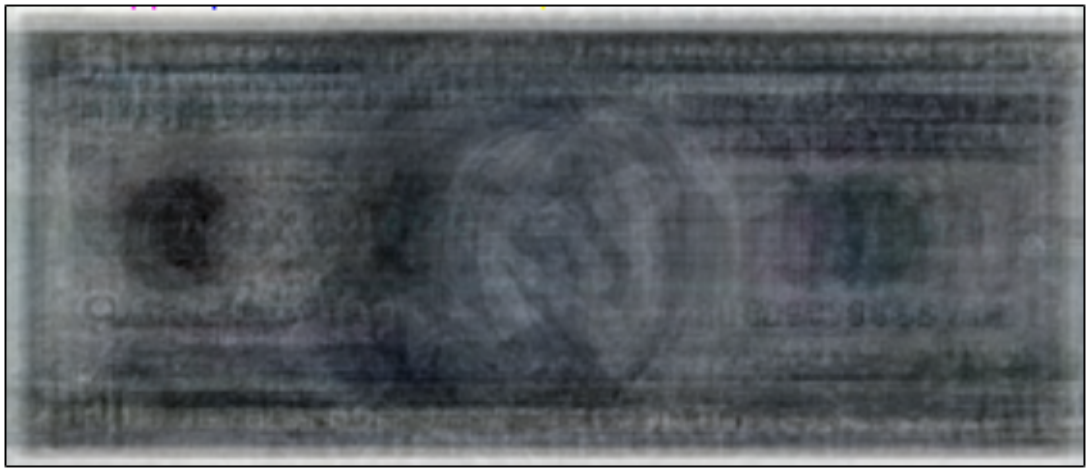
```
[ 0.35926306  0.06293312  0.0410784   0.03119548  0.02816986]
```

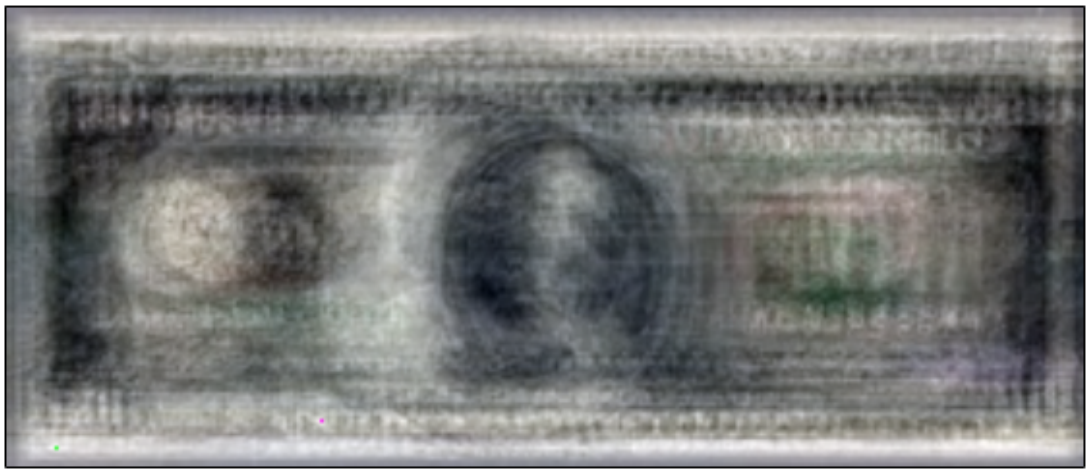
In [51]:
```
np.sum(pca5.explained_variance_ratio_)
```

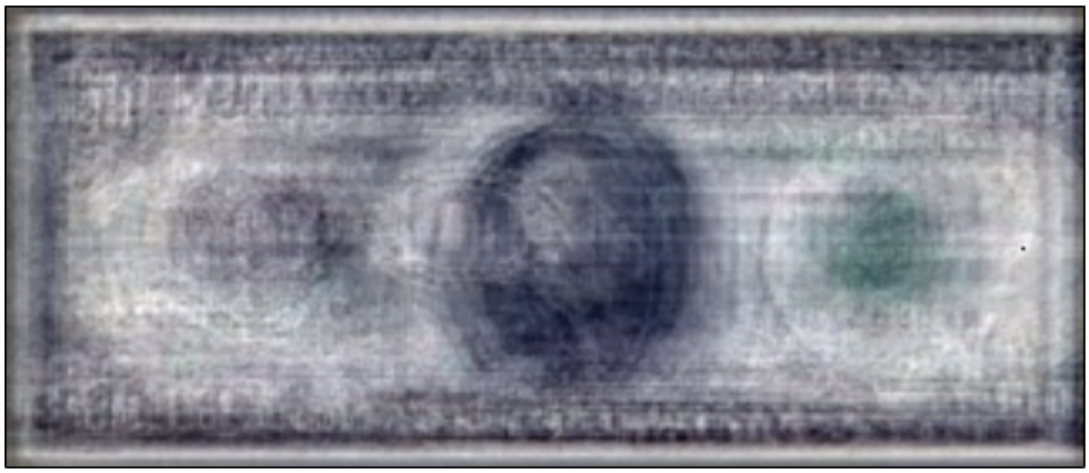
Out [51]:
```
0.52263993
```

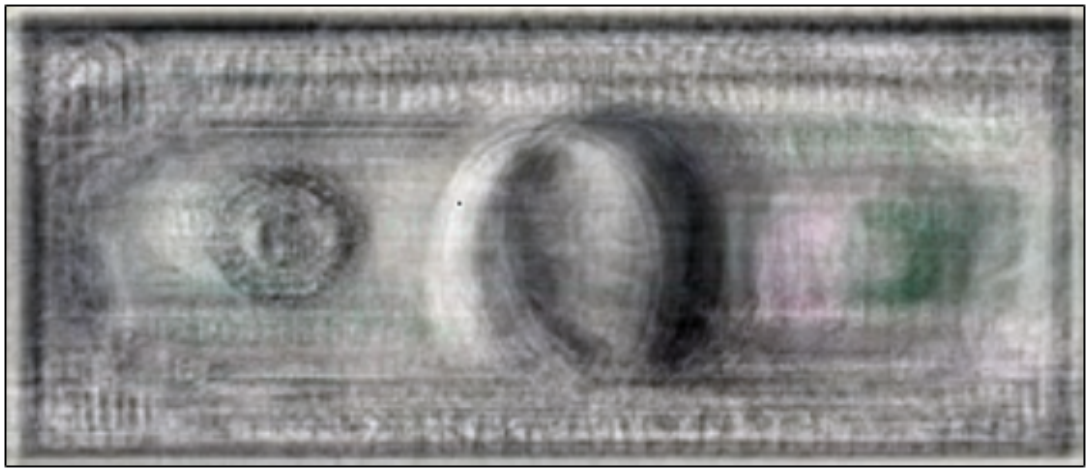
In [52]:
```
for i in range(5):
    plt.figure()
    display_component(pca5,i)
```











In [53]:
```
display_from_RGB(*getMean(pca5))
```

Out [53]:

```
<matplotlib.axes.AxesSubplot at 0xdc2fb50>
```

Using a logistic clasifier

We provide our usual code adapted from the scikit-learn web site to show classification boundaries.

In [54]:
```
from matplotlib.colors import ListedColormap

def points_plot(Xtr, Xte, ytr, yte, clf):
    X=np.concatenate((Xtr, Xte))
    h = .02
    x_min, x_max = X[:, 0].min() - .5, X[:, 0].max() + .5
    y_min, y_max = X[:, 1].min() - .5, X[:, 1].max() + .5
    xx, yy = np.meshgrid(np.linspace(x_min, x_max, 50),
                         np.linspace(y_min, y_max, 50))

    # just plot the dataset first
    cm = plt.cm.RdBu
    cm_bright = ListedColormap(['#FF0000', '#0000FF'])
    f,ax = plt.subplots()

    # Plot the training points
    ax.scatter(Xtr[:, 0], Xtr[:, 1], c=ytr, cmap=cm_bright)

    # and testing points
    ax.scatter(Xte[:, 0], Xte[:, 1], c=yte, cmap=cm_bright, marker="s",
    s=50, alpha=0.9)
    ax.set_xlim(xx.min(), xx.max())
    ax.set_ylim(yy.min(), yy.max())
    ax.set_xticks(())
    ax.set_yticks(())
    Z = clf.predict_proba(np.c_[xx.ravel(), yy.ravel()])[:, 1]
    Z = Z.reshape(xx.shape)
    ax.contourf(xx, yy, Z, cmap=cm, alpha=.4)
    cs2 = ax.contour(xx, yy, Z, cmap=cm, alpha=.4)
    plt.clabel(cs2, fmt = '%2.1f', colors = 'k', fontsize=14)
    return ax
```

Here we show a way of doing the train-test breakdown by ourselves

In [55]:
```
is_train = np.random.uniform(0, 1, len(data)) <= 0.7
train_x, train_y = data[is_train], y[is_train]
test_x, test_y = data[is_train==False], y[is_train==False]
```

We *fit (find PC's) and transform* the training data, and then use the PC's to transform the test data.

In [56]:
```
pca = PCA(n_components=2)
train_x = pca.fit_transform(train_x)
test_x = pca.transform(test_x)
```

We then do a cross-validated logistic regression. Note the large amount of the regularization. Why do you think this is the case?

In [57]:
```
logreg = cv_and_fit(train_x, train_y, np.logspace(-4, 3, num=100))
pd.crosstab(test_y, logreg.predict(test_x), rownames=["Actual"],
colnames=["Predicted"])
```

```
BP,BS {'C': 0.0001} 0.931034482759
```

Out [57]:

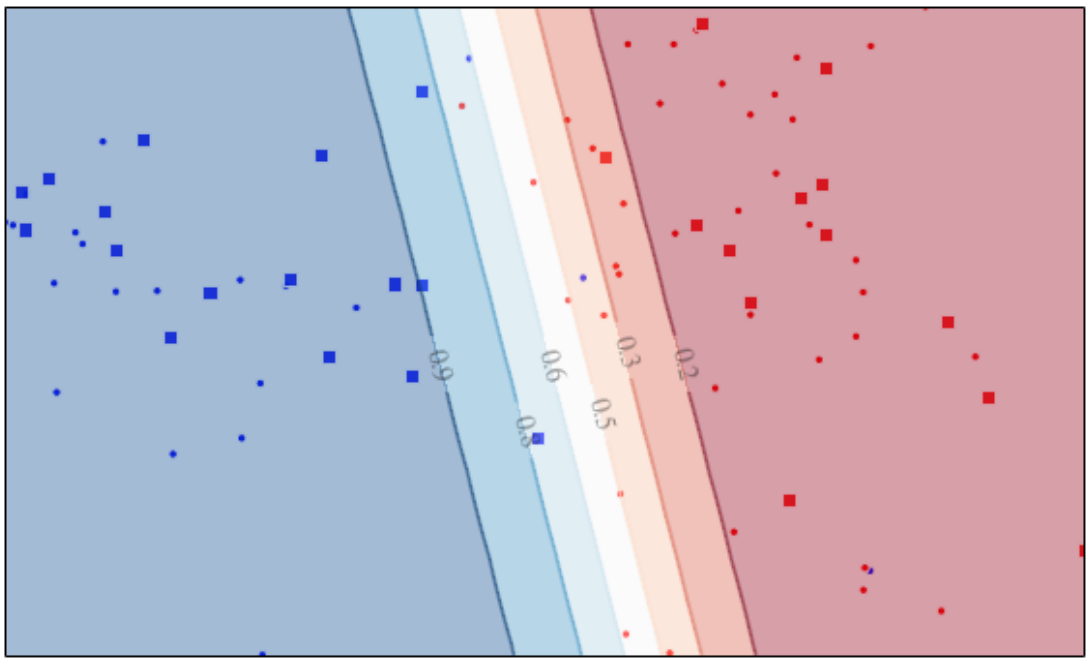
Predicted	0	1
Actual		
0	13	0
1	0	16

In [58]:
```
logreg.coef_, logreg.intercept_
```

Out [58]:
```
(array([[-0.00037867, -0.00012318]]), array([ -2.48012033e-07]))
```

In [59]:
```
points_plot(train_x, test_x, train_y, test_y, logreg)
```

Out [59]:
```
<matplotlib.axes.AxesSubplot at 0xde44fd0>
```



Lets try a L^1 penalty instead of L^2. This is strictly not a correct thing to do since PCA and L^2 regularization are both rotationally invariant. However, lets see what happen to the co-efficients.

In [60]:
```
logreg_l1=cv_and_fit(train_x, train_y, np.logspace(-4, 3, num=100),
penalty="l1")
pd.crosstab(test_y, logreg_l1.predict(test_x), rownames=["Actual"],
colnames=["Predicted"])
```

```
BP,BS {'C': 0.00022570197196339191} 0.948275862069
```

Out [60]:

Predicted	0	1
Actual		
0	13	0
1	0	16

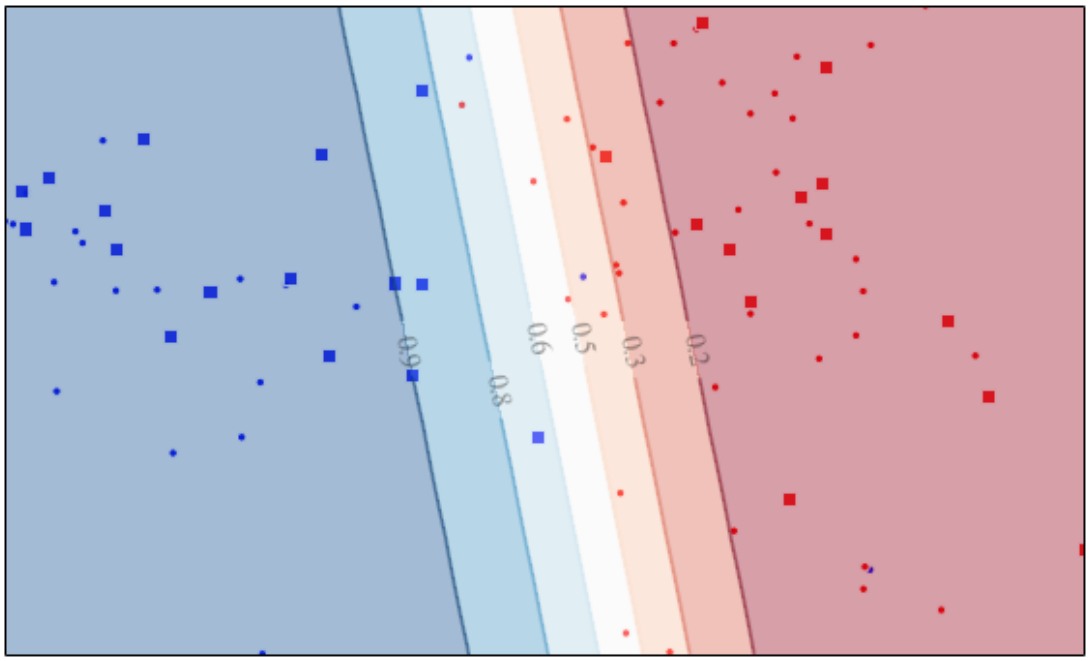
In [61]:
```
print logreg_l1.coef_, logreg_l1.intercept_
```

```
[[ -3.20574854e-04  -8.16300022e-05]] [ 0.]
```

In [62]:
```
points_plot(train_x, test_x, train_y, test_y, logreg_l1)
```

Out [62]:

```
<matplotlib.axes.AxesSubplot at 0xdc41bb0>
```



Notice L^1 regularization supresses the internet and reduces the importance of the second dimension. If one wants to minimize non-zero coefficients, one uses L^1 regularization.Let us carry out a 5 dimensional PCA and then a logistic rgression in both L^1 and L^2 modes, and let us create crosstabs and print co-efficents for both. What do you think we will find?

In [63]:
```
is_train = np.random.uniform(0, 1, len(data)) <= 0.7
train_x, train_y = data[is_train], y[is_train]
test_x, test_y = data[is_train==False], y[is_train==False]
pca = PCA(n_components=5)
train_x = pca.fit_transform(train_x)
test_x = pca.transform(test_x)
```

In [64]:
```
logreg5=cv_and_fit(train_x, train_y, np.logspace(-4, 3, num=100))
pd.crosstab(test_y, logreg5.predict(test_x), rownames=["Actual"],
colnames=["Predicted"])
```

```
BP,BS {'C': 0.0001} 0.946428571429
```

Out [64]:

Predicted	0	1
Actual		
0	19	1
1	2	9

In [65]:
```
print logreg5.coef_, logreg5.intercept_
```

```
[[-0.00215934 -0.00203808  0.00178739  0.00015102  0.000926  ]] [
-3.92628594e-07]
```

In [66]:
```
logreg5_l1=cv_and_fit(train_x, train_y, np.logspace(-5, 3, num=100),
penalty="l1")
pd.crosstab(test_y, logreg5_l1.predict(test_x), rownames=["Actual"],
colnames=["Predicted"])
```

```
BP,BS {'C': 1.0000000000000001e-05} 0.982142857143
```

Out [66]:

Predicted	0	1
Actual		
0	19	1
1	2	9

In [67]:
```
print logreg5_l1.coef_, logreg5_l1.intercept_
```

```
[[-0.00011258  0.          0.          0.          0.        ]] [ 0.]
```

10.6 A Simple Example of Neural Networks

A neural network "learns" by solving an optimization problem to choose a set of parameters that minimizes an error function, which is typically a squared error loss. This definition of learning isn't unique to a neural network model. Consider, in the simplest case, a linear model of the form $y = Ax$. Given a vector of data $y \in \mathbb{R}^m$, we choose $A \in \mathbb{R}^n$ that minimizes $||y - XA||_2$, where X is an m by n matrix with "training data" on the rows. Solving this least squares minimization problem has a nice well known closed form analytical result: $\hat{B} = (X^T X)^{-1} X^T y$. There is, however, a tradeoff between computational ease of finding optimal model parameters and model complexity. This is evident in the linear case since the model is extremely easy to fit but has a very simple form. The neural network attempts to find

a "sweet spot": while the model is highly non-linear, its particular functional form allows for a computationally slick fitting procedure called "backpropagation".

A "neural network" is a function from $f : \mathbb{R}^m \to \{-1, 1\}$. The input to each neuron is a linear combination of the outputs of each of the neurons in the lower layer. The output of the neuron is a nonlinear threshold applied to its input: it's something that maps the real line to (-1,1) in a 1-1 fashion such that "most" of the positive line is mapped pretty close to 1 and and "most" of the negative line is mapped pretty close to -1. A common choice is the function $g(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$.The final output of the signal is the sign of the top neuron. Note that the top layer is constrained to have a single neuron.

The goal of fitting the model is to find a suitable set of "weights", which we can compactly refer to as w, for the inputs of each of the neurons. One way to do this is to choose a set of weights that minimize the sum of squared errors for training examples we already have. The intuition here is to choose a model that is "close" to the true model, just like we would in a linear regression. However, unlike linear regression an analytical solution is not feasible because of the ugly threshold functions, so we need to resort to a computational approach like gradient descent. Gradient descent takes steps in the direction of greatest error decrease in the parameter space, hoping to find a (global) minimum. (Recall that from multivariable calculus, the gradient of a scalar field points in the direction of the greatest increase of the function, and so walking in the opposite direction points in the direction of greatest decrease repeating the argument with the negative of the function.)

We start by initializing the model with some arbitrary set of weights. Feeding forward, given an input $x \in \mathbb{R}^m$, we may compute the output as delineated above. Hence, we may calculate the error $E = (y - f(x))^2$. Now, we compute the gradient of this error function with respect to the weights.

The beauty of the method is that computing the gradient of the error is computationally slick and can be done recursively using the chain rule. To see this, we'll introduce some notation. Let $k = 1...L$ indicate layers, assume that s_{jk} is the output of the jth neuron in layer k, x_{jk} is the input into neuron j in layer k, and w_{ijk} is the weight for the signal input into the jth neuron in layer k coming from the ith neuron in layer $k-1$, so that $x_{jk} = \sum_{i=1}^{d_{k-1}} w_{ijk} s_{ik-1}$, $s_{jk} = g(x_{jk})$. Note that d_{k-1} stands for the number of neurons in layer $k-1$. By the chain rule,

$$\frac{\partial E}{\partial w_{ijk}} = \frac{\partial E}{\partial s_{jk}} \frac{\partial s_{jk}}{\partial w_{ijk}}$$

Since s_{jk} is linear in the weights, the tricky part is only in computing the former component of the product. We may recursively compute this as

$$\frac{\partial E}{\partial s_{jk}} = \frac{\partial E}{\partial x_{jk}} \frac{\partial x_{jk}}{\partial s_{jk}} = \sum_{i=1}^{d_{k+1}} \frac{\partial E}{\partial s_{ik+1}} \frac{\partial s_{ik+1}}{\partial x_{jk}} \frac{\partial x_{jk}}{\partial s_{jk}}$$

The recursion terminates since $\frac{\partial E}{\partial x_{1L}} = 2(x_{1L} - y)$. We start by computing the gradients from the top layer, and store the gradients as we progress down each layer,so that we need not recompute them. This leads to computational efficiency.

Thus "learning" in a neural network is nothing but switching between computing the error using the training data and updating the weights by calculating the gradient of the error function. In the code block that follows, we write a class for NeuralNetworks, and apply it to performing classification.

In [146]:
```
#Neural Network Class

class Neural_Net:

#constructor initializes a new neural network with randomly selected
weights and pre-specified height, and number of neurons per layer


def __init__(self,non,height):
```

```
#list to store the number of neurons in each layer of the network
self.num_of_neurons = non
#height of the network
self.L = height
#list to store number of weights in each layer of the network, indexed
by layer, output neuron, input neuron
self.weights = numpy.zeros(shape=(10,10,10))
#delta_matrix: stores the gradient that is used in backpropagation
self.deltas = numpy.zeros(shape=(10,10))
#matrix that stores thresholded signals
self.signals = numpy.zeros(shape=(10,10))
#(tunable) learning_rate used in backpropagation
self.learning_rate = .001
#initialize weights to be between -2 and 2

for i in range(1,self.L+1):

for j in range(1,self.num_of_neurons[i]+1):

for k in range(self.num_of_neurons[i-1]+1):

self.weights[i][j][k] = random.random()*4-2

#forward_pass computes the output of the neural network given an input

def forward_pass(self,x):

#(for convenience, we index neurons starting at 1 instead of zero)
self.signals[0][0] = -1

for i in range(1,self.num_of_neurons[0]+1):

self.signals[0][i] = x[i-1]

for i in range(1,self.L+1):

self.signals[i][0] = -1

for j in range(1,self.num_of_neurons[i]+1):

self.signals[i][j] = self.compute_signal(i,j)

return self.signals[self.L][1]

#tune_weights performs the backpropagation algorithm given a
training example as input
def tune_weights(self,y):
self.deltas[self.L][1] = 2*(self.signals[self.L][1]-y)*(1-math.pow(self.signals[self.L
for i in range(self.L-1,0,-1):
for j in range(1,self.num_of_neurons[i]+1):
self.deltas[i][j] = self.compute_delta(i,j)
for i in range(1,self.L+1):
for j in range(1,self.num_of_neurons[i]+1):
for k in range(self.num_of_neurons[i-1]+1):
self.weights[i][j][k] = self.weights[i][j][k]-self.learning_rate*self.signals[i-1][k]*

self.deltas[i][j]

#compute_signal: computes the delta for a given neuron at a given level
def compute_signal(self,level,neuron):
s = 0
for i in range(self.num_of_neurons[level-1]+1):
s += self.weights[level][neuron][i]*self.signals[level-1][i]
return self.g(s)

#compute_delta: computes the signal s for a given neuron at a given level
def compute_delta(self,level,neuron):
```

```
s = 0
for j in range(1,self.num_of_neurons[level+1]+1):
s += self.weights[level+1][j][neuron]*self.deltas[level+1][j]
return (1-math.pow(self.signals[level][neuron],2))*s

#soft threshold function
def g(self,s):
return (math.exp(s)-math.exp(-s))/(math.exp(s)+math.exp(-s))
```

Now let's train a neural network and see how well it performs on the test and training sets epoch by epoch. We will use a mock training and test set with two covariates. We instantiate a neural network with one hidden layer with four neurons, and a learning rate of .001. The learning rate is how much we scale the gradient in "walking" the parameter space.

To gain some intuition, try to tweak some of these knobs.

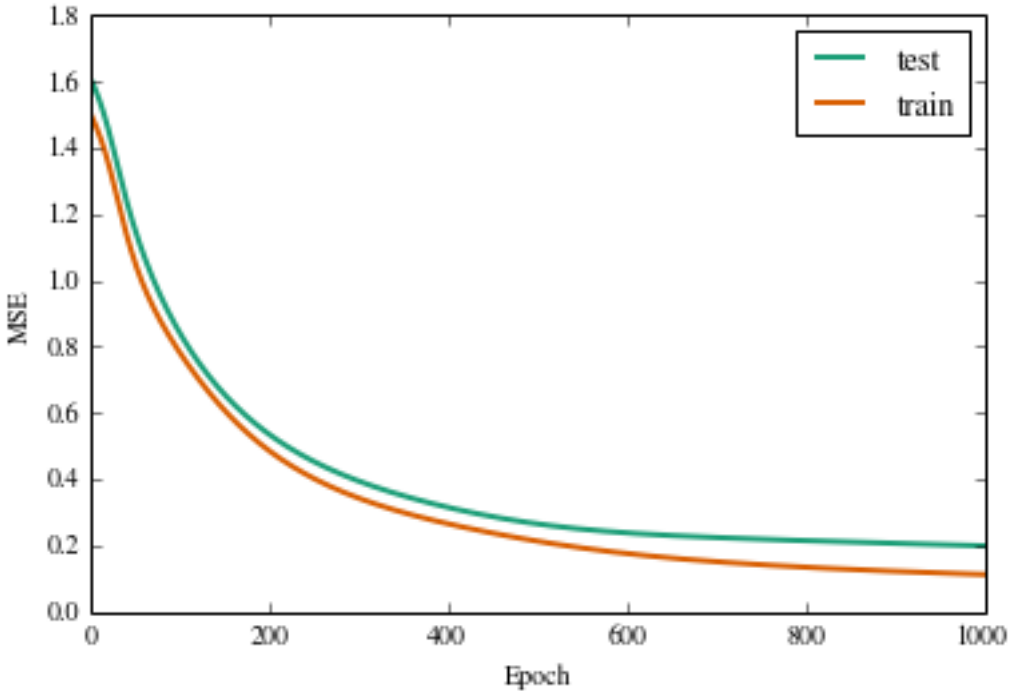
Note that this will take about a minute to run!

In [147]:
```
#read in the train and test dat, assuming csv format
training = numpy.genfromtxt('train.csv',delimiter = ',')
testing = numpy.genfromtxt('test.csv',delimiter = ',')

#specify the number of neurons in each layer
num_of_neurons = [2,4,1]

#initialize a new neural network
network = Neural_Net(num_of_neurons,2)

#store the training error and test error during each epoch
training_error = 0
test_error = 0

#store the training and test error for all epochs
train = numpy.zeros(shape = (1000))
test = numpy.zeros(shape = (1000))

for epoch in range(1000):
training_error = 0
test_error = 0
#compute the test errors
for j in range(250):
test_error = test_error+math.pow(network.forward_pass(testing[j]) - testing[j][2], 2)
#compute the training errors, SEQUENTIALLY. In other words, we perform backpropagation
#instead of all at once.
for i in range(25):
training_error = training_error+math.pow(network.forward_pass(training[i])- training[i
network.tune_weights(training[i][2])
training_error = training_error/25
test_error = test_error/250
train[epoch] = training_error
test[epoch]  = test_error
```

Now let's compare the training and test error, epoch by epoch. Do we see signs of overfitting?

In [151]:
```
fig, ax = plt.subplots()
ax.plot(numpy.arange(1000), test, lw=2, label = 'test')
ax.plot(numpy.arange(1000), train, lw=2, label = 'train')
ax.legend(loc=0)
ax.set_xlabel('Epoch')
ax.set_ylabel('MSE')
```

```
Out [151]:
          <matplotlib.text.Text at 0x10f9c17d0>
```



10.7 Support Vector Machines: Background and Visual Intuition

Assume your data set D is = $(x_1, y_1), \ldots, (x_n, y_n)$, where $x_i \in \mathbb{R}^k$ and $y_i \in \{1, -1\}$. A support vector machine attempts to find a plane in $\mathbb{R}^k$, that is, a "hyperplane", that separates the data of either class. While there may exist an infinite number of planes that achieve this goal (or perhaps none), intuitively we want the one which maximizes the distance between the separating hyperplane and the closest vectors, termed the "support vectors". The hope is that this will allow the model to generalize well. It turns out this condition can be codified as a quadratic programming (optimization) problem.

Denote an arbitrary hyperplane in $\mathbb{R}^k$ as $w^t x + b = 0$. The support vector machine is the solution to the constrained optimization problem:

$$minimize$$

$$w^t w$$

Subject to constraint:

$$(w^t x_i) + b \geq 1, y_i = 1$$

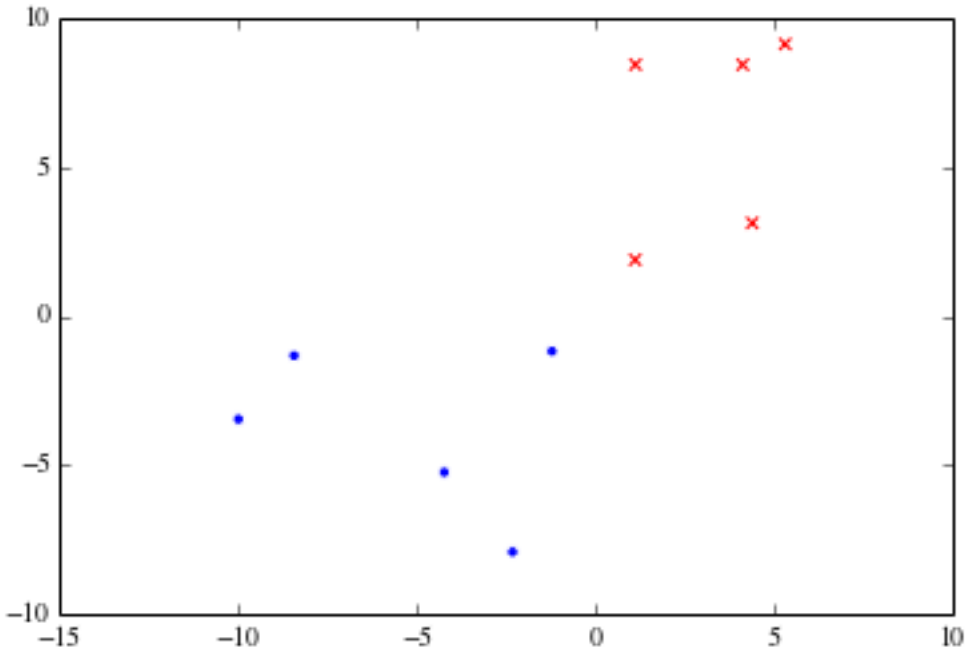
$$(w^t x_i) + b \leq -1, y_i = -1$$

Note that this method is useful only in the case that the data appear to be linearly separable to begin with. In the case they are not, we apply a "Kernel" function to the data which warps the points with the hope that they become linearly separable. There are a variety of kernels that one could use to achieve this goal, and scikit-learn supports a number of them.

In the following example, we visualize a small mock data set to gain some intuition before applying the SVM method of classification.

In [152]:
```
#read in svm data
svm_dat = numpy.genfromtxt('svm.csv',delimiter = ',')
class1 = svm_dat[0:5]
class2 = svm_dat[5:10]
plt.figure()
plt.scatter(class1[:,0],class1[:,1], c = 'blue', marker = 'o')
plt.scatter(class2[:,0],class2[:,1], c = 'red', marker = 'x')
```

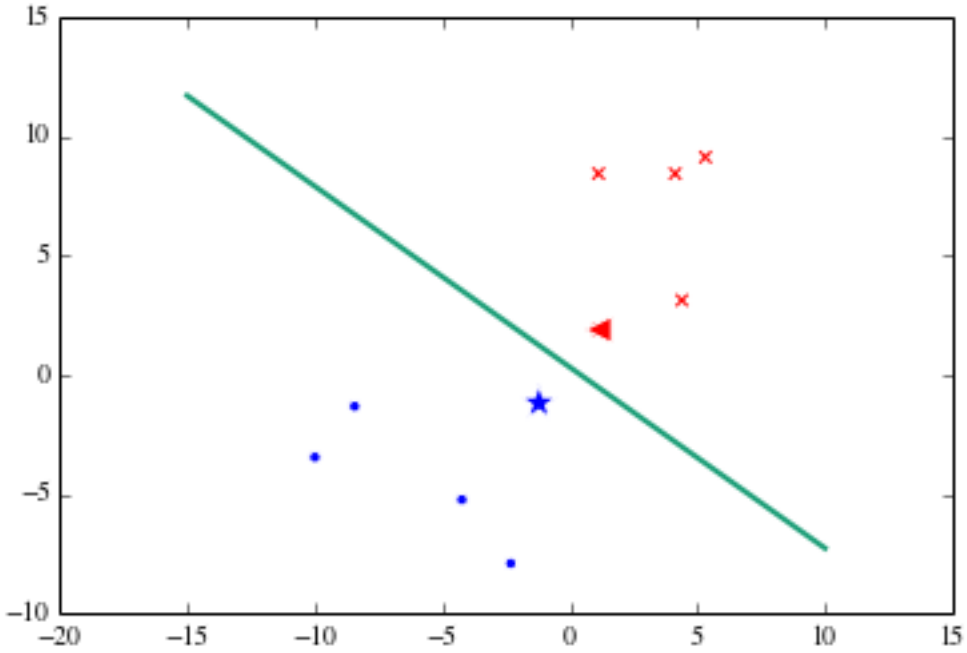
Out [152]:
```
<matplotlib.collections.PathCollection at 0x10fc75ad0>
```



Can you guess what the support vectors are in this data set? (Hopefully it is easy to see.) Let's actually fit a SVM to check our suspicions.

In [155]:
```
from sklearn import svm
X = np.column_stack((svm_dat[:,0],svm_dat[:,1]))
y = svm_dat[:,2]
clf = svm.SVC(kernel='linear')
clf.fit(X, y)
clf.support_vectors_
plt.scatter(class1[:,0],class1[:,1], c = 'blue', marker = 'o',s=20)
plt.scatter(class2[:,0],class2[:,1], c = 'red', marker = 'x',s=20)
plt.scatter(clf.support_vectors_[0,0],clf.support_vectors_[0,1],c='blue',marker = '*',
plt.scatter(clf.support_vectors_[1,0],clf.support_vectors_[1,1],c='red',marker = '<',
#Get the separating hyperplane
w = clf.coef_[0]
a = -w[0] / w[1]
xx = np.linspace(-15, 10)
yy = a * xx - (clf.intercept_[0]) / w[1]
t = numpy.linspace(-15,10,100) # 100 linearly spaced numbers
plt.plot(xx,yy)
```

```
Out [155]:
          [<matplotlib.lines.Line2D at 0x10ffdfa50>]
```



As you can see, there are precisely two support vectors from the data set of 10 vectors with one from each class. To play with this example, you may want to consider other kernels for data sets that are not inherently linearly separable like this example. Scikit-learn's documentation expounds on various choices of kernel.

CHAPTER

ELEVEN

A TIME-SERIES MODELLING WITH STATSMODEL

11.1 Some Basics before to start

Box-Jenkins Analysis refers to a systematic method of identifying, fitting, checking, and using integrated autoregressive, moving average (ARIMA) time series models. The method is appropriate for time series of medium to long length (at least 50 observations).

A time series is a set of values observed sequentially through time.The series may be denoted by $X_1, X_2, \ldots, X_t$, where t refers to the time period and X refers to the value. If the X's are exactly determined by a mathematical formula, the series is said to be deterministic. If future values can be described only by their probability distribution, the series is said to be a statistical or stochastic process.

A special class of stochastic processes is a stationary stochastic process. A statistical process is stationary if the probability distribution is the same for all starting values of t. This implies that the mean and variance are constant for all values of t. A series that exhibits a simple trend is not stationary because the values of the series depend on t. A stationary stochastic process is completely defined by its mean, variance, and autocorrelation function. One of the steps in the Box-Jenkins method is to transform a non-stationary series into a stationary one.

The **ARMA (autoregressive, moving average)** model is defined as follows:

$X_t =_1 X_{t-1} + ... +_p X_{t-p} + a_t -_1 a_{t-1} - ... -_q a_{t-q}$

where the 's (phis) are the autoregressive parameters to be estimated, the 's (thetas) are the moving average parameters to be estimated, the X's are the original series, and the a's are a series of unknown random errors (or residuals) which are assumed to follow the normal probability distribution.

The **Box-Jenkins** method refers to the iterative application of the following three steps:

1. **Identification**: Using plots of the data, autocorrelations, partial autocorrelations, and other information, a class of simple ARIMA models is selected. This amounts to estimating appropriate values for p, d, and q.
2. **Estimation**: The and of the selected model are estimated using maximum likelihood techniques, backcasting, etc., as outlined in Box-Jenkins (1976).
3. **Diagnostic Checking**: The fitted model is checked for inadequacies by considering the autocorrelations of the residual series (the series of residual, or error, values).

These steps are applied iteratively until step three does not produce any improvement in the model. We will now go over these steps in detail.

Statsmodels is a Python module, included in Anaconda 1.3, that allows users to explore data, estimate statistical models, and perform statistical tests. An extensive list of descriptive statistics, statistical tests, plotting functions, and result statistics are available for different types of data and each estimator. Features include:

- Linear regression models
- Generalized linear models
- Discrete choice models
- Robust linear models
- Many models and functions for time series analysis
- Nonparametric estimators
- A collection of datasets for examples
- A wide range of statistical tests
- Input-output tools for producing tables in a number of formats (Text, LaTex, HTML) and for reading Stata files into NumPy and Pandas.
- Plotting functions
- Extensive unit tests to ensure correctness of results
- Many more models and extensions in development

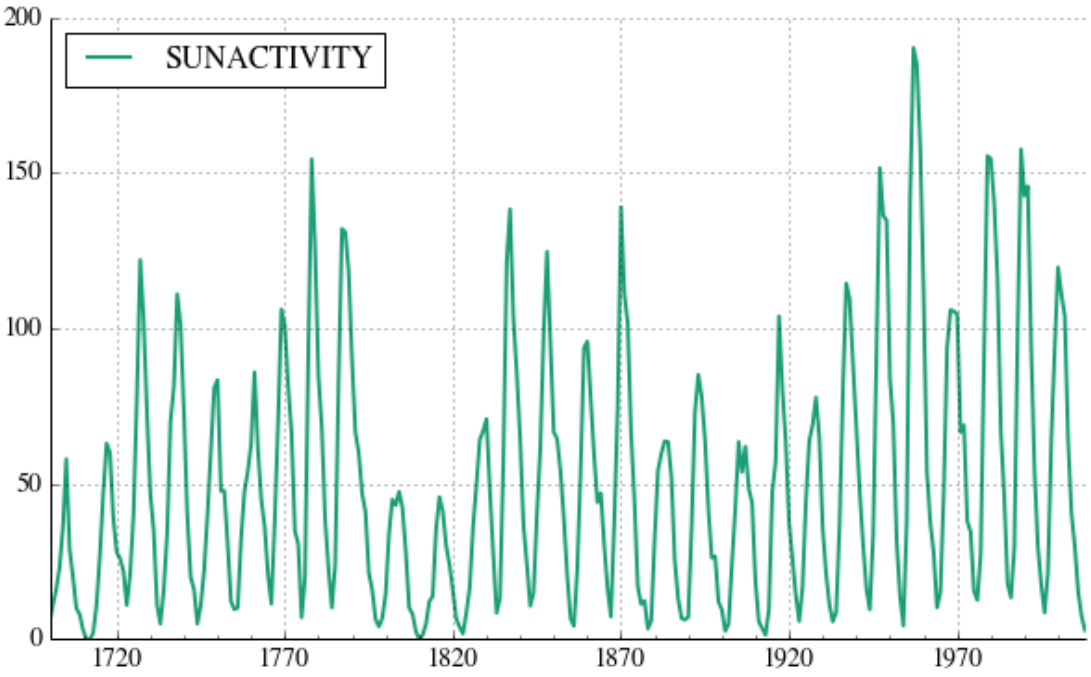
11.2 A Time Series case with Statsmodels: The Sunspots Box-Jenkins example

In [68]:
```
import statsmodels.api as sm
from statsmodels.graphics.api import qqplot
```

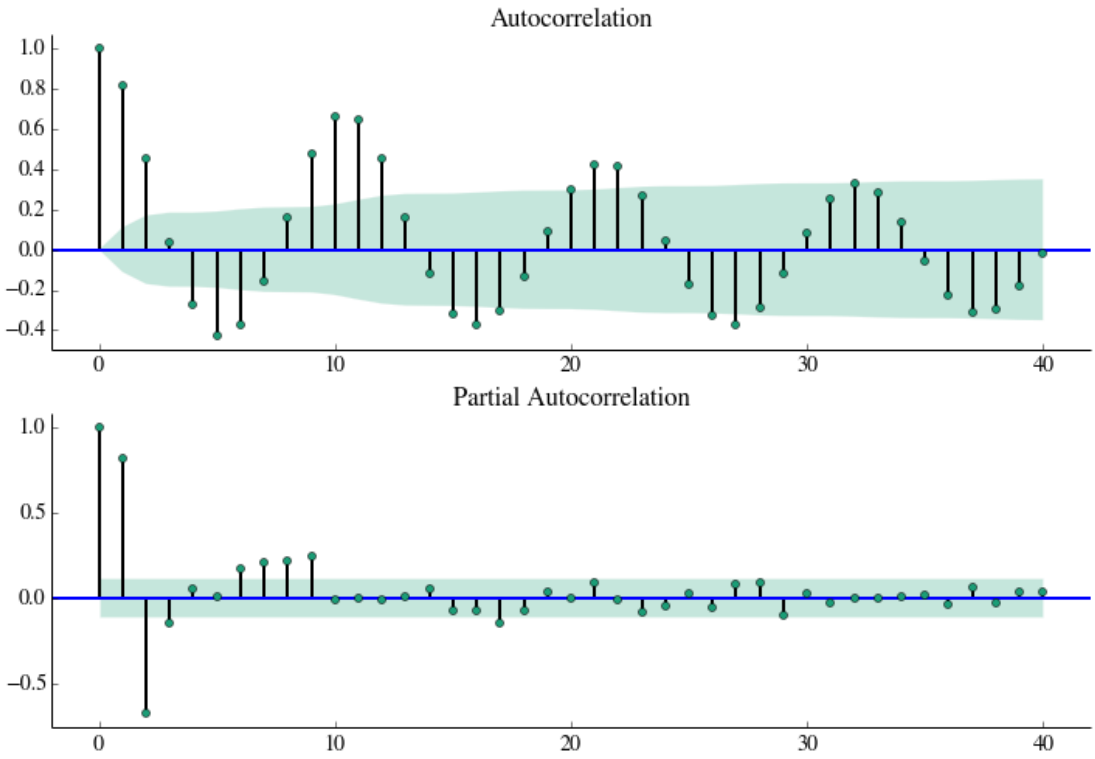
In [69]:
```
print sm.datasets.sunspots.NOTE
```

```
Number of Observations - 309 (Annual 1700 - 2008)
Number of Variables - 1
Variable name definitions::

    SUNACTIVITY - Number of sunspots for each year

The data file contains a 'YEAR' variable that is not returned by load.
```

In [70]:
```
dta = sm.datasets.sunspots.load_pandas().data
dta.index = pd.Index(sm.tsa.datetools.dates_from_range('1700', '2008'))
# print dta.head(10)
del dta["YEAR"]
dta.plot()
remove_border()
```



In [71]:
```
fig = plt.figure(figsize=(12,8))
ax1 = fig.add_subplot(211)
remove_border()
fig = sm.graphics.tsa.plot_acf(dta.values.squeeze(), lags=40, ax=ax1)
ax2 = fig.add_subplot(212)
fig = sm.graphics.tsa.plot_pacf(dta, lags=40, ax=ax2)
remove_border()
```



In [72]:
```
arma_mod20 = sm.tsa.ARMA(dta, (2,0)).fit()
print arma_mod20.params
print '-'*50
print arma_mod20.aic, arma_mod20.bic, arma_mod20.hqic
```

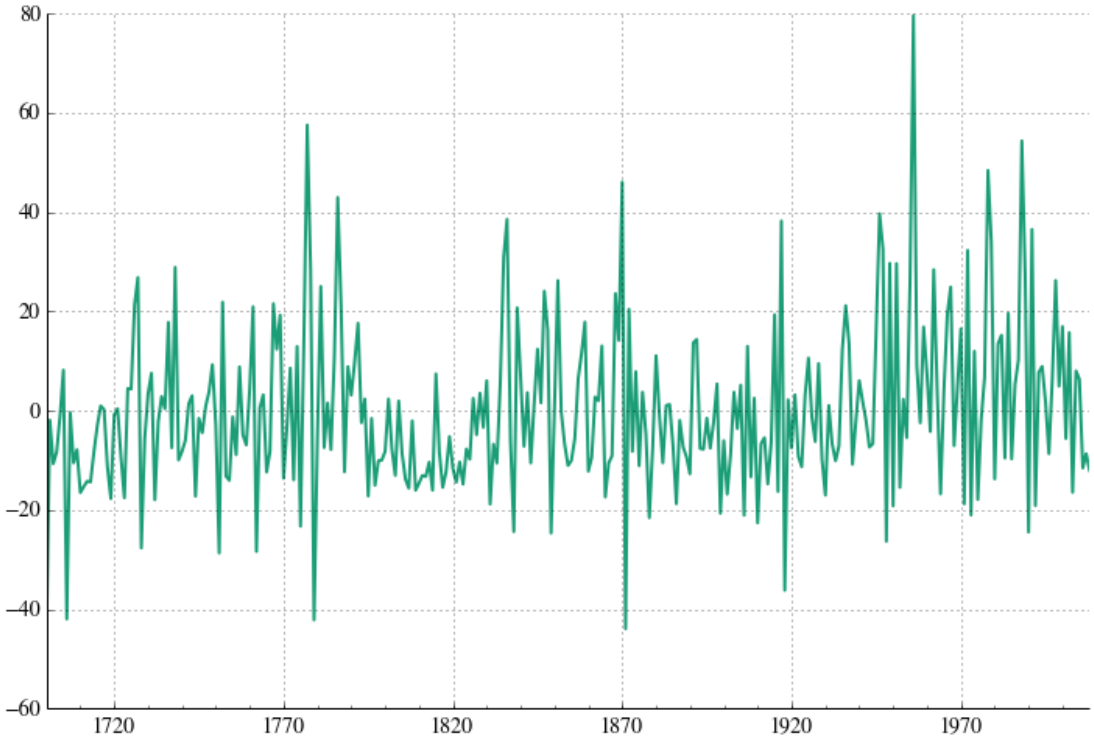
```
const                49.659306
ar.L1.SUNACTIVITY     1.390655
ar.L2.SUNACTIVITY    -0.688571
dtype: float64
--------------------------------------------------
2622.63633806 2637.56970317 2628.60672591
```

In [73]:
```
arma_mod30 = sm.tsa.ARMA(dta, (3,0)).fit()
print arma_mod30.params
print '-'*50
print arma_mod30.aic, arma_mod30.bic, arma_mod30.hqic
```

```
const                49.750059
ar.L1.SUNACTIVITY     1.300810
ar.L2.SUNACTIVITY    -0.508093
ar.L3.SUNACTIVITY    -0.129650
dtype: float64
--------------------------------------------------
2619.4036287 2638.07033508 2626.8666135
```

In [74]:
```
fig = plt.figure(figsize=(12,8))
```

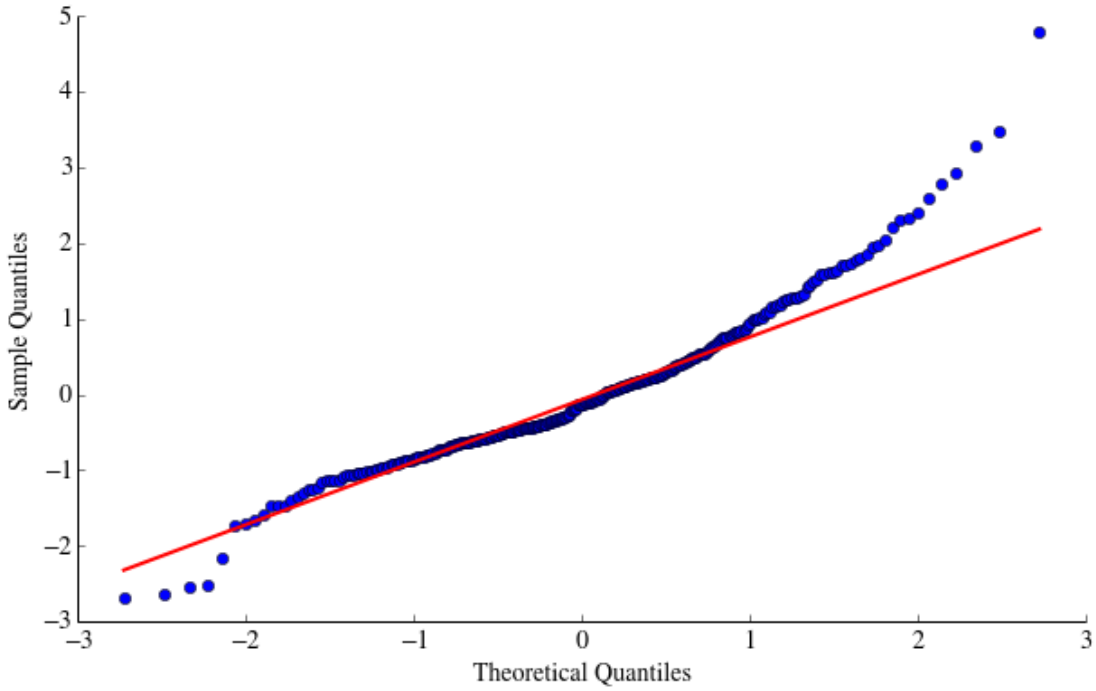
```
ax = fig.add_subplot(111)
ax = arma_mod30.resid.plot(ax=ax);
remove_border()
```



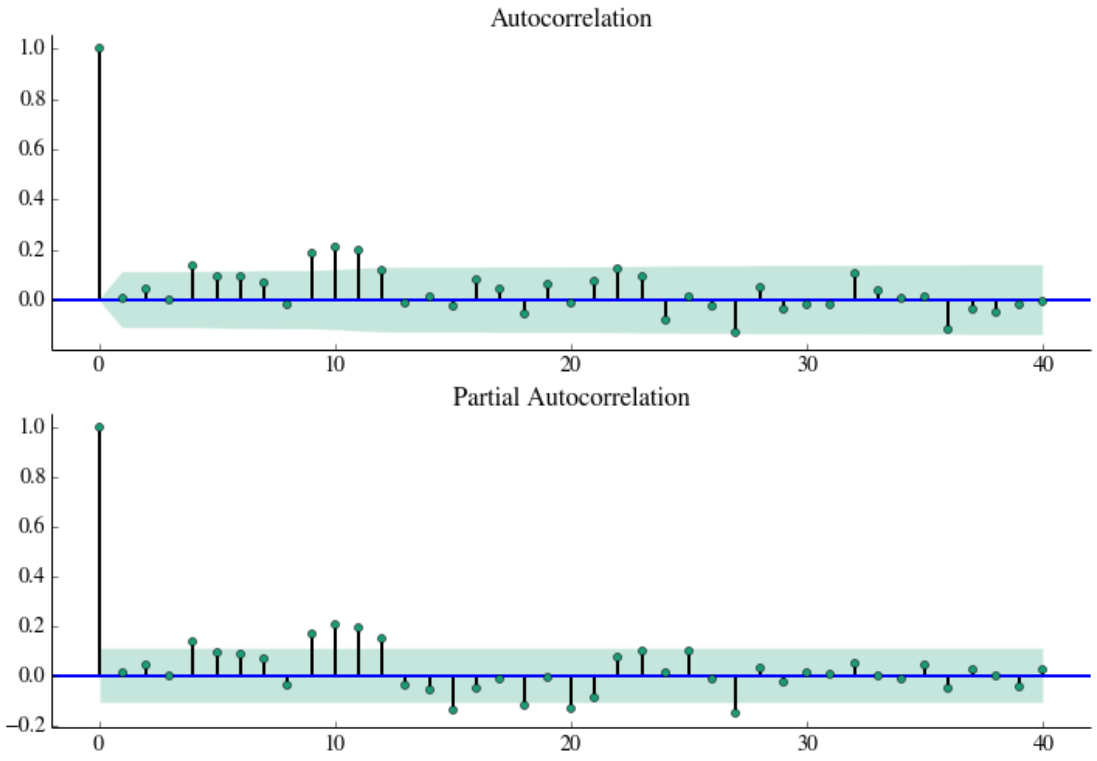
In [75]:
```
resid = arma_mod30.resid
stats.normaltest(resid)
```

Out [75]:
```
(49.845004290903262, 1.5007033188963596e-11)
```

In [76]:
```
fig = plt.figure(figsize=(10,6))
ax = fig.add_subplot(111)
fig = qqplot(resid, line='q', ax=ax, fit=True)
remove_border()
```



In [77]:
```
fig = plt.figure(figsize=(12,8))
ax1 = fig.add_subplot(211)
remove_border()
fig = sm.graphics.tsa.plot_acf(resid.values.squeeze(), lags=40, ax=ax1)
ax2 = fig.add_subplot(212)
fig = sm.graphics.tsa.plot_pacf(resid, lags=40, ax=ax2)
remove_border()
```


Autocorrelation
1.0
0.8
0.6
0.4
0.2
0.0
0
10
20
30
40
Partial Autocorrelation
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
0
10
20
30
40

In []:

CHAPTER

TWELVE

FURTHER READING RELATED WITH PYTHON PROGRAMMING

- http://www.python.org - The official web page of the Python programming language.
- http://www.python.org/dev/peps/pep-0008 - Style guide for Python programming. Highly recommended.
- http://www.greenteapress.com/thinkpython/ - A free book on Python programming.
- Python Essential Reference - A good reference book on Python programming.