Python for Athena

Exercises and Solutions

Enthought, Inc.

CONTENTS

1 Exercises **1**

1.1 Python Language Exercises . . . 1
- 1.1.1 Vote String . . . 1
- 1.1.2 Sort Words . . . 1
- 1.1.3 Roman Dictionary . . . 1
- 1.1.4 Filter Words . . . 2
- 1.1.5 Inventory . . . 2
- 1.1.6 Financial Module . . . 2
- 1.1.7 ASCII Log File . . . 3
- 1.1.8 Person Class . . . 4
- 1.1.9 Shape Inheritance . . . 4
- 1.1.10 Particle Class . . . 5
- 1.1.11 Near Star Catalog . . . 5
- 1.1.12 Generators . . . 6

1.2 Python Libraries Exercises . . . 7
- 1.2.1 Column Stats (subprocess) . . . 7
- 1.2.2 Dow Database . . . 7

1.3 Software Craftsmanship Exercises . . . 8
- 1.3.1 Test Driven Development Example . . . 8
- 1.3.2 Code Check . . . 8

1.4 Numpy Exercises . . . 9
- 1.4.1 Plotting . . . 9
- 1.4.2 Calculate Derivative . . . 10
- 1.4.3 Load Array from Text File . . . 10
- 1.4.4 Filter Image . . . 11
- 1.4.5 Dow Selection . . . 11
- 1.4.6 Wind Statistics . . . 12
- 1.4.7 Sinc Function . . . 13
- 1.4.8 Structured Array . . . 13

1.5 Scipy Exercises . . . 14
- 1.5.1 Statistics Functions . . . 14
- 1.5.2 Monte Carlo Options . . . 14
- 1.5.3 Gaussian Kernel Density Estimation . . . 15
- 1.5.4 Estimate Volatility . . . 15
- 1.5.5 Using fsolve . . . 16
- 1.5.6 Black-Scholes Pricing . . . 17
- 1.5.7 Black-Scholes Implied Volatility . . . 17
- 1.5.8 Data Fitting . . . 18

1.6 Interface With Other Languages Exercises . . . 18

1.6.1 Fancy science (hand wrap) 18
1.6.2 Blitz Inline Comparison 19
1.6.3 Weave Comparison 20
1.6.4 Mandelbrot with Weave 20
1.6.5 Mandelbrot Cython 20
1.7 Ipython Exercises 21
1.7.1 Mandelbrot Parallelized 21

2 Solutions **23**
2.1 Python Language - Solutions 23
2.1.1 Vote String - Solution 23
2.1.2 Sort Words - Solution 24
2.1.3 Roman Dictionary - Solution 24
2.1.4 Filter Words - Solution 26
2.1.5 Inventory - Solution 26
2.1.6 Financial Module - Solution 28
2.1.7 ASCII Log File - Solution 1 30
2.1.8 ASCII Log File - Solution 2 31
2.1.9 Person Class - Solution 33
2.1.10 Shape Inheritance - Solution 33
2.1.11 Particle Class - Solution 36
2.1.12 Near Star Catalog - Solution 38
2.1.13 Generators - Solution 41
2.2 Python Libraries - Solutions 42
2.2.1 Column Stats (subprocess) - Solution 1 42
2.2.2 Column Stats (subprocess) - Solution 2 44
2.2.3 Dow Database - Solution 45
2.3 Software Craftsmanship - Solutions 46
2.3.1 Test Driven Development Example - Solution 1 46
2.3.2 Code Check - Solution 47
2.4 Numpy - Solutions 49
2.4.1 Plotting - Solution 49
2.4.2 Calculate Derivative - Solution 50
2.4.3 Load Array from Text File - Solution 51
2.4.4 Filter Image - Solution 53
2.4.5 Dow Selection - Solution 54
2.4.6 Wind Statistics - Solution 55
2.4.7 Sinc Function - Solution 59
2.4.8 Structured Array - Solution 59
2.5 Scipy - Solutions 61
2.5.1 Statistics Functions - Solution 61
2.5.2 Monte Carlo Options - Solution 63
2.5.3 Gaussian Kernel Density Estimation - Solution 66
2.5.4 Estimate Volatility - Solution 67
2.5.5 Using fsolve - Solution 70
2.5.6 Black-Scholes Pricing - Solution 72
2.5.7 Black-Scholes Implied Volatility - Solution 74
2.5.8 Data Fitting - Solution 76
2.6 Interface With Other Languages - Solutions 78
2.6.1 Fancy science (hand wrap) - Solution 78
2.6.2 Blitz Inline Comparison - Solution 79
2.6.3 Weave Comparison - Solution 81
2.6.4 Mandelbrot with Weave - Solution 82
2.6.5 Mandelbrot Cython - Solution 1 84

2.6.6 Mandelbrot Cython - Solution 2 85
2.6.7 Mandelbrot Cython - Solution 3 85
2.7 Ipython - Solutions 85
2.7.1 Mandelbrot Parallelized - Solution 85

CHAPTER

ONE

EXERCISES

1.1 Python Language Exercises

1.1.1 Vote String

You have the string below, which is a set of "yes/no" votes, where "y" or "Y" means yes and "n" or "N" means no. Determine the percentages of yes and no votes.

```
votes = "y y n N Y Y n n N y Y n Y"
```

See *Vote String - Solution*.

1.1.2 Sort Words

Given a (partial) sentence from a speech, print out a list of the words in the sentence in alphabetical order. Also print out just the first two words and the last two words in the sorted list.

```
speech = '''Four score and seven years ago our fathers brought forth
         on this continent a new nation, conceived in Liberty, and
         dedicated to the proposition that all men are created equal.
         '''
```

Ignore case and punctuation.

See *Sort Words - Solution*.

1.1.3 Roman Dictionary

Mark Antony keeps a list of the people he knows in several dictionaries based on their relationship to him:

```
friends = {'julius': '100 via apian', 'cleopatra': '000 pyramid parkway'}
romans = dict(brutus='234 via tratorium', cassius='111 aqueduct lane')
countrymen = dict([('plebius','786 via bunius'),
                   ('plebia', '786 via bunius')])
```

1. Print out the names for all of Antony's friends.
2. Now all of their addresses.
3. Now print them as "pairs".
4. Hmmm. Something unfortunate befell Julius. Remove him from the friends list.

5. Antony needs to mail everyone for his second-triumvirate party. Make a single dictionary containing everyone.
6. Antony's stopping over in Egypt and wants to swing by Cleopatra's place while he is there. Get her address.
7. The barbarian hordes have invaded and destroyed all of Rome. Clear out everyone from the dictionary.

See *Roman Dictionary - Solution*.

1.1.4 Filter Words

Print out only words that start with "o", ignoring case:

```
lyrics = '''My Bonnie lies over the ocean.
            My Bonnie lies over the sea.
            My Bonnie lies over the ocean.
            Oh bring back my Bonnie to me.
            '''
```

Bonus points: print out words only once.

See *Filter Words - Solution*.

1.1.5 Inventory

Calculate and report the current inventory in a warehouse.

Assume the warehouse is initially empty.

The string, warehouse_log, is a stream of deliveries to and shipments from a warehouse. Each line represents a single transaction for a part with the number of parts delivered or shipped. It has the form:

```
part_id count
```

If "count" is positive, then it is a delivery to the warehouse. If it is negative, it is a shipment from the warehouse.

See *Inventory - Solution*.

1.1.6 Financial Module

Background

The future value (fv) of money is related to the present value (pv) of money and any recurring payments (pmt) through the equation:

```
fv + pv*(1+r)**n + pmt*(1+r*when)/r * ((1+r)**n - 1) = 0
```

or, when r == 0:

```
fv + pv + pmt*n == 0
```

Both of these equations assume the convention that cash in-flows are positive and cash out-flows are negative. The additional variables in these equations are:

- n: number of periods for recurring payments
- r: interest rate per period
- when: When payments are made:
  - 1. for beginning of the period

  – 0. for the end of the period

The interest calculations are made through the end of the final period regardless of when payments are due.

Problem

Take the script financial_calcs.py and use it to construct a module with separate functions that can perform the needed calculations with arbitrary inputs to solve general problems based on the time value of money equation given above.

Use keyword arguments in your functions to provide common default inputs where appropriate.

Bonus

1. Document your functions.
2. Add a function that calculates the number of periods from the other variables.
3. Add a function that calculates the rate from the other variables.

See *Financial Module - Solution*.

1.1.7 ASCII Log File

Read in a set of logs from an ASCII file.

Read in the logs found in the file *short_logs.crv*. The logs are arranged as follows:

```
DEPTH     S-SONIC     P-SONIC ...
8922.0    171.7472    86.5657
8922.5    171.7398    86.5638
8923.0    171.7325    86.5619
8923.5    171.7287    86.5600
...
```

So the first line is a list of log names for each column of numbers. The columns are the log values for the given log.

Make a dictionary with keys as the log names and values as the log data:

```
>>> logs['DEPTH']
[8922.0, 8922.5, 8923.0, ...]
>>> logs['S-SONIC']
[171.7472, 171.7398, 171.7325, ...]
```

Bonus

Time your example using:

```
run -t 'ascii_log_file.py'
```

And see if you can come up with a faster solution. You may want to try the *long_logs.crv* file in this same directory for timing, as it is much larger than the *short_logs.crv* file. As a hint, reading the data portion of the array in at one time combined with strided slicing of lists is useful here.

Bonus Bonus

Make your example a function so that it can be used in later parts of the class to read in log files:

```
def read_crv(file_name):
    ...
```

Copy it to the class_lib directory so that it is callable by all your other scripts.

See *ASCII Log File - Solution 1* and *ASCII Log File - Solution 2*.

1.1.8 Person Class

Write a class that represents a person with first and last name that can be initialized like so:

```
p = Person("eric", "jones")
```

Write a class method that returns the person's full name.

Write a __repr__ method that prints out a "nice" representation of the person:

```
Person("eric jones")
```

See *Person Class - Solution*.

1.1.9 Shape Inheritance

In this exercise, you will use class inheritance to define a few different shapes and draw them onto an image. All the classes will derive from a single base class, Shape, that is defined for you. The Shape base class requires two arguments, x and y, that are the position of the shape in the image. It also has two keyword arguments, color and line_width, to specify properties of the shape. In this exercise, color can be 'red', 'green', or 'blue'. The Shape class also has a method draw(image) that will draw the shape into the specified image.

One Shape sub-class, Square, is already defined for you. Study its draw(image) method and then define two more classes, Line and Rectangle. Use these classes to draw two more shapes to the image. You will need to override both the __init__ and the draw method in your sub-classes.

1. The constructor for Line should take 4 values:

   ```
   Line(x1, y1, x2, y2)
   ```

   Here x1, y1 define one end point and x2, y2 define the other end point. color and line_width should be optional arguments.

2. The constructor for Rectangle should take 4 values:

   ```
   Rectangle(x, y, width, height)
   ```

   Again, color and line_width should be optional arguments.

Bonus

Add a Circle class.

Hints

The “image” has several methods to specify and also “stroke” a path.

move_to(x, y) move to an x, y position

line_to(x, y) add a line from the current position to x, y

arc(x, y, radius, start_angle, end_angle) add an arc centered at x, y with the specified radius from the start_angle to end_angle (specified in radians)

close_path() draw a line from the current point to the starting point of the path being defined

stroke_path() draw all the lines that have been added to the current path

See *Shape Inheritance - Solution*.

1.1.10 Particle Class

This is a quick exercise to practice working with a class. The Particle class has already been defined. In this exercise, your task is to make a few simple changes to the class.

1. Change the __repr__() method to print “mass:” and “velocity:” instead of “m:” and “v:”.
2. Add an energy() method, where the energy is given by the formula m*v**2/2.

See *Particle Class - Solution*.

1.1.11 Near Star Catalog

Read data into a list of classes and sort in various ways.

The file *stars.dat* contains data about some of the nearest stars. Data is arranged in the file in fixed-width fields:

```
0:17    Star name
18:28   Spectral class
29:34   Apparent magnitude
35:40   Absolute magnitude
41:46   Parallax in thosandths of an arc second
```

A typical line looks like:

```
Proxima Centauri  M5  e      11.05 15.49 771.8
```

This module also contains a Star class with attributes for each of those data items.

1. Write a function *parse_stars* which returns a list of Star instances, one for each star in the file *stars.dat*.
2. Sort the list of stars by apparent magnitude and print names and apparent magnitudes of the 10 brightest stars by apparent magnitude (lower numbers are brighter).
3. Sort the list of stars by absolute magnitude and print names, spectral class and absolute magnitudes the 10 faintest stars by absolute magnitude (lower numbers are brighter).
4. The distance of a star from the Earth in light years is given by:

   ```
   1/parallax in arc seconds * 3.26163626
   ```

   Sort the list by distance from the Earth and print names, spectral class and distance from Earth of the 10 closest stars.

Bonus

Spectral classes are codes of a form like 'M5', consisting of a letter and a number, plus additional data about luminosity and abnormalities in the spectrum. Informally, the initial part of the code gives information about the surface temperature and colour of the star. From hottest to coldest, the initial letter codes are 'OBAFGKM', with 'O' blue and 'M' red. The number indicates a relative position between the lettes, so 'A2' is hotter than 'A3' which is in turn hotter than 'F0'.

Sort the list of stars and print the names, spectral classes and distances of the 10 hottest stars in the list.

(Ignore stars with no spectral class or a spectral class that doesn't start with 'OBAFGKM'.)

Notes

Data from:

> Preliminary Version of the Third Catalogue of Nearby Stars GLIESE W., JAHREISS H. Astron. Rechen-Institut, Heidelberg (1991)

See *Near Star Catalog - Solution*.

1.1.12 Generators

Generators are created by function definitions which contain one or more yield expressions in their body. They are a much easier way to create iterators than by creating a class and implementing the __iter__ and next methods manually.

In this exercise you will write generator functions that create various iterators by using the yield expression inside a function body.

Create generators to

1. Iterate through a file by reading N bytes at a time; i.e.:

   ```
   N = 10000
   for data in chunk_reader(file, N):
       print len(data), type(data)
   ```

   should print:

   ```
   10000 <type 'str'>
   10000 <type 'str'>
   ...
   10000 <type 'str'>
   X <type 'str'>
   ```

   where X is the last number of bytes.

2. Iterate through a sequence N-items at a time; i.e.:

   ```
   for i, j, k in grouped([1,2,3,4,5,6,7], 3):
       print i, j, k
       print "="*10
   ```

   should print (note the last number is missing):

   ```
   1, 2, 3
   ==========
   4, 5, 6
   ==========
   ```

3. Implement "izip" based on a several input sequences; i.e.:

   ```
   list(izip(x,y,z))
   ```

   should produce the same as zip(x,y,z) so that in a for loop izip and zip work identically, but izip does not create an intermediate list.

See *Generators - Solution*.

1.2 Python Libraries Exercises

1.2.1 Column Stats (subprocess)

Background

The script in this directory calculates the number of rows in a file that satisfy a certain expression.

The name of the file to process can either be provided to the script from the command line or on stdin.

Problem

Write a program to call out to calculate.py operating on sp500hst_part.txt in this directory.

1. Using command line arguments
2. Using a pipe to communicate via stdin

Bonus

The script can actually take an expression to evaluate either as an additional command line argument or as an additional string passed to stdin.

Repeat 1) and 2) but change the expression to {6} > 50000

See *Column Stats (subprocess) - Solution 1* and *Column Stats (subprocess) - Solution 2*.

1.2.2 Dow Database

Topics: Database DB-API 2.0

The database table in the file 'dow2008.csv' has records holding the daily performance of the Dow Jones Industrial Average from the beginning of 2008. The table has the following columns (separated by a comma).

DATE OPEN HIGH LOW CLOSE VOLUME ADJ_CLOSE 2008-01-02 13261.82 13338.23 12969.42 13043.96 3452650000 13043.96 2008-01-03 13044.12 13197.43 12968.44 13056.72 3429500000 13056.72 2008-01-04 13046.56 13049.65 12740.51 12800.18 4166000000 12800.18 2008-01-07 12801.15 12984.95 12640.44 12827.49 4221260000 12827.49 2008-01-08 12820.9 12998.11 12511.03 12589.07 4705390000 12589.07 2008-01-09 12590.21 12814.97 12431.53 12735.31 5351030000 12735.31

1. Create a database table that has the same structure (use real for all the columns except the date column).
2. Insert all the records from dow.csv into the database.
3. Select (and print out) the records from the database that have a volume greater than 5.5 billion. How many are there?

Bonus

1. Select the records which have a spread between high and low that is greater than 4% and sort them in order of that spread.
2. Select the records which have an absolute difference between open and close that is greater than 1% (of the open) and sort them in order of that spread.

See *Dow Database - Solution*.

1.3 Software Craftsmanship Exercises

1.3.1 Test Driven Development Example

Write a function called *slope* that calculates the slope between two points. A point is specified by a two element sequence of (x,y) coordinates.

```
>>> pt1 = [0.0, 0.0]
>>> pt2 = [1.0, 2.0]
>>> slope(pt1, pt2)
2.0
```

Use Test Driven Development. Write your tests first in a separate file called tests_fancy_math.py.

Run your tests using the “nosetests” shell command. You can do this by changing to the “slope” directory where your fancy_math.py is defined and running “nosetests”. From IPython, you can run it like this:

```
In [1]: cd <someplace>/exercises/slope
In [2]: !nosestests
...
--------------------------------------------------
Ran 3 tests in 0.157s
```

If you would like to see more verbose output, use the “-v” option:

```
In [3]: !nosetests -v
test_fancy_math.test_slope_xxx ... ok
test_fancy_math.test_slope_yyy ... ok
...
```

By default, nose captures all output and does not print it to the screen. If you would like to see the output of print statements, use the “-s” flag.

See: *Test Driven Development Example - Solution 1*.

1.3.2 Code Check

This code has an assortment of bugs, and its style doesn’t conform to PEP-8. Use pyflakes and pep8 to find and fix the code.

You may have to install pep8 with the command:

$ easy_install pep8

It might take a few iterations before pyflakes doesn’t complain about something.

See: *Code Check - Solution*.

1.4 Numpy Exercises

1.4.1 Plotting

In PyLab, create a plot display that looks like the following:

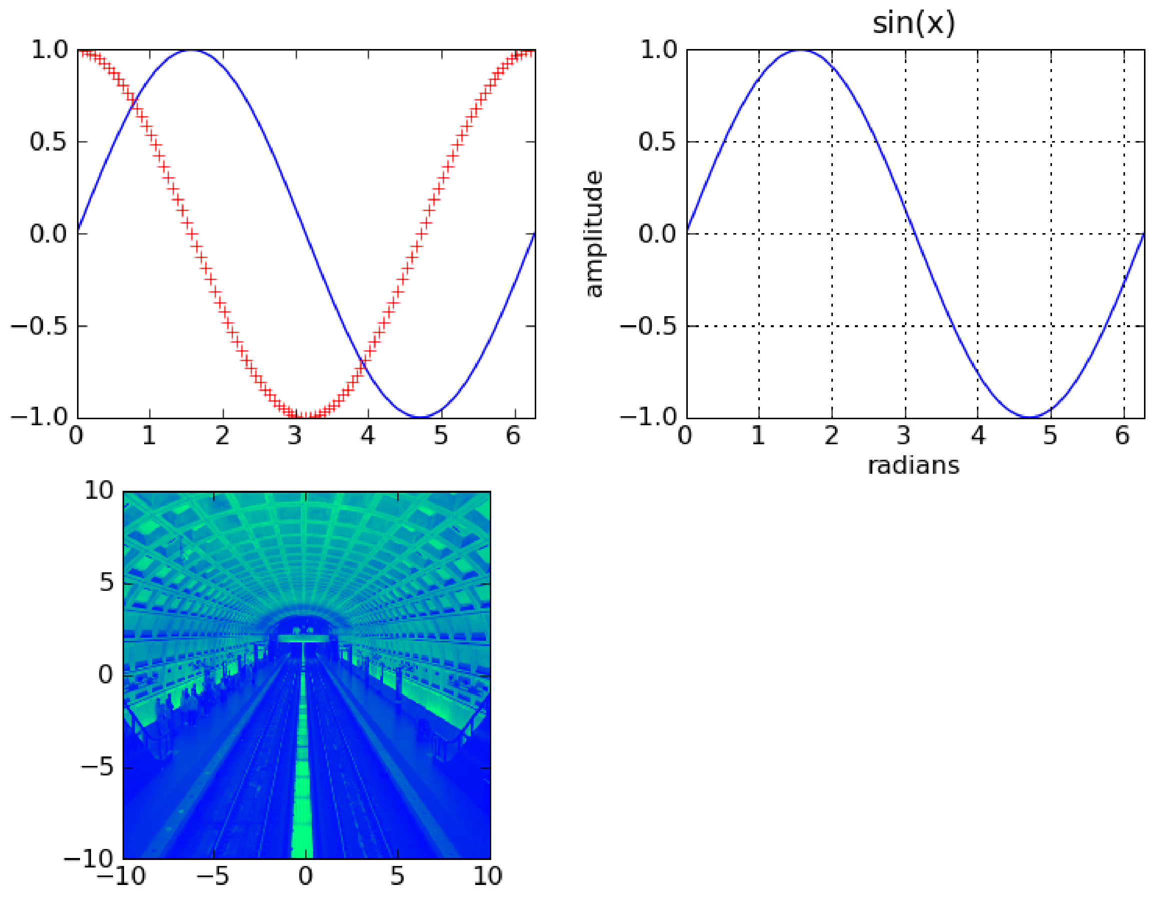

Photo credit: David Fettig

This is a 2x2 layout, with 3 slots occupied.

1. Sine function, with blue solid line; cosine with red '+' markers; the extents fit the plot exactly. Hint: see the axis() function for setting the extents.
2. Sine function, with gridlines, axis labels, and title; the extents fit the plot exactly.
3. Image with color map; the extents run from -10 to 10, rather than the default.

Save the resulting plot image to a file. (Use a different file name, so you don't overwrite the sample.)

The color map in the example is 'winter'; use 'cm.<tab>' to list the available ones, and experiment to find one you like.

Start with the following statements:

```
from scipy.misc.pilutil import imread

x = linspace(0, 2*pi, 101)
s = sin(x)
```

```
c = cos(x)

# 'flatten' creates a 2D array from a JPEG.
img = imread('dc_metro.jpg', flatten=True)
```

See *Plotting - Solution*.

1.4.2 Calculate Derivative

Topics: NumPy array indexing and array math.

Use array slicing and math operations to calculate the numerical derivative of `sin` from 0 to `2*pi`. There is no need to use a 'for' loop for this.

Plot the resulting values and compare to `cos`.

Bonus

Implement integration of the same function using Riemann sums or the trapezoidal rule.

See *Calculate Derivative - Solution*.

1.4.3 Load Array from Text File

0. From the IPython prompt, type:

   ```
   In [1]: loadtxt?
   ```

   to see the options on how to use the loadtxt command.

1. Use loadtxt to load in a 2D array of floating point values from 'float_data.txt'. The data in the file looks like:

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

   The resulting data should be a 2x4 array of floating point values.

2. In the second example, the file 'float_data_with_header.txt' has strings as column names in the first row:

   ```
   c1 c2 c3 c4
    1  2  3  4
    5  6  7  8
   ```

   Ignore these column names, and read the remainder of the data into a 2D array.

   Later on, we'll learn how to create a "structured array" using these column names to create fields within an array.

BONUS:

3. A third example is more involved. It contains comments in multiple locations, uses multiple formats, and includes a useless column to skip:

   ```
   -- THIS IS THE BEGINNING OF THE FILE --
   % This is a more complex file to read!

   % Day,  Month,  Year, Useless Col, Avg Power
      01,     01,  2000,      ad766,         30
      02,     01,  2000,       t873,         41
   ```

```
% we don't have Jan 03rd!
   04,     01,  2000,       r441,         55
   05,     01,  2000,       s345,         78
   06,     01,  2000,       x273,        134 % that day was crazy
   07,     01,  2000,       x355,         42

%-- THIS IS THE END OF THE FILE --
```

See *Load Array from Text File - Solution*

1.4.4 Filter Image

Read in the "dc_metro" image and use an averaging filter to "smooth" the image. Use a "5 point stencil" where you average the current pixel with its neighboring pixels:

```
0 0 0 0 0 0 0
0 0 0 x 0 0 0
0 0 x x x 0 0
0 0 0 x 0 0 0
0 0 0 0 0 0 0
```

Plot the image, the smoothed image, and the difference between the two.

Bonus

Re-filter the image by passing the result image through the filter again. Do this 50 times and plot the resulting image.

See *Filter Image - Solution*.

1.4.5 Dow Selection

Topics: Boolean array operators, sum function, where function, plotting.

The array 'dow' is a 2-D array with each row holding the daily performance of the Dow Jones Industrial Average from the beginning of 2008 (dates have been removed for exercise simplicity). The array has the following structure:

```
OPEN      HIGH      LOW       CLOSE     VOLUME      ADJ_CLOSE
13261.82  13338.23  12969.42  13043.96  3452650000  13043.96
13044.12  13197.43  12968.44  13056.72  3429500000  13056.72
13046.56  13049.65  12740.51  12800.18  4166000000  12800.18
12801.15  12984.95  12640.44  12827.49  4221260000  12827.49
12820.9   12998.11  12511.03  12589.07  4705390000  12589.07
12590.21  12814.97  12431.53  12735.31  5351030000  12735.31
```

0. The data has been loaded from a .csv file for you.
1. Create a "mask" array that indicates which rows have a volume greater than 5.5 billion.
2. How many are there? (hint: use sum).
3. Find the index of every row (or day) where the volume is greater than 5.5 billion. hint: look at the where() command.

Bonus

1. Plot the adjusted close for *every* day in 2008.
2. Now over-plot this plot with a 'red dot' marker for every day where the dow was greater than 5.5 billion.

See *Dow Selection - Solution*.

1.4.6 Wind Statistics

Topics: Using array methods over different axes, fancy indexing.

1. The data in 'wind.data' has the following format:

   ```
   61  1  1 15.04 14.96 13.17  9.29 13.96  9.87 13.67 10.25 10.83 12.58 18.50 15.04
   61  1  2 14.71 16.88 10.83  6.50 12.62  7.67 11.50 10.04  9.79  9.67 17.54 13.83
   61  1  3 18.50 16.88 12.33 10.13 11.17  6.17 11.25  8.04  8.50  7.67 12.75 12.71
   ```

   The first three columns are year, month and day. The remaining 12 columns are average windspeeds in knots at 12 locations in Ireland on that day.

   Use the 'loadtxt' function from numpy to read the data into an array.
2. Calculate the min, max and mean windspeeds and standard deviation of the windspeeds over all the locations and all the times (a single set of numbers for the entire dataset).
3. Calculate the min, max and mean windspeeds and standard deviations of the windspeeds at each location over all the days (a different set of numbers for each location)
4. Calculate the min, max and mean windspeed and standard deviations of the windspeeds across all the locations at each day (a different set of numbers for each day)
5. Find the location which has the greatest windspeed on each day (an integer column number for each day).
6. Find the year, month and day on which the greatest windspeed was recorded.
7. Find the average windspeed in January for each location.

You should be able to perform all of these operations without using a for loop or other looping construct.

Bonus

b1. Calculate the mean windspeed for each month in the dataset. Treat January 1961 and January 1962 as *different* months. (hint: first find a way to create an identifier unique for each month. The second step might require a for loop.)

b2. Calculate the min, max and mean windspeeds and standard deviations of the windspeeds across all locations for each week (assume that the first week starts on January 1 1961) for the first 52 weeks. This can be done without any for loop.

Bonus Bonus

Calculate the mean windspeed for each month without using a for loop. (Hint: look at *searchsorted* and *add.reduceat*.)

Notes

These data were analyzed in detail in the following article:

> Haslett, J. and Raftery, A. E. (1989). Space-time Modelling with Long-memory Dependence: Assessing Ireland's Wind Power Resource (with Discussion). Applied Statistics 38, 1-50.

See *Wind Statistics - Solution*.

1.4.7 Sinc Function

Topics: Broadcasting, Fancy Indexing

Calculate the sinc function: sin(r)/r. Use a Cartesian x,y grid and calculate `r = sqrt(x**2+y**2)` with 0 in the center of the grid. Calculate the function for -15,15 for both x and y.

See *Sinc Function - Solution*.

1.4.8 Structured Array

Topics: Read columns of data into a structured array using loadtxt.

1. The data in 'short_logs.crv' has the following format:

   ```
   DEPTH           CALI        S-SONIC   ...
   8744.5000   -999.2500   -999.2500   ...
   8745.0000   -999.2500   -999.2500   ...
   8745.5000   -999.2500   -999.2500   ...
   ```

   Here the first row defines a set of names for the columns of data in the file. Use these column names to define a dtype for a structured array that will have fields 'DEPTH', 'CALI', etc. Assume all the data is of the float64 data format.

2. Use the 'loadtxt' method from numpy to read the data from the file into a structured array with the dtype created in (1). Name this array 'logs'

3. The 'logs' array is nice for retrieving columns from the data. For example, logs['DEPTH'] returns the values from the DEPTH column of the data. For row-based or array-wide operations, it is more convenient to have a 2D view into the data, as if it is a simple 2D array of float64 values.

   Create a 2D array called 'logs_2d' using the view operation. Be sure the 2D array has the same number of columns as in the data file.

4. -999.25 is a "special" value in this data set. It is intended to represent missing data. Replace all of these values with NaNs. Is this easier with the 'logs' array or the 'logs_2d' array?

5. Create a mask for all the "complete" rows in the array. A complete row is one that doesn't have any NaN values measured in that row.

   HINT: The `all` function is also useful here.

6. Plot the VP vs VS logs for the "complete" rows.

See *Structured Array - Solution*.

1.5 Scipy Exercises

1.5.1 Statistics Functions

1. Import stats from scipy, and look at its docstring to see what is available in the stats module:

   ```
   In [1]: from scipy import stats
   In [2]: stats?
   ```

2. Look at the docstring for the normal distribution:

   ```
   In [3]: stats.norm?
   ```

   You'll notice it has these parameters:

   loc: This variable shifts the distribution left or right. For a normal distribution, this is the mean. It defaults to 0.

   scale: For a normal distribution, this is the standard deviation.

3. Create a single plot of the pdf of a normal distribution with mean=0 and standard deviation ranging from 1 to 3. Plot this on the range from -10 to 10.

4. Create a "frozen" normal distribution object with mean=20.0, and standard deviation=3:

   ```
   my_distribution = stats.norm(20.0, 3.0)
   ```

5. Plot this distribution's pdf and cdf side by side in two separate plots.

6. Get 5000 random variable samples from the distribution, and use the stats.norm.fit method to estimate its parameters. Plot the histogram of the random variables as well as the pdf for the estimated distribution. (Try using stats.histogram. There is also pylab.hist which makes life easier.)

See: *Statistics Functions - Solution*.

1.5.2 Monte Carlo Options

Background

One approach to pricing options is to simulate the instrument price over the lifetime of the option using Monte Carlo simulation (assuming some model). The resulting random walk of the stock price can be used to determine the option pay-off.

The option price is then usually calculated as the average value of the simulated pay-offs discounted at the risk-free rate of return (exp(-r*T)).

An often-used model for the stock price at time *T* is that it is log-normally distributed where log(ST) is normally distributed with:

```
mean - log(S0) + (mu-sigma**2 / 2) * T
```

and:

```
variance - sigma**2 * T
```

This implies that the *ST* is log-normally distributed with shape parameter `sigma * sqrt(T)` and scale parameter `S0*exp((mu-sigma**2/2)*T)`.

For option pricing, *mu* is the risk-free rate of return and *sigma* is the volatility.

The value of a call option at maturity is ST-K if ST>K and 0 if ST<K.

The value of a put option at maturity is K-ST if K>ST and 0 if ST>K.

Problem

1. Create a function that uses a Monte Carlo method to price vanilla call and put options, by drawing samples from a log-normal distribution.
2. Create a function that uses a Monte-Carlo method to price vanilla call and put options, by drawing samples from a normal distribution.
3. Bonus, compare 1 and 2 against the Black-Scholes method of pricing.

See *Monte Carlo Options - Solution*.

1.5.3 Gaussian Kernel Density Estimation

Estimate a probability density function for a set of random samples using a Gaussian Kernel Density estimator.

Create 100 normally distributed samples with mean=1.0 and std=0.0 using the stats.norm.rvs method.

Use stats.kde.gaussian_kde to estimate the pdf for the distribution of these samples. Compare that to the analytic pdf for the distribution as well as the histogram of the actual samples over the interval -3, 10.

Bonus:

Construct a set of samples (200 or so) from two different normal distributions, mean=0.0, std=0.5 and mean=5.0, std=1.0. As with the first example, compare this against the analytic pdf as well as the histogram of the actual samples over the interval -3, 10.

See: *Gaussian Kernel Density Estimation - Solution*.

1.5.4 Estimate Volatility

Background

A standard model of price fluctuation is:

```
dS/S = mu dt + sigma * epsilon * sqrt(dt)
```

where:

- *S* is the stock price.
- *dS* is the change in stock price.
- *mu* is the rate of return.
- *dt* is the time interval.
- *epsilon* is a normal random variable with mean 0 and variance 1 that is uncorrelated with other time intervals.
- *sigma* is the volatility.

It is desired to make estimates of *sigma* from historical price information. There are simple approaches to do this that assume volatility is constant over a period of time. It is more accurate, however, to recognize that *sigma* changes with each day and therefore should be estimated at each day. To effectively do this from historical price data alone, some kind of model is needed.

The GARCH(1,1) model for volatility at time *n*, estimated from data available at the end of time *n-1* is:

```
sigma_n**2 = gamma V_L + alpha u_{n-1}**2 + beta sigma_{n-1}**2
```

where:

- *V_L* is long-running volatility
- `alpha+beta+gamma = 1`
- `u_n = log(S_n / S_{n-1})` or `(S_n - S_{n-1})/S_{n-1}`

Estimating *V_L* can be done as the mean of `u_n**2` (variance of *u_n*). Estimating parameters *alpha* and *beta* is done by finding the parameters that maximize the likelihood that the data *u_n* would be observed. If it is assumed that the *u_n* are normally distributed with mean 0 and variance *sigma_n*, this is equivalent to finding *alpha* and *beta* that minimize:

```
L(alpha, beta) = sum_{n}(log(sigma_n**2) + u_n**2 / sigma_n**2)
```

where `sigma_n**2` is computed as above.

Problem

1. Create a function to read in daily data from `sp500hst.txt` for the S&P 500 for a particular stock. The file format is:

   ```
   date, symbol, open, high, low, close, volume
   ```

2. Create a function to estimate volatility per annum for a specific number of periods (assume 252 trading days in a year).
3. Create a function to compute `sigma**2_n` for each *n* from *alpha* and *beta* and ``u_n**2` (you may need to use a for loop for this). Use *V_L* to start the recursion.
4. Create a function that will estimate volatility using GARCH(1,1) approach by minimizing `L(alpha, beta)`.
5. Use the functions to construct a plot of volatility per annum for a stock of your choice (use 'AAPL' if you don't have a preference) using quarterly, monthly, and GARCH(1,1) estimates.

You may find the repeat method useful to extend quarterly and monthly estimates to be the same size as GARCH(1,1) estimates.

See *Estimate Volatility - Solution*.

1.5.5 Using fsolve

1. Set up the system of equations:

   ```
   f_0 = x**2 + y**2 - 2.0
   f_1 = x**2 - y**2 - 1.0
   ```

2. Find x_0 and x_1 so that f_0==0 and f_1==0. Hint: See scipy.optimize.fsolve
3. Create images of f_0 and f_1 around 0 and plot the guess point and the solution point.

   http://www.scipy.org/doc/api_docs/scipy.optimize.minpack.html

See *Using fsolve - Solution*.

1.5.6 Black-Scholes Pricing

Background

The Black-Scholes option pricing models for European-style options on a non-dividend paying stock are:

```
c = S0 * N(d1) - K * exp(-r*T)* N(d2)   for a call option and

p = K*exp(-r*T)*N(-d2) - S0 * N(-d1)    for a put option
```

where:

```
d1 = (log(S0/K) + (r + sigma**2 / 2)*T) / (sigma * sqrt(T))
d2 = d1 - sigma * sqrt(T)
```

Also:

- `log()` is the natural logarithm.
- `N(x)` is the cumulative density function of a standardized normal distribution.
- *S0* is the current price of the stock.
- *K* is the strike price of the option.
- *T* is the time to maturity of the option.
- *r* is the (continuously-compounded) risk-free rate of return.
- *sigma* is the stock price volatility.

Problem

Create a function that returns the call and put options prices for using the Black-Scholes formula and the inputs of *S0*, *K*, *T*, *r*, and *sigma*.

Hint: You will need scipy.special.ndtr or scipy.stats.norm.cdf. Notice that N(x) + N(-x) = 1.

See: *Black-Scholes Pricing - Solution*.

1.5.7 Black-Scholes Implied Volatility

Background

The Black-Scholes option pricing models for European-style options on a non-dividend paying stock are:

```
c = S0 * N(d1) - K * exp(-r*T)* N(d2)   for a call option and

p = K*exp(-r*T)*N(-d2) - S0 * N(-d1)    for a put option
```

where:

```
d1 = (log(S0/K) + (r + sigma**2 / 2)*T) / (sigma * sqrt(T))
d2 = d1 - sigma * sqrt(T)
```

Also:

- `log()` is the natural logarithm.
- `N(x)` is the cumulative density function of a standardized normal distribution.

- *S0* is the current price of the stock.
- *K* is the strike price of the option.
- *T* is the time to maturity of the option.
- *r* is the (continuously-compounded) risk-free rate of return.
- *sigma* is the stock price volatility.

Problem

1. The one parameter in the Black-Scholes formula that is not readily available is *sigma*. Suppose you observe that the price of a call option is *cT*. The value of *sigma* that would produce the observed value of *cT* is called the "implied volatility". Construct a function that calculates the implied volatility from *S0*, *K*, *T*, *r*, and *cT*.

   Hint: Use a root-finding technique such as scipy.optimize.fsolve. (or scipy.optimize.brentq since this is a one-variable root-finding problem).
2. Repeat #1, but use the observed put option price to calculate implied volatility.
3. Bonus: Make the implied volatility functions work for vector inputs (at least on the call and put prices)

See: *Black-Scholes Implied Volatility - Solution*

1.5.8 Data Fitting

1. Define a function with four arguments, x, a, b, and c, that computes:

   ```
   y = a*exp(-b*x) + c
   ```

   (This is done for you in the code below.)

   Then create an array x of 75 evenly spaced values in the interval 0 <= x <= 5, and an array y = f(x,a,b,c) with a=2.0, b = 0.76, c=0.1.
2. Now use scipy.stats.norm to create a noisy signal by adding Gaussian noise with mean 0.0 and standard deviation 0.2 to y.
3. Calculate 1st, 2nd and 3rd degree polynomial functions to fit to the data. Use the polyfit and poly1d functions from scipy.
4. Do a least squares fit to the orignal exponential function using scipy.curve_fit.

See: *Data Fitting - Solution*.

1.6 Interface With Other Languages Exercises

1.6.1 Fancy science (hand wrap)

This exercise provides practice at writing a C extension module "by hand." That is, you will write all the interface code yourself instead of using a tool such as SWIG.

fancy_science.h and fancy_science.c are the header and implementation file for a *very* simple library that contains three functions:

```
/* Calculate the hypotenuse of a triangle using the Pythagorean theorem */
double my_hypot(double a, double b);

/* Calculate Euler's number, e, using n series elements. */
double eulers_number(int n);

/* Einstein's e=mc^2 calculation */
float energy(float mass)
```

In this exercise, you will wrap these functions and expose them to Python through an extension module. The skeleton for this module, `func_module.c` already exists, and the first function, `my_hypot()`, has already been wrapped for you.

There is a `setup_handwrap.py` file that is the Python equivalent of a "make" file. To build the extension on Windows using the mingw compiler and so that the resulting module is put in this directory instead of a separate build directory, do the following:

```
c:\path\exercise\> setup_handwrap.py build_ext --compiler=mingw32 --inplace
```

The `build.bat` file has this command in it already for convenience, so you can also build the module by typing:

```
c:\path\exercise\> build.bat
```

To test the module, import it from the Python command interpreter and call its functions. The following will work "out of the box.":

```
c:\path\exercise\> python
>>> import fancy_science
>>> fancy_science.my_hypot(3.0, 4.0)
5.0
```

Help

The demo directory has a more detailed example of the `my_hypot()` function that may be useful. See the `readme.txt` in that directory to see how to build the examples.

The python documentation on building extensions is here:

> http://docs.python.org/ext/ext.html

The format strings for parsing tuples is here:

> http://www.python.org/doc/1.5.2p2/ext/parseTuple.html

For more information about writing setup.py files, look here:

> http://docs.python.org/inst/inst.html

See *Fancy science (hand wrap) - Solution*

1.6.2 Blitz Inline Comparison

This example takes the numpy expression used in the filter_image exercise and compares the speed of numpy to that of weave.blitz.

Read in the lena image and use an averging filter to "smooth" the image. Use a "5 point stencil" where you average the current pixel with its neighboring pixels:

```
0 0 0 0 0 0 0
0 0 0 x 0 0 0
0 0 x x x 0 0
0 0 0 x 0 0 0
0 0 0 0 0 0 0
```

Once you have a numpy expression that works correctly, time it using time.time (or time.clock on Windows).

Use scipy.weave.blitz to run the same expression. Again time it.

Compare the speeds of the two function and calculate the speed-up (numpy_time/weave_time).

Plot two images that result from the two approaches and compare them.

See *Blitz Inline Comparison - Solution*.

1.6.3 Weave Comparison

Compare the speed of NumPy, weave.blitz and weave.inline for the following equation:

```
result=a+b*(c-d)
```

Set up all arrays so that they are one-dimensional and have 1 million elements in them.

See: *Weave Comparison - Solution*.

1.6.4 Mandelbrot with Weave

The code in this script generates a plot of the Mandelbrot set.

1. Use weave to speed up this code
2. Compare the speed of the weave code with the NumPy-only version.

Hints: * The mandelbrot_escape function should be placed in support_code and weave.inline should be used to inline a 2-d loop over the data passed in * The output data should be pre-allocated and passed in as another array. * Recall that weave creates C++ extensions so that std:

```
complex<double>
```

can be used.

See *Mandelbrot with Weave - Solution*.

1.6.5 Mandelbrot Cython

This exercise provides practice at writing a C extension module using Cython. The object of this is to take an existing Python module and speed it up by re-writing it in Cython.

The code in this script (mandelbrot.py) generates a plot of the Mandelbrot set.

1. The pure Python version of the code is contained in this file. Run this and see how long it takes to run on your system.
2. The file mandelbrot.pyx contains the two functions mandelbrot_escape and generate_mandelbrot which are identical to the Python versions in this file. Use the setup.py file to build a Cython module with the command:

   ```
   python setup.py build_ext --inplace
   ```

and use the script mandelbrot_test.py to run the resulting code. How much of a speed-up (if any) do you get from compiling the unmodified Python code in Cython.

3. Add variable typing for the scalar variables in the mandelbrot.pyx file. Re-compile, and see how much of a speed-up you get.
4. Turn the mandelbrot_escape function into a C only function. Re-compile, and see how much of a speed-up you get.

Bonus

Use the numpy Cython interface to optimize the code even further. Re-compile, and see how much of a speed-up you get.

See *Mandelbrot Cython - Solution 1* and *Mandelbrot Cython - Solution 2* and *Mandelbrot Cython - Solution 3*.

1.7 Ipython Exercises

1.7.1 Mandelbrot Parallelized

The code in this script generates a plot of the Mandelbrot set.

Use ipython's multi-client engine to parallelize this, scattering over the y-axis to break computation into chunks and then gathering the results before plotting.

Bonus

How fast is the parallel version compared to the regular version on your system?

See *Mandelbrot Parallelized - Solution*.

CHAPTER TWO

SOLUTIONS

2.1 Python Language - Solutions

2.1.1 Vote String - Solution

vote_string_solution.py:

```
"""

You have the string below, which is a set of "yes/no" votes,
where "y" or "Y" means yes and "n" or "N" means no. Determine
the percentages of yes and no votes.

::

    votes = "y y n N Y Y n n N y Y n Y"
"""

votes = "y y n N Y Y n n N y Y n Y"

# Force everything to lower case:
votes = votes.lower()

# Now count the yes votes:
yes = votes.count("y")
no = votes.count("n")

total = yes + no

# Notice cast to float! Otherwise you have an integer division.

print "vote outcome:"
print "% yes:", yes/float(total) * 100
print "% no:", no/float(total) * 100

# Alternative approach using future behavior of integer
# division yielding floats.

#from __future__ import division

#print "% yes:", yes/total * 100
#print "% no:", no/total * 100
```

2.1.2 Sort Words - Solution

sort_words_solution.py:

```
"""
Given a (partial) sentence from a speech, print out
a list of the words in the sentence in alphabetical order.
Also print out just the first two words and the last
two words in the sorted list.

::

    speech = '''Four score and seven years ago our fathers brought forth
             on this continent a new nation, conceived in Liberty, and
             dedicated to the proposition that all men are created equal.
             '''


Ignore case and punctuation.
"""

speech = '''Four score and seven years ago our fathers brought forth
      on this continent a new nation, conceived in Liberty, and
      dedicated to the proposition that all men are created equal.
      '''

# Convert to lower case so that case is ignored in sorting:
speech = speech.lower()

# Replace punctuation with spaces
# (you could just remove them as well):
speech = speech.replace(",", " ")
speech = speech.replace(".", " ")

# Split the words into a list:
words = speech.split()

# Sort the words "in place":
words.sort()

print "All words, in alphabetical order:"
print words
print
print "The first 2 words: "
print words[:2]
print
print "The last 2 words: "
print words[-2:]
```

2.1.3 Roman Dictionary - Solution

roman_dictionary_solution.py:

```
"""
Mark Antony keeps a list of the people he knows in several dictionaries
based on their relationship to him::
    friends = {'julius': '100 via apian', 'cleopatra': '000 pyramid parkway'}
    romans = dict(brutus='234 via tratorium', cassius='111 aqueduct lane')
    countrymen = dict([('plebius','786 via bunius'),
                       ('plebia', '786 via bunius')])


1. Print out the names for all of Antony's friends.
2. Now all of their addresses.
3. Now print them as "pairs".
4. Hmmm.  Something unfortunate befell Julius.  Remove him from the
   friends list.
5. Antony needs to mail everyone for his second-triumvirate party.  Make
   a single dictionary containing everyone.
6. Antony's stopping over in Egypt and wants to swing by Cleopatra's
   place while he is there. Get her address.
7. The barbarian hordes have invaded and destroyed all of Rome.
   Clear out everyone from the dictionary.
"""

friends = {'julius': '100 via apian', 'cleopatra': '000 pyramid parkway'}
romans = dict(brutus='234 via tratorium', cassius='111 aqueduct lane')
countrymen = dict([('plebius','786 via bunius'), ('plebia', '786 via bunius')])

# Print out the names for all of Antony's friends:
print 'friend names:', friends.keys()
print

# Now all of their addresses:
print 'friend addresses:', friends.values()
print

# Now print them as "pairs":
print 'friend (name, address) pairs:', friends.items()
print

# Hmmm.  Something unfortunate befell Julius.  Remove him from the friends
# list:
del friends['julius']

# Antony needs to mail everyone for his second-triaumvirate party.  Make a
# single dictionary containing everyone:
mailing_list = {}
mailing_list.update(friends)
mailing_list.update(romans)
mailing_list.update(countrymen)

print 'party mailing list:'
print mailing_list
print

# Or, using a loop (which we haven't learned about yet...):
print 'party mailing list:'
for name, address in mailing_list.items():
    print name, ':\t', address
print

# Antony's stopping over in Egypt and wants to swing by Cleopatra's place
# while he is there. Get her address:
```

```
print "Cleopatra's address:", friends['cleopatra']

# The barbarian hordes have invaded and destroyed all of Rome. Clear out
# everyone from the dictionary:
mailing_list.clear()
```

2.1.4 Filter Words - Solution

filter_words_solution.py:

```
"""
Print out only words that start with "o", ignoring case::

    lyrics = '''My Bonnie lies over the ocean.
                My Bonnie lies over the sea.
                My Bonnie lies over the ocean.
                Oh bring back my Bonnie to me.
                '''

Bonus points: print out words only once.
"""

lyrics = """My Bonnie lies over the ocean.
            My Bonnie lies over the sea.
            My Bonnie lies over the ocean.
            Oh bring back my Bonnie to me.
         """

# Ignore case:
lyrics = lyrics.lower()

# Get rid of periods:
lyrics = lyrics.replace('.'," ")

# Split into a list of words:
words = lyrics.split()

# Make a list to fill with 'o' words:
o_words = []

for word in words:
    if word[0] == "o":
        o_words.append(word)

print "words that start with 'o':"
print o_words

# Converting the list to a set removes all non-unique entries:
print "unique words that start with 'o':"
print set(o_words)
```

2.1.5 Inventory - Solution

inventory_solution.py:

```python
"""
Calculate and report the current inventory in a warehouse.

Assume the warehouse is initially empty.

The string, warehouse_log, is a stream of deliveries to
and shipments from a warehouse.  Each line represents
a single transaction for a part with the number of
parts delivered or shipped.  It has the form::

    part_id count

If "count" is positive, then it is a delivery to the
warehouse. If it is negative, it is a shipment from
the warehouse.
"""

warehouse_log = """ frombicator      10
                    whitzlegidget    5
                    splatzleblock    12
                    frombicator      -3
                    frombicator      20
                    foozalator       40
                    whitzlegidget    -4
                    splatzleblock    -8
                """

# Remove any leading and trailing whitespace from the string::

warehouse_log = warehouse_log.strip()

# Create a list of transactions from the log with part as string.
# and count as integer::

transactions = []
for line in warehouse_log.split("\n"):
    part, count = line.split()
    transaction = (part, int(count))
    transactions.append(transaction)

# Process the transactions, keeping track of inventory::

# Initialize the inventory dictionary so that all the
# parts have a count of 0.
inventory = {}
for part, count in transactions:
    inventory[part] = 0

for part, count in transactions:
    # read the part and count out of the transaction line.
    inventory[part] += count

# And print it out::

for part in inventory:
    print "%-20s %d" % (part, inventory[part])
```

2.1.6 Financial Module - Solution

financial_module_solution.py:

```
"""
Background
-----------

The future value (fv) of money is related to the present value (pv)
of money and any recurring payments (pmt) through the equation::

    fv + pv*(1+r)**n + pmt*(1+r*when)/r * ((1+r)**n - 1) = 0

or, when r == 0::

    fv + pv + pmt*n == 0

Both of these equations assume the convention that cash in-flows are
positive and cash out-flows are negative.  The additional variables in
these equations are:

* n: number of periods for recurring payments
* r: interest rate per period (as a decimal fraction)
* when: When payments are made:

  - (1) for beginning of the period
  - (0) for the end of the period

The interest calculations are made through the end of the
final period regardless of when payments are due.

Problem
--------

Take the script financial_calcs.py and use it to construct a module with
separate functions that can perform the needed calculations with
arbitrary inputs to solve general problems based on the time value
of money equation given above.

Use keyword arguments in your functions to provide common default
inputs where appropriate.

Bonus
-----

1) Document your functions.
2) Add a function that calculates the number of periods from the other variables.
3) Add a function that calculates the rate from the other variables.

"""


def future_value(r, n, pmt, pv=0.0, when=1):
    """Future value in "time value of money" equations.

    * r: interest rate per payment (decimal fraction)
    * n: number of payments
    * pv: present value
    * when: when payment is made, either 1 (begining of period, default) or
```

```
      0 (end of period)

    """
    return -pv*(1+r)**n - pmt*(1+r*when)/r * ((1+r)**n - 1)



def present_value(r, n, pmt, fv=0.0, when=0):
    """Present value in "time value of money" equations.

    The present value of an annuity (or a one-time future value
    to be realized later)

    * r: interest rate per period (decimal fraction)
    * n: number of periods
    * pmt: the fixed payment per period
    * fv: the amount that should be available after the final period.
    * when: when payment is made, either 1 (begining of period) or
      0 (end of period, default)
    """
    return -(fv + pmt*(1+r*when)/r * ((1+r)**n - 1)) / (1+r)**n



def payment(r, n, pv, fv=0.0, when=0):
    """Payment in "time value of money" equations.

    Calculate the payment required to convert the present value into the future
    value.

    * r: interest rate per period (decimal fraction)
    * n: number of periods
    * pv: present value
    * fv: the amount that should be available after the final period.
    * when: when payment is made, either 1 (begining of period) or
      0 (end of period, default)

    """
    return -(fv + pv*(1+r)**n) * r / (1+r*when) / ((1+r)**n - 1)


# Future Value Example.
yearly_rate = .0325
monthly_rate = yearly_rate / 12
monthly_periods = 5 * 12 # 5 years
monthly_payment = -100 # $100 per period.
print "future value is ", future_value(monthly_rate, monthly_periods,
                                                                monthly_payment)


# Present Value Example.
yearly_rate = .06
monthly_rate = yearly_rate / 12
monthly_periods = 10 * 12 # 10 years
monthly_income = 500
fv = 1000
pv = present_value(monthly_rate, monthly_periods, monthly_income,
                                  fv=fv)
```

```
print "present value is ", pv

# Loan Payment
yearly_rate = .065
monthly_rate = yearly_rate / 12
monthly_periods = 15 * 12 # 15 years
loan_amount = 300000
print "payment is ", payment(monthly_rate, monthly_periods, loan_amount)
```

2.1.7 ASCII Log File - Solution 1

ascii_log_file_solution.py:

```
"""
Read in a set of logs from an ASCII file.

Read in the logs found in the file `short_logs.crv`.
The logs are arranged as follows::

    DEPTH    S-SONIC    P-SONIC ...
    8922.0   171.7472   86.5657
    8922.5   171.7398   86.5638
    8923.0   171.7325   86.5619
    8923.5   171.7287   86.5600
    ...

So the first line is a list of log names for each column of numbers.
The columns are the log values for the given log.

Make a dictionary with keys as the log names and values as the
log data::

    >>> logs['DEPTH']
    [8922.0, 8922.5, 8923.0, ...]
    >>> logs['S-SONIC']
    [171.7472, 171.7398, 171.7325, ...]

Bonus
-----

Time your example using::

    run -t 'ascii_log_file.py'

And see if you can come up with a faster solution. You may want to try the
`long_logs.crv` file in this same directory for timing, as it is much larger
than the `short_logs.crv` file. As a hint, reading the data portion of the array
in at one time combined with strided slicing of lists is useful here.

Bonus Bonus
-----------

Make your example a function so that it can be used in later parts of the class
to read in log files::

        def read_crv(file_name):
            ...
```

```
Copy it to the class_lib directory so that it is callable by all your other
scripts.
"""

log_file = open('long_logs.crv')

# The first line is a header that has all the log names:
header = log_file.readline()
log_names = header.split()
log_count = len(log_names)

# Read in each row of values, converting them to floats as
# they are read in.  Assign them to the log name for their
# particular column:
logs = {}

# Initialize the logs dictionary so that it contains the log names
# as keys, and an empty list for the values.
for name in log_names:
    logs[name] = []

for line in log_file:
    values = [float(val) for val in line.split()]
    for i, name in enumerate(log_names):
        logs[name].append(values[i])

log_file.close()

# output the first 10 values for the DEPTH log.

print 'DEPTH:', logs['DEPTH'][:10]
```

2.1.8 ASCII Log File - Solution 2

ascii_log_file_solution2.py:

```
"""
This version is about 35% faster than the original on largish files
because it reads in all the data at once and then uses array slicing
to assign the data elements to the correct column (or log).

Read in a set of logs from an ASCII file.

Read in the logs found in the file `short_logs.crv`.
The logs are arranged as follows::

    DEPTH    S-SONIC    P-SONIC ...
    8922.0   171.7472   86.5657
    8922.5   171.7398   86.5638
    8923.0   171.7325   86.5619
    8923.5   171.7287   86.5600
    ...

So the first line is a list of log names for each column of numbers.
The columns are the log values for the given log.
```

```
Make a dictionary with keys as the log names and values as the
log data::

    >>> logs['DEPTH']
    [8922.0, 8922.5, 8923.0, ...]
    >>> logs['S-SONIC']
    [171.7472, 171.7398, 171.7325, ...]

Bonus
-----

Time your example using::

    run -t 'ascii_log_file.py'

And see if you can come up with a faster solution. You may want to try the
`long_logs.crv` file in this same directory for timing, as it is much larger
than the `short_logs.crv` file. As a hint, reading the data portion of the array
in at one time combined with strided slicing of lists is useful here.

Bonus Bonus
-----------

Make your example a function so that it can be used in later parts of the class
to read in log files::

        def read_crv(file_name):
            ...

Copy it to the class_lib directory so that it is callable by all your other
scripts.
"""

log_file = open('long_logs.crv')

# The first line is a header that has all the log names:
header = log_file.readline()
log_names = header.split()
log_count = len(log_names)

# Everything left is data.
# Now, read in all of the data in one fell swoop, translating
# it into floating point values as we go:
value_text = log_file.read()
values = [float(val) for val in value_text.split()]

# Once this is done, we can go back through and split the "columns" out
# of the values and associating them with their log names.  This can be
# done efficiently using strided slicing. The starting position for
# each log is just its column number, and, the "stride" for the slice
# is the number of logs in the file:
logs = {}
for offset, log_name in enumerate(log_names):
    logs[log_name] = values[offset::log_count]
```

2.1.9 Person Class - Solution

person_class_solution.py:

```
"""
Write a class that represents a person with first and last name that
can be initialized like so::

    p = Person("eric", "jones")

Write a class method that returns the person's full name.

Write a __repr__ method that prints out a "nice" representation
of the person::

    Person("eric jones")
"""

class Person(object):

    def __init__(self, first, last):
        self.first = first
        self.last = last

    def full_name(self):
        return "%s %s" % (self.first, self.last)

    def __repr__(self):
        return '%s("%s")' % (self.__class__.__name__, self.full_name())


p = Person("eric", "jones")
print p.full_name()
print p
```

2.1.10 Shape Inheritance - Solution

shape_inheritance_solution.py:

```
#!/usr/bin/env python
"""
Shape Inheritance
-----------------

In this exercise, you will use class inheritance to define a few
different shapes and draw them onto an image.  All the classes will
derive from a single base class, Shape, that is defined for you.
The Shape base class requires two arguments, x and y, that are the
position of the shape in the image.  It also has two keyword arguments,
color and line_width, to specify properties of the shape.  In this
exercise, color can be 'red', 'green', or 'blue'.  The Shape class
also has a method draw(image) that will draw the shape into the
specified image.

One Shape sub-class, Square, is already defined for you.  Study its
draw(image) method and then define two more classes, Line and Rectangle.
Use these classes to draw two more shapes to the image.  You will need
```

```
to override both the __init__ and the draw method in your sub-classes.

1. The constructor for Line should take 4 values::

        Line(x1, y1, x2, y2)

   Here x1, y1 define one end point and x2, y2 define the other end point.
   color and line_width should be optional arguments.

2. The constructor for Rectangle should take 4 values::

        Rectangle(x, y, width, height)

   Again, color and line_width should be optional arguments.

Bonus
^^^^^

    Add a Circle class.

Hints
^^^^^

    The "image" has several methods to specify and also "stroke" a path.

    move_to(x, y)
        move to an x, y position
    line_to(x, y)
        add a line from the current position to x, y
    arc(x, y, radius, start_angle, end_angle)
        add an arc centered at x, y with the specified radius
        from the start_angle to end_angle (specified in radians)
    close_path()
        draw a line from the current point to the starting point
        of the path being defined
    stroke_path()
        draw all the lines that have been added to the current path
"""

from kiva.agg import GraphicsContextArray as Image


# Map color strings to RGB values.
color_dict = dict(red=(1.0, 0.0, 0.0),
                  green=(0.0, 1.0, 0.0),
                  blue=(0.0, 0.0, 1.0))

class Shape(object):

    def __init__(self, x, y, color='red', line_width=2):
        self.color = color
        self.line_width = line_width
        self.x = x
        self.y = y

    def draw(self, image):
        raise NotImplementedError
```

```
class Square(Shape):

    def __init__(self, x, y, size, color='red', line_width=2):
        super(Square, self).__init__(x, y, color=color, line_width=line_width)
        self.size = size

    def draw(self, image):
        image.set_stroke_color(color_dict[self.color])
        image.set_line_width(self.line_width)

        image.move_to(self.x, self.y)
        image.line_to(self.x+self.size, self.y)
        image.line_to(self.x+self.size, self.y+self.size)
        image.line_to(self.x, self.y+self.size)

        image.close_path()
        image.stroke_path()

class Rectangle(Shape):

    def __init__(self, x, y, width, height, color='red', line_width=2):
        super(Rectangle, self).__init__(x, y, color=color, line_width=line_width)
        self.width = width
        self.height = height

    def draw(self, image):
        image.set_stroke_color(color_dict[self.color])
        image.set_line_width(self.line_width)

        image.move_to(self.x, self.y)
        image.line_to(self.x+self.width, self.y)
        image.line_to(self.x+self.width, self.y+self.height)
        image.line_to(self.x, self.y+self.height)

        image.close_path()
        image.stroke_path()

class Line(Shape):

    def __init__(self, x1, y1, x2, y2, color='red', line_width=2):
        super(Line, self).__init__(x1, y1, color=color, line_width=line_width)
        self.x2 = x2
        self.y2 = y2

    def draw(self, image):
        image.set_stroke_color(color_dict[self.color])
        image.set_line_width(self.line_width)

        image.move_to(self.x, self.y)
        image.line_to(self.x2, self.y2)

        image.stroke_path()

class Rectangle(Shape):

    def __init__(self, x, y, width, height, color='red', line_width=2):
        super(Rectangle, self).__init__(x, y, color=color, line_width=line_width)
        self.width = width
        self.height = height

    def draw(self, image):
        image.set_stroke_color(color_dict[self.color])
        image.set_line_width(self.line_width)

        image.move_to(self.x, self.y)
        image.line_to(self.x+self.width, self.y)
        image.line_to(self.x+self.width, self.y+self.height)
        image.line_to(self.x, self.y+self.height)

        image.close_path()
        image.stroke_path()

class Circle(Shape):

    def __init__(self, x, y, radius, color='red', line_width=2):
        super(Circle, self).__init__(x, y, color=color, line_width=line_width)
        self.radius = radius

    def draw(self, image):
        image.set_stroke_color(color_dict[self.color])
        image.set_line_width(self.line_width)

        image.arc(self.x, self.y, self.radius, 0, 6.28318)
        image.close_path()
        image.stroke_path()

# Create an image that we can draw our shapes into
image_size = (300,300)
image = Image(image_size)

# Create a box and add it to the image.
box = Square(30, 30, 100, color='green')
box.draw(image)

line = Line(50, 250, 250, 50)
line.draw(image)

rect = Rectangle( 50, 50, 30, 50)
rect.draw(image)

circle = Circle( 150, 150, 60, color='blue')
circle.draw(image)

# Save the image out as a png image.
image.save('shapes.png')
```

2.1.11 Particle Class - Solution

particle_class_solution.py:

```
"""
Particle Class
--------------

This is a quick exercise to practice working with a class.
```

```
The Particle class is defined below.  In this exercise,
your task is to make a few simple changes to the class.

1. Change the __repr__() function to print "mass:" and
   "velocity:" instead of "m:" and "v:".

2. Add an energy() function, where the energy is given
   by the formula m*v**2/2.

"""


class Particle(object):
    """ Simple particle with mass and velocity attributes.
    """

    def __init__(self, mass, velocity):
        """ Constructor method.
        """

        self.mass = mass
        self.velocity = velocity

    def momentum(self):
        """ Calculate the momentum of a particle (mass*velocity).
        """
        return self.mass * self.velocity

    def energy(self):
        """ Calculate the energy (m*v**2/2) of the particle.
        """
        return 0.5 * (self.mass * self.velocity ** 2)

    def __repr__(self):
        """ A "magic" method defines object's string representation.
        """
        msg = "(mass:%2.1f, velocity:%2.1f)" % (self.mass, self.velocity)
        return msg

    def __add__(self, other):
        """ A "magic" method to overload the '+' operator by
        adding the masses and determining the velocity to conserve
        momentum.
        """
        if not isinstance(other, Particle):
            return NotImplemented
        mnew = self.mass + other.mass
        vnew = (self.momentum() + other.momentum()) / mnew
        return Particle(mnew, vnew)

if __name__ == "__main__":
    p = Particle(2.0, 13.0)
    print p
    print "Energy is", p.energy()
```

2.1.12 Near Star Catalog - Solution

sort_stars_solution.py:

```
"""
Near Star Catalog
-----------------

Read data into a list of classes and sort in various ways.

The file `stars.dat` contains data about some of the nearest stars.  Data is
arranged in the file in fixed-width fields::

    0:17    Star name
    18:28   Spectral class
    29:34   Apparent magnitude
    35:40   Absolute magnitude
    41:46   Parallax in thosandths of an arc second

A typical line looks like::

    Proxima Centauri  M5  e      11.05 15.49 771.8

This module also contains a Star class with attributes for each of those
data items.

1. Write a function `parse_stars` which returns a list of Star instances,
    one for each star in the file `stars.dat`.

2. Sort the list of stars by apparent magnitude and print names and apparent
    magnitudes of the 10 brightest stars by apparent magnitude (lower numbers
    are brighter).

3. Sort the list of stars by absolute magnitude and print names, spectral class
    and absolute magnitudes the 10 faintest stars by absolute magnitude (lower
    numbers are brighter).

4. The distance of a star from the Earth in light years is given by::

        1/parallax in arc seconds * 3.26163626

    Sort the list by distance from the Earth and print names, spectral class
    and distance from Earth of the 10 closest stars.

Bonus
^^^^^

Spectral classes are codes of a form like 'M5', consisting of a letter and
a number, plus additional data about luminosity and abnormalities in the
spectrum.  Informally, the initial part of the code gives information about
the surface temperature and colour of the star.  From hottest to coldest, the
initial letter codes are 'OBAFGKM', with 'O' blue and 'M' red.  The number
indicates a relative position between the lettes, so 'A2' is hotter than 'A3'
which is in turn hotter than 'F0'.

Sort the list of stars and print the names, spectral classes and distances
of the 10 hottest stars in the list.

(Ignore stars with no spectral class or a spectral class that doesn't start
with 'OBAFGKM'.)


Notes
^^^^^

Data from:

    Preliminary Version of the Third Catalogue of Nearby Stars
    GLIESE W., JAHREISS H.
    Astron. Rechen-Institut, Heidelberg (1991)


See :ref:`sort-stars-solution`.

"""

from __future__ import with_statement

SPECTRAL_CODES = 'OBAFGKM'

class Star(object):
    """ An class which holds data about a star"""
    def __init__(self, name, spectral_class, app_mag, abs_mag, parallax):
        self.name = name
        self.spectral_class = spectral_class
        self.app_mag = app_mag
        self.abs_mag = abs_mag
        self.parallax = parallax

    def distance(self):
        """ The distance of the star from the earth in light years"""
        return 1/self.parallax * 3.26163626


def parse_stars(filename):
    """Create a list of Star objects from a data file"""
    with open(filename) as star_file:
        stars = []
        for line in star_file.read().split('\n'):
            if line.strip() == '':
                continue
            name = line[:17].strip()
            spectral_class = line[18:28].strip()
            app_mag = float(line[29:34])
            abs_mag = float(line[35:40])
            parallax = float(line[41:46])/1000.0
            stars.append(Star(name=name, spectral_class=spectral_class,
                app_mag=app_mag, abs_mag=abs_mag, parallax=parallax))
    return stars


def key_app_mag(star):
    """Extract the apparent magnitude"""
    return star.app_mag


def key_abs_mag(star):
    """Extract the absolute magnitude"""
    return star.abs_mag


def key_distance(star):
    """Extract the distance"""
    return star.distance()


def key_spectral_class(star):
    """Return a tuple of numbers that orders spectral classes

    Returns a tuple (a, b), where a=0 if class is O, a=1 if class is B, etc
    and b is the numerical part of the code.
    """
    if not star.spectral_class:
        return (7,0)
    code = star.spectral_class.split()[0]
    letter = code[0]
    if letter not in SPECTRAL_CODES:
        return (7,0)
    letter_index = SPECTRAL_CODES.find(letter)
    if not code[1:]:
        return (7,0)
    number = float(code[1:])
    return (letter_index, number)


if __name__ == "__main__":
    stars = parse_stars('stars.dat')

    print "10 brightest stars by apparent magnitude"
    stars.sort(key=key_app_mag)
    for star in stars[:10]:
        print "  %-17s %5.2f" % (star.name, star.app_mag)
    print

    print "10 faintest stars by absolute magnitude"
    stars.sort(key=key_abs_mag)
    stars.reverse()
    for star in stars[:10]:
        print "  %-17s %-10s %5.2f" % (star.name, star.spectral_class, star.abs_mag)
    print

    print "10 closest stars"
    stars.sort(key=key_distance)
    for star in stars[:10]:
        print "  %-17s %-10s %6.3f" % (star.name, star.spectral_class, star.distance())
    print

    print "10 hottest stars"
    stars.sort(key=key_spectral_class)
    for star in stars[:10]:
        print "  %-17s %-10s %6.3f" % (star.name, star.spectral_class, star.distance())
    print
```

2.1.13 Generators - Solution

generators_solution.py:

```
"""
Generators
----------

Generators are created by function definitions which contain one or
more yield expressions in their body.  They are a much easier way to
create iterators than by creating a class and implementing the
__iter__ and next methods manually.

In this exercise you will write generator functions that create
various iterators by using the yield expression inside a function
body.

Create generators to

1. Iterate through a file by reading N bytes at a time; i.e.::

        N = 10000
        for data in chunk_reader(file, N):
            print len(data), type(data)

   should print::

        10000 <type 'str'>
        10000 <type 'str'>
        ...
        10000 <type 'str'>
        X <type 'str'>

   where X is the last number of bytes.

2. Iterate through a sequence N-items at a time; i.e.::

        for i, j, k in grouped([1,2,3,4,5,6,7], 3):
            print i, j, k
            print "="*10

   should print (note the last number is missing)::

        1, 2, 3
        ==========
        4, 5, 6
        ==========


3. Implement "izip" based on a several input sequences; i.e.::

        list(izip(x,y,z))

   should produce the same as zip(x,y,z) so that in a for loop izip and zip
   work identically, but izip does not create an intermediate list.

"""

def chunk_reader(obj, N):
    while True:
        data = obj.read(N)
        if data == '':
            break
        else:
            yield data

def grouped(seq, N):
    for i in range(0,len(seq)-N,N):
        yield seq[i:i+N]

def izip(*args):
    iters = [iter(x) for x in args]
    # StopIteration from arg.next will terminate loop.
    while True:
        res = []
        for arg in iters:
            res.append(arg.next())
        yield tuple(res)

if __name__ == "__main__":
    from cStringIO import StringIO
    val = StringIO("3"*1000)
    N = 100
    for data in chunk_reader(val, N):
        print len(data), type(data)

    print
    print "*" * 10
    print

    seq = [1,2,3,4,5,6,7]
    for i, j, k in grouped(seq, 3):
        print i, j, k
        print "="*10

    print
    print "*" * 10
    print

    for i, j in izip([1,2,3],[4,5]):
        print i, j
```

2.2 Python Libraries - Solutions

2.2.1 Column Stats (subprocess) - Solution 1

column_stats_solution.py:

```
"""
Background
~~~~~~~~~~

The script in this directory calculates the number of rows in a file that
satisfy a certain expression.
```

```
The name of the file to process can either be provided to the script from the
command line or on stdin.

Problem
~~~~~~~

Write a program to call out to calculate.py operating on sp500hst_part.txt
in this directory.

1) Using command line arguments

2) Using a pipe to communicate via stdin

Bonus
~~~~~

The script can actually take an expression to evaluate either as an
additional command line argument or as an additional string passed to
stdin.

Repeat 1) and 2) but change the expression to {6} > 50000

"""


from subprocess import Popen, PIPE

# Part 1

ret = Popen(["python", "calculate.py",
             "sp500hst_part.txt"]).wait()


# Part 2

p = Popen(["python", "calculate.py"], stdin=PIPE, stdout=PIPE)
print p.communicate(input="sp500hst_part.txt")[0].strip()
# or
# p.stdin.write("sp500hst_part.txt")
# p.stdin.close()
# print p.stdout.read().strip()


# Bonus 1

ret = Popen(["python", "calculate.py",
             "sp500hst_part.txt", "{3} < 53.0"]).wait()

# Bonus 2

p = Popen(["python", "calculate.py"], stdin=PIPE, stdout=PIPE)
p.stdin.write('sp500hst_part.txt')
p.stdin.write(' {6} > 50000')
p.stdin.close()
print p.stdout.read().strip()
```

2.2.2 Column Stats (subprocess) - Solution 2

column_stats_solution2.py:

```
"""
Background
~~~~~~~~~~

The script in this directory calculates the number of rows in a file that
satisfy a certain expression.

The name of the file to process can either be provided to the script from the
command line or on stdin.

Problem
~~~~~~~

Write a program to call out to calculate.py operating on sp500hst_part.txt
in this directory.

1) Using command line arguments

2) Using a pipe to communicate via stdin

Bonus
~~~~~

The script can actually take an expression to evaluate either as an
additional command line argument or as an additional string passed to
stdin.

Repeat 1) and 2) but change the expression to {6} > 50000

"""


from subprocess import Popen, PIPE

# Part 1

ret = Popen(["grep", '"AEE"',
             "sp500hst_part.txt"]).wait()


# Part 2

p = Popen(["python", "calculate.py"], stdin=PIPE, stdout=PIPE)
print p.communicate(input="sp500hst_part.txt")[0].strip()
# or
# p.stdin.write("sp500hst_part.txt")
# p.stdin.close()
# print p.stdout.read().strip()


# Bonus 1

ret = Popen(["python", "calculate.py",
             "sp500hst_part.txt", "{3} < 53.0"]).wait()
```

```
# Bonus 2

p = Popen(["python", "calculate.py"], stdin=PIPE, stdout=PIPE)
p.stdin.write('sp500hst_part.txt')
p.stdin.write(' {6} > 50000')
p.stdin.close()
print p.stdout.read().strip()
```

2.2.3 Dow Database - Solution

dow_database_solution.py:

```
"""
Dow Database
-------------

Topics: Database DB-API 2.0

The database table in the file 'dow2008.csv' has records holding the
daily performance of the Dow Jones Industrial Average from the
beginning of 2008.  The table has the following columns (separated by
a comma).

DATE        OPEN      HIGH      LOW       CLOSE     VOLUME      ADJ_CLOSE
2008-01-02  13261.82  13338.23  12969.42  13043.96  3452650000  13043.96
2008-01-03  13044.12  13197.43  12968.44  13056.72  3429500000  13056.72
2008-01-04  13046.56  13049.65  12740.51  12800.18  4166000000  12800.18
2008-01-07  12801.15  12984.95  12640.44  12827.49  4221260000  12827.49
2008-01-08  12820.9   12998.11  12511.03  12589.07  4705390000  12589.07
2008-01-09  12590.21  12814.97  12431.53  12735.31  5351030000  12735.31

1. Create a database table that has the same structure (use real
   for all the columns except the date column).

2. Insert all the records from dow.csv into the database.

3. Select (and print out) the records from the database that have a volume
   greater than 5.5 billion.   How many are there?

Bonus
~~~~~
1. Select the records which have a spread between high and low that is greater
   than 4% and sort them in order of that spread.

2. Select the records which have an absolute difference between open and close
   that is greater than 1% (of the open) and sort them in order of that spread.

See :ref:`dow-selection-solution`.
"""

import sqlite3 as db

# 1.

conn = db.connect(':memory:')
c = conn.cursor()
```

```
sql = """create table dow(date date, open float, high float, low float,
  close float, volume float, adj_close float)"""
c.execute(sql)

# 2.

f = open('dow2008.csv')
headers = f.readline()

sql = "insert into dow values(?,?,?,?,?,?,?)"
for line in f:
    c.execute(sql, line.strip().split(','))

conn.commit()

# 3.

sql = "select * from dow where volume > ?"
c.execute(sql, (5.5e9,))

N = 0
for row in c:
    print row
    N += 1

print "Number = ", N


# Bonus 1
sql = "select * from dow where (high-low)/low > ? order by (high-low)/low"
c.execute(sql, (0.04,))

N = 0
for row in c:
    print row
    N += 1
print "Bonus 1 number of rows: ", N

# Bonus 2
sql = "select * from dow where abs(open-close)/open > ? order by abs(open-close)/open"
c.execute(sql, (0.01,))

N = 0
for row in c:
    print row
    N += 1
print "Bonus 2 number of rows: ", N
```

2.3 Software Craftsmanship - Solutions

2.3.1 Test Driven Development Example - Solution 1

fancy_math_solution.py:

```
""" Test Driven Development Example

    Write a function called `slope` that calculates the slope
    between two points.  A point is specified by a two element
    sequence of (x,y) coordinates.

        >>> pt1 = [0.0, 0.0]
        >>> pt2 = [1.0, 2.0]
        >>> slope(pt1, pt2)
        2.0

    Use Test Driven Development.  Write your tests first in
    a separate file called tests_fancy_math.py.

    Run your tests using the "nosetests" shell command.  You can
    do this by changing to the "slope" directory where your
    fancy_math.py is defined and running "nosetests".  From IPython,
    you can run it like this:

       In [1]: cd <someplace>/exercises/slope
       In [2]: !nosestests
       ...
       --------------------------------------------------
       Ran 3 tests in 0.157s

    If you would like to see more verbose output, use the "-v"
    option:

       In [3]: !nosetests -v
       test_fancy_math.test_slope_xxx ... ok
       test_fancy_math.test_slope_yyy ... ok
       ...

    By default, nose captures all output and does not print it
    to the screen.  If you would like to see the output of print
    statements, use the "-s" flag.
"""
from __future__ import division
from numpy import Inf

def slope(pt1, pt2):
    dy = pt2[1] - pt1[1]
    dx = pt2[0] - pt1[0]
    try:
        slope = dy/dx
    except ZeroDivisionError:
        if dy > 0:
            slope = Inf
        else:
            slope = -Inf

    return slope
```

2.3.2 Code Check - Solution

code_check_solution.py:

```python
"""
Code Check
----------

This code has an assortment of bugs, and its style doesn't
conform to PEP-8.  Use pyflakes and pep8 to find and fix
the code.

You may have to install pep8 with the command:

$ easy_install pep8

It might take a few iterations before pyflakes doesn't
complain about something.

"""
from math import acos, sqrt


class Vector(object):

    def __init__(self, x, y, z):
        """ Constructor method.
        """
        self.x = x
        self.y = y
        self.z = z

    def dot(self, v):
        d = self.x * v.x + self.y * v.y + self.z * v.z
        return d

    def abs(self):
        m = sqrt(self.x ** 2 + self.y ** 2 + self.z ** 2)
        return m

    def angle(self, v):
        theta = acos(self.dot(v) / (self.abs() * v.abs()))
        return theta

    def __repr__(self):
        s = "Vector(x=%s, y=%s, z=%s)" % (self.x, self.y, self.z)
        return s


if __name__ == "__main__":
    v1 = Vector(2.0, 13.0, -1.0)
    print v1, " magnitude is", v1.abs()
    v2 = Vector(1.0, 2.0, 3.0)
    print "v1.angle(v2) =", v1.angle(v2)
```

2.4 Numpy - Solutions

2.4.1 Plotting - Solution

plotting_solution.py:

```
"""
Plotting
--------

In PyLab, create a plot display that looks like the following:

.. image:: plotting/sample_plots.png

`Photo credit: David Fettig <http://www.publicdomainpictures.net/view-image.php?image=507>`_


This is a 2x2 layout, with 3 slots occupied.

1. Sine function, with blue solid line; cosine with red '+' markers; the extents
   fit the plot exactly. Hint: see the axis() function for setting the extents
2. Sine function, with gridlines, axis labels, and title; the extents fit the
   plot exactly.
3. Image with color map; the extents run from -10 to 10, rather than the
   default.

Save the resulting plot image to a file. (Use a different file name, so you
don't overwrite the sample.)

The color map in the example is 'winter'; use 'cm.<tab>' to list the available
ones, and experiment to find one you like.

Start with the following statements::

    from scipy.misc.pilutil import imread

    x = linspace(0, 2*pi, 101)
    s = sin(x)
    c = cos(x)

    # 'flatten' creates a 2D array from a JPEG.
    img = imread('dc_metro.jpg', flatten=True)

"""


# The following imports are *not* needed in PyLab, but are needed in this file.
from numpy import linspace, pi, sin, cos
from pylab import plot, subplot, cm, imshow, xlabel, ylabel, title, grid, \
                  axis, show, savefig, gcf, figure, close, subplots_adjust

# The following import *is* needed in PyLab.
# The PyLab version of 'imread' does not read JPEGs.
from scipy.misc.pilutil import imread

x = linspace(0, 2*pi, 101)
s = sin(x)
c = cos(x)
```

```
# 'flatten' creates a 2D array from a JPEG.
img = imread('dc_metro.JPG', flatten=True)

close('all')
# 2x2 layout, first plot: sin and cos
subplot(2,2,1)
plot(x, s, 'b-', x, c, 'r+')
axis('tight')

# 2nd plot: gridlines, labels
subplot(2,2,2)
plot(x, s)
grid()
xlabel('radians')
ylabel('amplitude')
title('sin(x)')
axis('tight')

# 3rd plot, image
subplot(2,2,3)
imshow(img, extent=[-10, 10, -10, 10], cmap=cm.winter)

subplots_adjust(wspace=0.31)

show()


savefig('my_plots.png')
```

2.4.2 Calculate Derivative - Solution

calc_derivative_solution.py:

```
"""
Topics: NumPy array indexing and array math.

Use array slicing and math operations to calculate the
numerical derivative of ``sin`` from 0 to ``2*pi``.  There is no
need to use a for loop for this.

Plot the resulting values and compare to ``cos``.

Bonus
~~~~~

Implement integration of the same function using Riemann sums or the
trapezoidal rule.

"""
from numpy import linspace, pi, sin, cos, cumsum
from pylab import plot, show, subplot, legend, title

# calculate the sin() function on evenly spaced data.
x = linspace(0,2*pi,101)
y = sin(x)
```

```
# calculate the derivative dy/dx numerically.
# First, calculate the distance between adjacent pairs of
# x and y values.
dy = y[1:]-y[:-1]
dx = x[1:]-x[:-1]

# Now divide to get "rise" over "run" for each interval.
dy_dx = dy/dx

# Assuming central differences, these derivative values
# centered in-between our original sample points.
centers_x = (x[1:]+x[:-1])/2.0

# Plot our derivative calculation.  It should match up
# with the cos function since the derivative of sin is
# cos.
subplot(1,2,1)
plot(centers_x, dy_dx,'rx', centers_x, cos(centers_x),'b-')
title(r"$\rm{Derivative\ of}\ sin(x)$")

# Trapezoidal rule integration.
avg_height = (y[1:]+y[:-1])/2.0
int_sin = cumsum(dx * avg_height)

# Plot our integration against -cos(x) - -cos(0)
closed_form = -cos(x)+cos(0)
subplot(1,2,2)
plot(x[1:], int_sin,'rx', x, closed_form,'b-')
legend(('numerical', 'actual'))
title(r"$\int \, \sin(x) \, dx$")
show()
```

2.4.3 Load Array from Text File - Solution

load_text_solution.py:

```
"""
Load Array from Text File
-------------------------

0. From the IPython prompt, type::

       In [1]: loadtxt?

   to see the options on how to use the loadtxt command.


1. Use loadtxt to load in a 2D array of floating point values from
   'float_data.txt'.  The data in the file looks like::

        1 2 3 4
        5 6 7 8

   The resulting data should be a 2x4 array of floating point values.

2. In the second example, the file 'float_data_with_header.txt' has
   strings as column names in the first row::
        c1 c2 c3 c4
         1  2  3  4
         5  6  7  8

   Ignore these column names, and read the remainder of the data into
   a 2D array.

   Later on, we’ll learn how to create a "structured array" using
   these column names to create fields within an array.

BONUS:

3. A third example is more involved. It contains comments in multiple
   locations, uses multiple formats, and includes a useless column to
   skip::

    -- THIS IS THE BEGINNING OF THE FILE --
    % This is a more complex file to read!

    % Day,  Month,  Year, Useless Col, Avg Power
       01,     01,  2000,      ad766,         30
       02,     01,  2000,       t873,         41
    % we don’t have Jan 03rd!
       04,     01,  2000,       r441,         55
       05,     01,  2000,       s345,         78
       06,     01,  2000,       x273,        134 % that day was crazy
       07,     01,  2000,       x355,         42

    %-- THIS IS THE END OF THE FILE --
"""

from numpy import loadtxt

#############################################################################
# 1. Simple example loading a 2x4 array of floats from a file.
#############################################################################
ary1 = loadtxt(’float_data.txt’)

print ’example 1:’
print ary1


#############################################################################
# 2. Same example, but skipping the first row of column headers
#############################################################################
ary2 = loadtxt(’float_data_with_header.txt’, skiprows=1)

print ’example 2:’
print ary2

#############################################################################
# 3. More complex example with comments and columns to skip
#############################################################################
ary3 = loadtxt("complex_data_file.txt", delimiter=",", comments="%",
               usecols=(0,1,2,4), dtype=int, skiprows=1)

print ’example 3:’
print ary3
```

2.4.4 Filter Image - Solution

filter_image_solution.py:

```
"""
Read in the "dc_metro" image and use an averaging filter
to "smooth" the image.  Use a "5 point stencil" where
you average the current pixel with its neighboring pixels::

              0 0 0 0 0 0 0
              0 0 0 x 0 0 0
              0 0 x x x 0 0
              0 0 0 x 0 0 0
              0 0 0 0 0 0 0

Plot the image, the smoothed image, and the difference between the
two.

Bonus
~~~~~

Re-filter the image by passing the result image through the filter again. Do
this 50 times and plot the resulting image.

"""

from scipy.misc.pilutil import imread
from pylab import figure, subplot, imshow, title, show, gray, cm

def smooth(img):
    avg_img =(  img[1:-1 ,1:-1]  # center
              + img[ :-2 ,1:-1]  # top
              + img[2:   ,1:-1]  # bottom
              + img[1:-1 , :-2]  # left
              + img[1:-1 ,2:  ]  # right
              ) / 5.0
    return avg_img

# 'flatten' creates a 2D array from a JPEG.
img = imread('dc_metro.JPG', flatten=True)
avg_img = smooth(img)


figure()
# Set colormap so that images are plotted in gray scale.
gray()
# Plot the original image first
subplot(1,3,1)
imshow(img)
title('original')

# Now the filtered image.
subplot(1,3,2)
imshow(avg_img)
title('smoothed once')

# And finally the difference between the two.
subplot(1,3,3)
imshow(img[1:-1,1:-1] - avg_img)
```

```
title('difference')


# Bonus: Re-filter the image by passing the result image
#        through the filter again.  Do this 50 times and plot
#        the resulting image.

for num in range(50):
    avg_img = smooth(avg_img)

print avg_img.shape, img.shape

# Plot the original image first
figure()
subplot(1,2,1)
imshow(img)
title('original')

# Now the filtered image.
subplot(1,2,2)
imshow(avg_img)
title('smoothed 50 times')

show()
```

2.4.5 Dow Selection - Solution

dow_selection_solution.py:

```
"""

Topics: Boolean array operators, sum function, where function, plotting.

The array 'dow' is a 2-D array with each row holding the
daily performance of the Dow Jones Industrial Average from the
beginning of 2008 (dates have been removed for exercise simplicity).
The array has the following structure::

       OPEN      HIGH      LOW       CLOSE     VOLUME      ADJ_CLOSE
       13261.82  13338.23  12969.42  13043.96  3452650000  13043.96
       13044.12  13197.43  12968.44  13056.72  3429500000  13056.72
       13046.56  13049.65  12740.51  12800.18  4166000000  12800.18
       12801.15  12984.95  12640.44  12827.49  4221260000  12827.49
       12820.9   12998.11  12511.03  12589.07  4705390000  12589.07
       12590.21  12814.97  12431.53  12735.31  5351030000  12735.31

0. The data has been loaded from a .csv file for you.
1. Create a "mask" array that indicates which rows have a volume
   greater than 5.5 billion.
2. How many are there?  (hint: use sum).
3. Find the index of every row (or day) where the volume is greater
   than 5.5 billion. hint: look at the where() command.

Bonus
~~~~~
1. Plot the adjusted close for *every* day in 2008.
2. Now over-plot this plot with a 'red dot' marker for every
```

```
   day where the dow was greater than 5.5 billion.

"""

from numpy import where, loadtxt
from pylab import figure, hold, plot, show

# Constants that indicate what data is held in each column of
# the 'dow' array.
OPEN  = 0
HIGH = 1
LOW = 2
CLOSE = 3
VOLUME = 4
ADJ_CLOSE = 5

# 0. The data has been loaded from a csv file for you.

# 'dow' is our NumPy array that we will manipulate.
dow = loadtxt('dow.csv', delimiter=',')


# 1. Create a "mask" array that indicates which rows have a volume
#    greater than 5.5 billion.
high_volume_mask = dow[:,VOLUME] > 5.5e9

# 2. How many are there?  (hint: use sum).
high_volume_days = sum(high_volume_mask)
print "The dow volume has been above 5.5 billion on" \
      " %d days this year." % high_volume_days

# 3. Find the index of every row (or day) where the volume is greater
#    than 5.5 billion. hint: look at the where() command.
high_vol_index = where(high_volume_mask)[0]

# BONUS:
# 1. Plot the adjusted close for EVERY day in 2008.
# 2. Now over-plot this plot with a 'red dot' marker for every
#    day where the dow was greater than 5.5 billion.

# Create a new plot.
figure()

# Plot the adjusted close for every day of the year as a blue line.
# In the format string 'b-', 'b' means blue and '-' indicates a line.
plot(dow[:,ADJ_CLOSE],'b-')

# Plot the days where the volume was high with red dots...
plot(high_vol_index, dow[high_vol_index, ADJ_CLOSE],'ro')

# Scripts must call the plot "show" command to display the plot
# to the screen.
show()
```

2.4.6 Wind Statistics - Solution

wind_statistics_solution.py:

```
"""
Wind Statistics
----------------

Topics: Using array methods over different axes, fancy indexing.

1. The data in 'wind.data' has the following format::

        61  1  1 15.04 14.96 13.17  9.29 13.96  9.87 13.67 10.25 10.83 12.58 18.50 15.04
        61  1  2 14.71 16.88 10.83  6.50 12.62  7.67 11.50 10.04  9.79  9.67 17.54 13.83
        61  1  3 18.50 16.88 12.33 10.13 11.17  6.17 11.25  8.04  8.50  7.67 12.75 12.71

   The first three columns are year, month and day.  The
   remaining 12 columns are average windspeeds in knots at 12
   locations in Ireland on that day.

   Use the 'loadtxt' function from numpy to read the data into
   an array.

2. Calculate the min, max and mean windspeeds and standard deviation of the
   windspeeds over all the locations and all the times (a single set of numbers
   for the entire dataset).

3. Calculate the min, max and mean windspeeds and standard deviations of the
   windspeeds at each location over all the days (a different set of numbers
   for each location)

4. Calculate the min, max and mean windspeed and standard deviations of the windspeeds
   across all the locations at each day (a different set of numbers for each day)

5. Find the location which has the greatest windspeed on each day (an integer column
   number for each day).

6. Find the year, month and day on which the greatest windspeed was recorded.

7. Find the average windspeed in January for each location.

You should be able to perform all of these operations without using a for
loop or other looping construct.

Bonus
~~~~~

b1. Calculate the mean windspeed for each month in the dataset.  Treat
    January 1961 and January 1962 as *different* months.

b2. Calculate the min, max and mean windspeeds and standard deviations of the
    windspeeds across all locations for each week (assume that the first week
    starts on January 1 1961) for the first 52 weeks.

Bonus Bonus
~~~~~~~~~~~

Calculate the mean windspeed for each month without using a for loop.
(Hint: look at `searchsorted` and `add.reduceat`.)

Notes
^^^^^
```

```
These data were analyzed in detail in the following article:

   Haslett, J. and Raftery, A. E. (1989). Space-time Modelling with
   Long-memory Dependence: Assessing Ireland's Wind Power Resource
   (with Discussion). Applied Statistics 38, 1-50.


See :ref:`wind-statistics-solution`.
"""

from numpy import loadtxt, arange, searchsorted, add, zeros

wind_data = loadtxt('wind.data')

data = wind_data[:,3:]

print '2. Statistics over all values'
print '  min:', data.min()
print '  max:', data.max()
print '  mean:', data.mean()
print '  standard deviation:', data.std()
print

print '3. Statistics over all days at each location'
print '  min:', data.min(axis=0)
print '  max:', data.max(axis=0)
print '  mean:', data.mean(axis=0)
print '  standard deviation:', data.std(axis=0)
print

print '4. Statistics over all locations for each day'
print '  min:', data.min(axis=1)
print '  max:', data.max(axis=1)
print '  mean:', data.mean(axis=1)
print '  standard deviation:', data.std(axis=1)
print

print '5. Statistics over all days at each location'
print '  daily max location:', data.argmax(axis=1)
print

daily_max = data.max(axis=1)
max_row = daily_max.argmax()

print '6. Day of maximum reading'
print '  Year:', int(wind_data[max_row,0])
print '  Month:', int(wind_data[max_row,1])
print '  Day:', int(wind_data[max_row,2])
print

january_indices = wind_data[:,1] == 1
january_data = data[january_indices]

print '7. Statistics for January'
print '  mean:', january_data.mean(axis=0)
print

# Bonus
# compute the month number for each day in the dataset
months = (wind_data[:,0]-61)*12 + wind_data[:,1] - 1

# get set of unique months
month_values = set(months)

# initialize an array to hold the result
monthly_means = zeros(len(month_values))

for month in month_values:
    # find the rows that correspond to the current month
    day_indices = (months == month)

    # extract the data for the current month using fancy indexing
    month_data = data[day_indices]

    # find the mean
    monthly_means[month] = month_data.mean()

    # Note: experts might do this all-in one
    # monthly_means[month] = data[months==month].mean()

# In fact the whole for loop could reduce to the following one-liner
# monthly_means = array([data[months==month].mean() for month in month_values])


print "Bonus"
print "  mean:", monthly_means
print

# Bonus b2...
# Extract the data for the first 52 weeks. Then reshape the array to put
# on the same line 7 days worth of data for all locations. Let Numpy
# figure out the number of lines needed to do so
weekly_data = data[:52*7].reshape(-1, 7*12)

print 'b2. Weekly statistics over all locations'
print '  min:', weekly_data.min(axis=1)
print '  max:', weekly_data.max(axis=1)
print '  mean:', weekly_data.mean(axis=1)
print '  standard deviation:', weekly_data.std(axis=1)
print

# Bonus Bonus : this is really tricky...

# compute the month number for each day in the dataset
months = (wind_data[:,0]-61)*12 + wind_data[:,1] - 1

# find the indices for the start of each month
# this is a useful trick - we use range from 0 to the
# number of months + 1 and searchsorted to find the insertion
# points for each.
month_indices = searchsorted(months, arange(months[-1]+2))

# now use add.reduceat to get the sum at each location
monthly_loc_totals = add.reduceat(data, month_indices[:-1])

# now use add to find the sum across all locations for each month
```

```
monthly_totals = monthly_loc_totals.sum(axis=1)

# now find total number of measurements for each month
month_days = month_indices[1:] - month_indices[:-1]
measurement_count = month_days*12

# compute the mean
monthly_means = monthly_totals/measurement_count

print "Bonus Bonus"
print "  mean:", monthly_means

# Notes: this method relies on the fact that the months are contiguous in the
# data set - the method used in the bonus section works for non-contiguous
# days.
```

2.4.7 Sinc Function - Solution

sinc_function_solution.py:

```
"""
Topics: Broadcasting, Fancy Indexing

Calculate the sinc function: sin(r)/r.  Use a Cartesian x,y grid
and calculate ``r = sqrt(x**2+y**2)`` with 0 in the center of the grid.
Calculate the function for -15,15 for both x and y.
"""

from numpy import linspace, sin, sqrt, newaxis
from pylab import imshow, gray, show

x = linspace(-15,15,101)
# flip y up so that it is a "column" vector.
y = linspace(-15,15,101)[:,newaxis]

# because of broadcasting rules, r is 2D.
r = sqrt(x**2+y**2)

# calculate our function.
sinc = sin(r)/r

# replace any location where r is 0 with 1.0
sinc[r==0] = 1.0

imshow(sinc, extent=[-15,15,-15,15])
gray()
show()
```

2.4.8 Structured Array - Solution

structured_array_solution.py:

```
""" Read columns of data into a structured array using loadtxt.

    1. The data in 'short_logs.crv' has the following format::
           DEPTH          CALI       S-SONIC   ...
           8744.5000   -999.2500   -999.2500   ...
           8745.0000   -999.2500   -999.2500   ...
           8745.5000   -999.2500   -999.2500   ...

       Here the first row defines a set of names for the columns
       of data in the file.  Use these column names to define a
       dtype for a structured array that will have fields 'DEPTH',
       'CALI', etc.  Assume all the data is of the float64 data
       format.

    2. Use the 'loadtxt' method from numpy to read the data from
       the file into a structured array with the dtype created
       in (1).  Name this array 'logs'

    3. The 'logs' array is nice for retrieving columns from the data.
       For example, logs['DEPTH'] returns the values from the DEPTH
       column of the data.  For row-based or array-wide operations,
       it is more convenient to have a 2D view into the data, as if it
       is a simple 2D array of float64 values.

       Create a 2D array called 'logs_2d' using the view operation.
       Be sure the 2D array has the same number of columns as in the
       data file.

    4. -999.25 is a "special" value in this data set.  It is
       intended to represent missing data.  Replace all of these
       values with NaNs.  Is this easier with the 'logs' array
       or the 'logs_2d' array?

    5. Create a mask for all the "complete" rows in the array.
       A complete row is one that doesn't have any NaN values measured
       in that row.

       HINT: The ``all`` function is also useful here.

    6. Plot the VP vs VS logs for the "complete" rows.
"""
from numpy import dtype, loadtxt, float64, NaN, isfinite, all
from pylab import plot, show, xlabel, ylabel

# Open the file.
log_file = open('short_logs.crv')

# 1.Create a dtype from the names in the file header.
header = log_file.readline()
log_names = header.split()

# Construct the array "dtype" that describes the data.  All fields
# are 8 byte (64 bit) floating point.
fields = zip(log_names, ['f8']*len(log_names))
fields_dtype = dtype(fields)

#2. Use loadtxt to load the data into a structured array.
logs = loadtxt(log_file, dtype=fields_dtype)

# 3. Make a 2D, float64 view of the data.
#    The -1 value for the row shape means that numpy should
#    make this dimension whatever it needs to be so that
#    rows*cols = size for the array.
values = logs.view(float64)
values.shape = -1, len(fields)

# 4. Relace any values that are -999.25 with NaNs.
values[values==-999.25] = NaN

# 5. Make a mask for all the rows that don't have any missing values.
#    Pull out these samples from the logs array into a separate array.
data_mask = all(isfinite(values), axis=-1)
good_logs = logs[data_mask]


# 6. Plot VP vs. VS for the "complete rows.
plot(good_logs['VS'], good_logs['VP'], 'o')
xlabel('VS')
ylabel('VP')
show()
```

2.5 Scipy - Solutions

2.5.1 Statistics Functions - Solution

stats_functions_solution.py:

```
"""
1. Import stats from scipy, and look at its docstring to see
   what is available in the stats module::

       In [1]: from scipy import stats
       In [2]: stats?

2. Look at the docstring for the normal distribution::

       In [3]: stats.norm?

   You'll notice it has these parameters:

      loc:
        This variable shifts the distribution left or right.
        For a normal distribution, this is the mean.  It defaults to 0.
      scale:
        For a normal distribution, this is the standard deviation.

3. Create a single plot of the pdf of a normal distribution with mean=0
   and standard deviation ranging from 1 to 3.  Plot this on the range
   from -10 to 10.

4. Create a "frozen" normal distribution object with mean=20.0,
   and standard deviation=3::

       my_distribution = stats.norm(20.0, 3.0)

5. Plot this distribution's pdf and cdf side by side in two
   separate plots.
```

```
6. Get 5000 random variable samples from the distribution, and use
   the stats.norm.fit method to estimate its parameters.
   Plot the histogram of the random variables as well as the
   pdf for the estimated distribution. (Try using stats.histogram.
   There is also pylab.hist which makes life easier.)

"""

from numpy import linspace, arange
from scipy.stats import norm, histogram

from pylab import plot, clf, show, figure, legend, title, subplot, \
                  bar, hist

x = linspace(-10,10, 1001)

# 2. Create a single plot with the pdf of a normal distribution with
#    mean=0 and std ranging from 1 to 3.  Plot this on the range from
#    -10 to 10.

for std in linspace(1, 3, 5):
    plot(x, norm.pdf(x, scale=std), label="std=%s" % std)
legend()
title("Standard Deviation")


#    3. Create a "frozen" normal distribution object with mean=20.0,
#       and standard devation=3.

my_dist = norm(20.0, 3.0)


# 4. Plot this distribution's pdf and cdf side by side in two
#    separate plots.
x = linspace(0, 40, 1001)

figure()
subplot(1,2,1)
plot(x, my_dist.pdf(x))
title("PDF of norm(20, 3)")

subplot(1,2,2)
plot(x, my_dist.cdf(x))
title("CDF of norm(20, 3)")


# 5. Get 5000 random variable samples from the distribution, and use
#    the stats.norm.fit method to estimate its parameters.
#    Plot the histogram of the random variables as well as the
#    pdf for the estimated distribution.

random_values = my_dist.rvs(5000)

# Calculate the histogram using stats.histogram.
num_bins = 50
bin_counts, min_bin, bin_width, outside = histogram(random_values,
                                                    numbins=num_bins)
bin_x = min_bin + bin_width * arange(num_bins)
```

```
# Normalize the bin counts so that they integrate to 1.0 like
# a pdf would.
hist_pdf = bin_counts/(len(random_values)*bin_width)

mean_est, std_est = norm.fit(random_values)
print "estimate of mean, std:", mean_est, std_est

# Plotting.
figure()
bar(bin_x, hist_pdf, width=bin_width)

# or for the lazy, use pylab.hist...
#hist(random_values, bins=50, normed=True)

plot(bin_x, norm(mean_est, std_est).pdf(bin_x), 'r', linewidth=2)
show()
```

2.5.2 Monte Carlo Options - Solution

montecarlo_options_solution.py:

```
"""

Background
~~~~~~~~~~

One approach to pricing options is to simulate the instrument price over the
lifetime of the option using Monte Carlo simulation (assuming some model). The
resulting random walk of the stock price can be used to determine the option
pay-off.

The option price is then usually calculated as the average value of the
simulated pay-offs discounted at the risk-free rate of return (exp(-r*T)).

An often-used model for the stock price at time *T* is that it is log-normally
distributed where log(ST) is normally distributed with::

     mean - log(S0) + (mu-sigma**2 / 2) * T

and::

     variance - sigma**2 * T

This implies that the *ST* is log-normally distributed with shape parameter
``sigma * sqrt(T)`` and scale parameter ``S0*exp((mu-sigma**2/2)*T)``.

For option pricing, *mu* is the risk-free rate of return and *sigma* is
the volatility.

The value of a call option at maturity is ST-K if ST>K and 0 if ST<K.

The value of a put option at maturity is K-ST if K>ST and 0 if ST>K.

Problem
~~~~~~~

1) Create a function that uses a Monte Carlo method to price vanilla
```

```
   call and put options, by drawing samples from a log-normal distribution.

2) Create a function that uses a Monte-Carlo method to price vanilla
   call and put options, by drawing samples from a normal distribution.

3) Bonus, compare 1 and 2 against the Black-Scholes method of pricing.

"""

from numpy import exp, sqrt, maximum
from scipy.stats import lognorm, norm


def mcprices(S0, K, T, r, sigma, N=5000):
    """
    Call and put option prices using log-normal Monte-Carlo method

    Parameters
    ----------
    S0 :
        Current price of the underlying stock
    K :
        Strike price of the option
    T :
        Time to maturity of the option
    r :
        Risk-free rate of return (continuously-compounded)
    sigma :
        Stock price volatility
    N :
        Number of stock prices to simulate

    Returns
    -------
    c :
        Call option price
    p :
        Put option price

    Notes
    -----
    r, T, and sigma must be expressed in consistent units of time
    """
    scale = S0*exp((r-sigma**2/2)*T)
    shape = sigma*sqrt(T)
    ST_sim = lognorm.rvs(shape,scale=scale, size=N)
    call_pay_off = maximum(ST_sim - K, 0)
    put_pay_off = maximum(K - ST_sim, 0)
    discount = exp(-r*T)
    return (call_pay_off.mean()*discount,
            put_pay_off.mean()*discount)

def mcprices2(S0, K, T, r, sigma, N=5000):
    """
    Call and put option prices using normal Monte-Carlo method

    Parameters
    ----------
    S0 :
        Current price of the underlying stock
    K :
        Strike price of the option
    T :
        Time to maturity of the option
    r :
        Risk-free rate of return (continuously-compounded)
    sigma :
        Stock price volatility

    Returns
    -------
    c :
        Call option price
    p :
        Put option price

    Notes
    -----
    r, T, and sigma must be expressed in consistent units of time

    """
    mean = (r - sigma**2/2)*T
    std = sigma*sqrt(T)
    values = norm.rvs(loc=mean, scale=std, size=N)
    ST_sim = S0*exp(values)
    call_pay_off = maximum(ST_sim - K, 0)
    put_pay_off = maximum(K-ST_sim, 0)
    discount = exp(-r*T)
    return (call_pay_off.mean()*discount,
            put_pay_off.mean()*discount)


if __name__ == "__main__":
    S0 = 40 # $40 current price
    K  = 42 # $46 strike price
    T = 3/12.0  # 3 months in units of years
    r = 0.01 # annual risk-free rate of return
    sigma = 0.30 # volatility per annum
    msg = "Call price: %4.2f; Put price: %4.2f"
    c1, p1 = mcprices(S0, K, T, r, sigma, N=10000)
    print msg % (c1, p1)
    c1, p1 = mcprices2(S0, K, T, r, sigma, N=10000)
    print msg % (c1, p1)

    # FIXME: this is not a new-student-friendly technique
    # Get black scholes equation from other directory
    #
    import os, sys
    fullpath = os.path.abspath(__file__)
    direc = os.sep.join([os.path.split(os.path.split(fullpath)[0])[0],
                         'black_scholes'])
    sys.path.insert(0, direc)
    from black_scholes_solution import bsprices
    c1, p1 = bsprices(S0, K, T, r, sigma)
    print msg % (c1, p1)
    sys.path.pop()
```

2.5.3 Gaussian Kernel Density Estimation - Solution

gaussian_kde_solution.py:

```
"""
Estimate a probability density function for a set of random
samples using a Gaussian Kernel Density estimator.

Create 100 normally distributed samples with mean=1.0
and std=0.0 using the stats.norm.rvs method.

Use stats.kde.gaussian_kde to estimate the pdf for the
distribution of these samples.  Compare that to the analytic
pdf for the distribution as well as the histogram of the
actual samples over the interval -3, 10.

Bonus:

Construct a set of samples (200 or so) from two different normal
distributions, mean=0.0, std=0.5 and mean=5.0, std=1.0.
As with the first example, compare this against the analytic
pdf as well as the histogram of the actual samples over the
interval -3, 10.

"""

from numpy import linspace, concatenate
from scipy import stats
from pylab import figure, plot, hold, legend, show, hist

# Create a normal distribution object and get 100
# samples from it.
N = 100
dist = stats.norm(0.0, 1.0)
samples = dist.rvs(size=N)

# Create gaussian_kde object with the given samples.
distribution_estimate = stats.kde.gaussian_kde(samples)

# Evaluate the pdf using the gaussian_kde object as well
# as the analytic solution available on the stats.norm
x = linspace(-3,10,101)
y = distribution_estimate.evaluate(x)
actual = dist.pdf(x)

# Display the results
figure()
hist(samples,bins=10,normed=True,fc='w')
plot(x, actual, 'b-', label='actual',linewidth=3)
plot(x, y, 'r-', label='estimate', linewidth=3)
legend()
show()

# Bonus

# Create two distributions and concatenate them into a single
# array of samples.
N=100
dist1 = stats.norm(0.0, 0.5)
```

```
dist2 = stats.norm(5.0, 1.0)
samples = concatenate([dist1.rvs(size=N), dist2.rvs(size=N)])

# Construct a gaussian_kde object from the samples and
# use it to estimate the pdf.  Also, calculate the analytic
# pdf over the same interval.
distribution_estimate = stats.kde.gaussian_kde(samples)
y = distribution_estimate.evaluate(x)
actual =  (dist1.pdf(x) + dist2.pdf(x)) / 2.0

# Display the results
figure()
hist(samples,bins=20,normed=True,fc='w')
plot(x, actual, 'b-', label='actual',linewidth=3)
plot(x, y, 'r-', label='estimate', linewidth=3)
legend()
show()
```

2.5.4 Estimate Volatility - Solution

estimate_volatility_solution.py:

```
"""
Background
~~~~~~~~~~

A standard model of price fluctuation is::

   dS/S = mu dt + sigma * epsilon * sqrt(dt)

where:

* *S* is the stock price.
* *dS* is the change in stock price.
* *mu* is the rate of return.
* *dt* is the time interval.
* *epsilon* is a normal random variable with mean 0 and variance 1 that is
  uncorrelated with other time intervals.
* *sigma* is the volatility.

It is desired to make estimates of *sigma* from historical price information.
There are simple approaches to do this that assume volatility is constant over a
period of time. It is more accurate, however, to recognize that *sigma* changes
with each day and therefore should be estimated at each day. To effectively do
this from historical price data alone, some kind of model is needed.

The GARCH(1,1) model for volatility at time *n*, estimated from data
available at the end of time *n-1* is::

   sigma_n**2 = gamma V_L + alpha u_{n-1}**2 + beta sigma_{n-1}**2

where:

* *V_L* is long-running volatility
* ``alpha+beta+gamma = 1``
* ``u_n = log(S_n / S_{n-1})`` or ``(S_n - S_{n-1})/S_{n-1}``
```

```
Estimating *V_L* can be done as the mean of ``u_n**2`` (variance of *u_n*).
Estimating parameters *alpha* and *beta* is done by finding the parameters that
maximize the likelihood that the data *u_n* would be observed. If it is assumed
that the *u_n* are normally distributed with mean 0 and variance *sigma_n*, this
is equivalent to finding *alpha* and *beta* that minimize::

   L(alpha, beta) = sum_{n}(log(sigma_n**2) + u_n**2 / sigma_n**2)

where ``sigma_n**2`` is computed as above.

Problem
~~~~~~~

1) Create a function to read in daily data from :file:`sp500hst.txt` for the
   S&P 500 for a particular stock. The file format is::

       date, symbol, open, high, low, close, volume

2) Create a function to estimate volatility per annum for a specific
   number of periods (assume 252 trading days in a year).

3) Create a function to compute ``sigma**2_n` for each *n* from *alpha* and
   *beta* and ``u_n**2`` (you may need to use a for loop for this).  Use *V_L*
   to start the recursion.

4) Create a function that will estimate volatility using GARCH(1,1)
   approach by minimizing ``L(alpha, beta)``.

5) Use the functions to construct a plot of volatility per annum for a
   stock of your choice (use 'AAPL' if you don't have a preference)
   using quarterly, monthly, and GARCH(1,1) estimates.

You may find the repeat method useful to extend quarterly and monthly
estimates to be the same size as GARCH(1,1) estimates.

"""

import numpy as np
from scipy.optimize import fmin
TRADING_DAYS = 252


fmt = [('date', int), ('symbol', 'S4'), ('open', float),
       ('high', float), ('low', float), ('close', float),
       ('volume', int)]


def read_data(filename):
    """Read all historical price data in filename into a structured
    numpy array with fields:

    date, symbol, open, high, low, close, volume
    """
    # converter functions
    types = [int, str, float, float, float, float, int]
    datafile = open(filename)
    newdata = []
    for line in datafile:
```

```
        converted = [types[i](x) for i,x in enumerate(line.split(','))]
        newdata.append(tuple(converted))
    return np.array(newdata, dtype=fmt)

def read_symbol(filename, symbol):
    """Read all historical price data for a particular symbol in filename
    into a structured numpy array with fields:

    date, symbol, open, high, low, close, volume
    """
    # converter functions
    types = [int, str, float, float, float, float, int]
    datafile = open(filename)
    newdata = []
    for line in datafile:
        if symbol not in line:
            continue
        converted = [types[i](x) for i,x in enumerate(line.split(','))]
        newdata.append(tuple(converted))
    return np.array(newdata, dtype=fmt)

def volatility(S, periods=4, repeat=False):
    """Estimate of volatility using the entire data set
    divided into periods.  If repeat is True, then copy the
    estimate so that len(sigma) == len(S)-1
    """
    N = len(S)
    div = N//periods
    S = S[:periods*div]
    # place each quarter on its own row
    S = S.reshape(periods,-1)
    # Compute u
    u = np.log(S[:,1:]/S[:,:-1])
    # Estimate volatility per annum
    #   by adjusting daily volatility calculation
    sigma = np.sqrt(u.var(axis=-1)*TRADING_DAYS)
    if repeat:
        sigma = sigma.repeat(S.shape[-1])
    return sigma[1:]

def sigmasq_g(usq, alpha, beta):
    """sigma_n**2 assuming the GARCH(1,1) model of::

        sigma_n**2 = gamma*VL + alpha*sigma_n**2 + beta*u_n**2

    where gamma + alpha + beta = 1
    and  VL = mean(usq)
    """
    sigmasq = np.empty_like(usq)
    VL = usq.mean()
    sigmasq[0] = VL
    omega = VL*(1-alpha-beta)
    for i in range(1, len(usq)):
        sigmasq[i] = omega + alpha*sigmasq[i-1] + beta * usq[i-1]
    return sigmasq

# Function to minimize to find parameters of GARCH model.
def _minfunc(x, usq):
    alpha, beta = x
    sigsq = sigmasq_g(usq, alpha, beta)
    return (np.log(sigsq) + usq / sigsq).sum()

def garch_volatility(S):
    """Volatility per annum for each day computed from historical
    close price information using the GARCH(1,1) and maximum
    likelihood estimation of the parameters.
    """
    x0 = [0.5, 0.5]
    usq = np.log(S[1:]/S[:-1])**2
    xopt = fmin(_minfunc, x0, args=(usq,))
    sigmasq = sigmasq_g(usq, *xopt)
    print "alpha = ", xopt[0]
    print "beta = ", xopt[1]
    print "V_L = ", usq.mean()
    return np.sqrt(sigmasq*TRADING_DAYS)


if __name__ == "__main__":
    from pylab import plot, title, xlabel, ylabel, legend, show
    stock = 'MSFT'
    data = read_symbol('sp500hst.txt', stock)
    S = data['close']
    sig_4 = volatility(S, 4, repeat=True)
    sig_12 = volatility(S, 12, repeat=True)
    sig_g = garch_volatility(S)
    plot(sig_g,label='GARCH(1,1)')
    plot(sig_12,label='Monthly average')
    plot(sig_4, label='Quarterly average')
    title('Volatility estimates')
    xlabel('trading day')
    ylabel('volatility per annum')
    legend(loc='best')
    show()
```

2.5.5 Using fsolve - Solution

solve_function_solution.py:

```
"""
1. Setup the system of equations::

        f_0 = x**2 + y**2 - 2.0
        f_1 = x**2 - y**2 - 1.0

2. Find x_0 and x_1 so that f_0==0 and f_1==0.
   Hint: See scipy.optimize.fsolve

3. Create images of f_0 and f_1 around 0 and
   plot the guess point and the solution point.

   http://www.scipy.org/doc/api_docs/scipy.optimize.minpack.html
"""

# Numeric library imports
from numpy import array, ogrid
from scipy import optimize

# Plotting functions
from pylab import subplot, imshow, hold, figure, clf, plot, \
                    colorbar, title, show

# 1. Create the non-linear map

def funcv(x):
    """ System of equations to solve for. The input x has 2 elements,
        and the function returns two results.
    """
    f0 = x[0]**2 + x[1]**2 - 2.0
    f1 = x[0]**2 - x[1]**2 - 1.0
    return f0, f1

# 2. Solve the system of equations.

# Set a starting point for the solver
x_guess = array([4, 4])

# Solve the equation using fsolve
# x_guess is changed in place, so were going to make a copy...

x_opt = optimize.fsolve(funcv, x_guess.copy())
print 'optimal x:', x_opt
print 'optimal f(x):', funcv(x_opt)


# 3. Plot the results and the guess:

# Make some pretty plots to show the function space as well
# as the solver starting point and the solution.

# Create 2D arrays x and y and evaluate them so that we
# can get the results for f0 and f1 in the system of equations.
x,y = ogrid[-6:6:100j,-6:6:100j]
f0, f1 = funcv([x,y])

# Set up a plot of f0 and f1 vs. x and y and show the
# starting and ending point of the solver on each plot.
figure(1)
clf()
subplot(1,2,1)
imshow(f0, extent=(-6, 6, -6, 6), vmin=-20, vmax=50)
hold(True)
plot([x_guess[0]], [x_guess[1]], 'go')
plot([x_opt[0]], [x_opt[1]], 'ro')
colorbar()
title(r'$f_0$')


subplot(1,2,2)
imshow(f1, extent=(-6, 6, -6, 6), vmin=-20, vmax=50)
hold(True)
plot([x_guess[0]], [x_guess[1]], 'go')
plot([x_opt[0]], [x_opt[1]], 'ro')
colorbar()
```

```
title(r'$f_1$')

show()
```

2.5.6 Black-Scholes Pricing - Solution

black_scholes_solution.py:

```
"""
Black-Scholes Models
--------------------

Background
~~~~~~~~~~
The Black-Scholes option pricing models for European-style options on
a non-dividend paying stock are::

    c = S0 * N(d1) - K * exp(-r*T)* N(d2)  for a call option and

    p = K*exp(-r*T)*N(-d2) - S0 * N(-d1)   for a put option

where::

    d1 = (log(S0/K) + (r + sigma**2 / 2)*T) / (sigma * sqrt(T))
    d2 = d1 - sigma * sqrt(T)

Also:

* :func:`log` is the natural logarithm.
* ``N(x)`` is the cumulative density function of a standardized normal distribution.
* *S0* is the current price of the stock.
* *K* is the strike price of the option.
* *T* is the time to maturity of the option.
* *r* is the (continuously-compounded) risk-free rate of return.
* *sigma* is the stock price volatility.

Problem
~~~~~~~

1. Create a function that returns the call and put options prices
   for using the Black-Scholes formula and the inputs of
   *S0*, *K*, *T*, *r*, and *sigma*.

   Hint:  You will need scipy.special.ndtr or scipy.stats.norm.cdf.
   Notice that N(x) + N(-x) = 1.

2. The one parameter in the Black-Scholes formula that is not
   readily available is *sigma*.  Suppose you observe that the price
   of a call option is *cT*.  The value of *sigma* that would produce
   the observed value of *cT* is called the "implied volatility".
   Construct a function that calculates the implied volatility from
   *S0*, *K*, *T*, *r*, and *cT*.

  Hint:  Use a root-finding technique such as scipy.optimize.fsolve.
  (or scipy.optimize.brentq since this is a one-variable root-finding problem).

3. Repeat #2, but use the observed put option price to calculate
```

```
   implied volatility.

4) Bonus:  Make the implied volatility functions work for vector inputs (at
   least on the call and put prices)

See: :ref:`black-scholes-solution`.
"""

# Ensure integer values for prices, etc. will work correctly.
from __future__ import division

from numpy import log, exp, sqrt

from scipy.stats import norm

def bsprices(S0, K, T, r, sigma):
    """Black-Scholes call and put option pricing

    Parameters
    ----------
    S0 :
        Current price of the underlying stock
    K :
        Strike price of the option
    T :
        Time to maturity of the option
    r :
        Risk-free rate of return (continuously-compounded)
    sigma :
        Stock price volatility

    Returns
    -------
    c :
        Call option price
    p :
        Put option price

    Notes
    -----
    r, T, and sigma must be expressed in consistent units of time
    """

    x0 = sigma * sqrt(T)
    erT = exp(-r*T)
    d1 = (log(S0/K) + (r + sigma**2 / 2) * T) / x0
    d2 = d1 - x0
    Nd1 = norm.cdf(d1)
    Nd2 = norm.cdf(d2)
    c = S0*Nd1 - K*erT*Nd2
    p = K*erT*(1-Nd2) - S0*(1-Nd1)
    return c, p


if __name__ == "__main__":
    S0 = 100    # $100 stock price
    K = 105     # $105 strike price
    T = 3/12    # 3 month period
    r = 0.004   # 3 month T-bill
    sigma = .3 # 30% per annum
    call, put = bsprices(S0, K, T, r, sigma)
    print "Call and Put Prices: %3.2f, %3.2f" % (call, put)
```

2.5.7 Black-Scholes Implied Volatility - Solution

black_scholes_implied_volatility_solution.py:

```
"""
Black-Scholes Implied Volatility
--------------------------------

Background
~~~~~~~~~~
The Black-Scholes option pricing models for European-style options on
a non-dividend paying stock are::

    c = S0 * N(d1) - K * exp(-r*T)* N(d2)  for a call option and

    p = K*exp(-r*T)*N(-d2) - S0 * N(-d1)   for a put option

where::

    d1 = (log(S0/K) + (r + sigma**2 / 2)*T) / (sigma * sqrt(T))
    d2 = d1 - sigma * sqrt(T)

Also:

* :func:`log` is the natural logarithm.
* ``N(x)`` is the cumulative density function of a standardized normal distribution.
* *S0* is the current price of the stock.
* *K* is the strike price of the option.
* *T* is the time to maturity of the option.
* *r* is the (continuously-compounded) risk-free rate of return.
* *sigma* is the stock price volatility.

Problem
~~~~~~~

1. The one parameter in the Black-Scholes formula that is not
   readily available is *sigma*.  Suppose you observe that the price
   of a call option is *cT*.  The value of *sigma* that would produce
   the observed value of *cT* is called the "implied volatility".
   Construct a function that calculates the implied volatility from
   *S0*, *K*, *T*, *r*, and *cT*.

   Hint:  Use a root-finding technique such as scipy.optimize.fsolve.
   (or scipy.optimize.brentq since this is a one-variable root-finding problem).

2. Repeat #1, but use the observed put option price to calculate
   implied volatility.

3. Bonus:  Make the implied volatility functions work for vector inputs (at
   least on the call and put prices)

"""
```

```
# Ensure integer values for prices, etc. will work correctly.
from __future__ import division

from numpy import log, exp, sqrt, ones_like

from scipy.stats import norm
from scipy.optimize import fsolve

# FIXME: this is not a new-student-friendly technique
# patch up an import from the black_scholes prices solution.
import os, sys
fullpath = os.path.abspath(__file__)
direc = os.sep.join([os.path.split(os.path.split(fullpath)[0])[0],
                    'black_scholes'])
sys.path.insert(0, direc)
from black_scholes_solution import bsprices

def _call_zerofunc(sigma, S0, K, T, r, c):
    return bsprices(S0, K, T, r, sigma)[0] - c

def _put_zerofunc(sigma, S0, K, T, r, p):
    return bsprices(S0, K, T, r, sigma)[1] - p


def imp_volatility_call(S0, K, T, r, cT):
    """Implied volatility from actual call price

    Parameters
    ----------
    S0 :
        Current price of the underlying stock
    K :
        Strike price of the option
    T :
        Time to maturity of the option
    r :
        Risk-free rate of return (continuously-compounded)
    cT :
        Observed price of the call option at maturity

    Returns
    -------
    sigma :
        Implied volatility that gives a Black-Scholes call
          option price equal to the observed price

    Notes
    -----
    r and T must be expressed in consistent units of time
    """
    # Make sure solution works for vector inputs on cT
    sigma0 = 0.5*ones_like(cT)
    return fsolve(_call_zerofunc, sigma0, args=(S0, K, T, r, cT))

def imp_volatility_put(S0, K, T, r, pT):
    """Implied volatility from actual call price

    Parameters
    ----------
    S0 :
        Current price of the underlying stock
    K :
        Strike price of the option
    T :
        Time to maturity of the option
    r :
        Risk-free rate of return (continuously-compounded)
    pT :
        Observed price of the put option at maturity

    Returns
    -------
    sigma :
        Implied volatility that gives a Black-Scholes call
          option price equal to the observed price

    Notes
    -----
    r and T must be expressed in consistent units of time
    """
    # Make sure solution works for vector inputs on pT
    sigma0 = 0.5*ones_like(pT)
    return fsolve(_put_zerofunc, sigma0, args=(S0, K, T, r, pT))


S0 = 100    # $100 stock price
K = 105     # $105 strike price
T = 3/12    # 3 month period
r = 0.004   # 3 month T-bill
cT = 3.98   # Observed price of call option
pT = 8.88   # Observed price of put option
print "Implied volatility based on call price: ", imp_volatility_call(S0, K, T, r, cT)
print "Implied volatility based on put price: ",  imp_volatility_call(S0, K, T, r, pT)
```

2.5.8 Data Fitting - Solution

data_fitting_solution.py:

```
"""
1. Define a function with four arguments, x, a, b, and c, that computes::

       y = a*exp(-b*x) + c

   (This is done for you in the code below.)

   Then create an array x of 75 evenly spaced values in the interval
   0 <= x <= 5, and an array y = f(x,a,b,c) with a=2.0, b = 0.76, c=0.1.

2. Now use scipy.stats.norm to create a noisy signal by adding Gaussian
   noise with mean 0.0 and standard deviation 0.2 to y.

3. Calculate 1st, 2nd and 3rd degree polynomial functions to fit to the data.
   Use the polyfit and poly1d functions from scipy.

4. Do a least squares fit to the orignal exponential function using
```

```
   scipy.curve_fit.

"""

from pylab import plot, title, show, hold, legend, subplot
from numpy import exp, linspace
from scipy import polyfit, poly1d
from scipy.stats import norm
from scipy.optimize import curve_fit

# 1. Define the function and create the signal.

def function(x, a, b, c):
    y = a*exp(-b*x) + c
    return y

a = 2.0
b = 0.76
c = 0.1
x = linspace(0, 5.0, 75)
y = function(x, a, b, c)

# 2. Now add some noise.

noisy_y = y + norm.rvs(loc=0, scale=0.2, size=y.shape)

subplot(1, 3, 1)
plot(x, noisy_y, '.', label="Noisy")
plot(x, y, label="Original", linewidth=2)
title("$%3.2fe^{-%3.2fx}+%3.2f$" % (a,b,c))
legend()

# 3. polynomial fit for 1st, 2nd, and 3rd degree.

subplot(1, 3, 2)
plot(x, noisy_y, '.')
hold('on')
for deg in [1,2,3]:
    # Compute the coefficients of the polynomial function of degree deg.
    coef = polyfit(x, noisy_y, deg)
    # Create a python function `poly' that computes values of the polynomial.
    poly = poly1d(coef)
    # Evalate the polynomial at x and plot it.
    poly_y = poly(x)
    plot(x, poly_y, linewidth=2, label="order=%d" % deg)
title("Polynomial Fit")
legend()

# 4. Use scipy.curve_fit to fit the actual function.

best, pcov = curve_fit(function, x, noisy_y)

fit_y = function(x, *best)

subplot(1, 3, 3)
plot(x, noisy_y, 'b.', label="Noisy")
plot(x, y, label="Original", linewidth=2)
label = r"$%3.2fe^{-%3.2fx}+%3.2f$" % tuple(best)
```

```
plot(x, fit_y, label=label, linewidth=2)
title("Exponential Fit")
legend()
show()
```

2.6 Interface With Other Languages - Solutions

2.6.1 Fancy science (hand wrap) - Solution

setup_handwrap_solution.py:

```
"""
Hand-Wrapping Exercise
----------------------

This exercise provides practice at writing a C extension module
"by hand."   That is, you will write all the interface code yourself
instead of using a tool such as SWIG.

fancy_science.h and fancy_science.c are the header and implementation file
for a *very* simple library that contains three functions::

    /* Calculate the hypotenuse of a triangle using the Pythagorean theorem */
    double my_hypot(double a, double b);

    /* Calculate Euler's number, e, using n series elements. */
    double eulers_number(int n);

    /* Einstein's e=mc^2 calculation */
    float energy(float mass)

In this exercise, you will wrap these functions and expose them to
Python through an extension module.  The skeleton for this module,
:file:`func_module.c` already exists, and the first function, :func:`my_hypot`,
has already been wrapped for you.

There is a :file:`setup.py` file that is the Python equivalent of a "make" file.
To build the extension on Windows using the mingw compiler and so
that the resulting module is put in this directory instead of a separate
build directory, do the following::

    c:\path\exercise\> setup.py build_ext --compiler=mingw32 --inplace

The :file:`build.bat` file has this command in it already for convenience, so
you can also build the module by typing::

    c:\path\exercise\> build.bat

To test the module, import it from the Python command interpreter and
call its functions.  The following will work "out of the box."::

    c:\path\exercise\> python
    >>> import fancy_science
    >>> fancy_science.my_hypot(3.0, 4.0)
    5.0
```

```
Help
~~~~

The demo directory has a more detailed example of the :func:`my_hypot` function
that may be useful.  See the :file:`readme.txt` in that directory to see how
to build the examples.

The python documentation on building extensions is here:

    http://docs.python.org/ext/ext.html


The format strings for parsing tuples is here:

    http://www.python.org/doc/1.5.2p2/ext/parseTuple.html

For more information about writing setup.py files, look here:

    http://docs.python.org/inst/inst.html


See :ref:`setup-handwrap-solution`
"""

from distutils.core import Extension, setup

ext = Extension(name="fancy_science",
                sources=['fancy_science.c', 'fancy_science_solution.c'],
                include_dirs=['.'])

if __name__ == '__main__':
    setup(name='fancy_science',
          version='1.0',
          ext_modules=[ext])
```

2.6.2 Blitz Inline Comparison - Solution

blitz_inline_compare_solution.py:

```
"""
This example takes the numpy expression used in the filter_image
exercise and compares the speed of numpy to that of weave.blitz.

Read in the lena image and use an averging filter
to "smooth" the image.  Use a "5 point stencil" where
you average the current pixel with its neighboring pixels::

               0 0 0 0 0 0 0
               0 0 0 x 0 0 0
               0 0 x x x 0 0
               0 0 0 x 0 0 0
               0 0 0 0 0 0 0

Once you have a numpy expression that works correctly, time it
using time.time (or time.clock on windows).
```

```
Use scipy.weave.blitz to run the same expression.  Again time it.

Compare the speeds of the two function and calculate the speed-up
(numpy_time/weave_time).

Plot two images that result from the two approaches and compare them.
"""

from numpy import empty, float64
from scipy import lena
from scipy import weave
from pylab import subplot, imshow, title, show, gray, figure

img = lena()

expr = """avg_img =(  img[1:-1 ,1:-1]  # center
                    + img[ :-2 ,1:-1]  # left
                    + img[2:   ,1:-1]  # right
                    + img[1:-1 , :-2]  # top
                    + img[1:-1 ,2:  ]  # bottom
                    ) / 5.0"""

import time
t1 = time.clock()
for i in range(10):
    exec expr
t2 = time.clock()
numpy_time = t2-t1
numpy_avg_img = avg_img

avg_img = empty((img.shape[0]-2, img.shape[1]-2), dtype=float64)

# Run it once so that it gets compiled.
weave.blitz(expr)

import time
t1 = time.clock()
for i in range(10):
    weave.blitz(expr)
t2 = time.clock()
blitz_time = t2-t1
blitz_avg_img = avg_img

avg_img = empty((img.shape[0]-2, img.shape[1]-2), dtype=float64)
code = """for (int i=0; i<Navg_img[0];i++)
          {
            for (int j=0; j<Navg_img[1];j++)
            {
              AVG_IMG2(i,j) =(  IMG2(i+1 ,j+1)  // center
                              + IMG2(i ,   j+1)  // left
                              + IMG2(i+2, j+1)  // right
                              + IMG2(i+1 ,j  )  // top
                              + IMG2(i+1 ,j+2)  // bottom
                              ) / 5.0;
            }
          }
       """
```

```
weave.inline(code,['avg_img', 'img'],compiler='gcc')
t1 = time.clock()
for i in range(10):
    weave.inline(code,['avg_img', 'img'],compiler='gcc')
t2 = time.clock()
inline_time = t2-t1

inline_avg_img = avg_img

print 'numpy, blitz, speed-up: %4.3f, %4.3f, %4.3f' % \
      (numpy_time, blitz_time, numpy_time/blitz_time)
print 'numpy, inline, speed-up: %4.3f, %4.3f, %4.3f' % \
      (numpy_time, inline_time, numpy_time/inline_time)
# Set colormap so that images are plotted in gray scale.
gray()

# Plot the original image first
figure()
subplot(2,3,1)
imshow(numpy_avg_img)
title('numpy filtered image')

# Now the filtered image.
subplot(2,3,2)
imshow(blitz_avg_img)
title('blitz filtered image')

# Now the filtered image.
subplot(2,3,3)
imshow(inline_avg_img)
title('inline filtered image')

# And finally the difference between the two.
subplot(2,3,5)
imshow(numpy_avg_img - blitz_avg_img)
title('blitz difference')

# And finally the difference between the two.
subplot(2,3,6)
imshow(numpy_avg_img - inline_avg_img)
title('inline difference')

show()
```

2.6.3 Weave Comparison - Solution

weave_compare_solution.py:

```
"""
Compare the speed of NumPy, weave.blitz and weave.inline for the following
equation::

        result=a+b*(c-d)

Set up all arrays so that they are one-dimensional and have 1 million
elements in them.
"""
```

```
import time

from numpy import arange, empty, float64
from scipy import weave

N = 1000000

a = arange(N,dtype=float64)
b = arange(N,dtype=float64)
c = arange(N,dtype=float64)
d = arange(N,dtype=float64)

result = empty(N,dtype=float64)

t1= time.clock()
result = a + b*(c-d)
t2=time.clock()
numpy_time = t2-t1

t1= time.clock()
weave.blitz("result = a+b*(c-d)")
t2=time.clock()
blitz_time = t2-t1
print "array size, numpy (sec), blitz (sec), numpy/blitz: %d, %3.2f, %3.2f, %3.2f" % \
      (N, numpy_time, blitz_time, numpy_time/blitz_time)


code = """
       for(int i=0;i<Na[0];i++)
       {
           result[i] = a[i]+b[i]*(c[i]-d[i]);
       }
       """


t1= time.clock()
weave.inline(code,['a','b','c','d','result'], compiler='gcc')
t2=time.clock()
inline_time = t2-t1
print "array size, numpy (sec), blitz (sec), inline (sec): %d, %3.2f, %3.2f %3.2f" % \
      (N, numpy_time, blitz_time, inline_time)

print 'Speedup for blitz and inline: %3.2f %3.2f' % \
      (numpy_time/blitz_time, numpy_time/inline_time)
```

2.6.4 Mandelbrot with Weave - Solution

mandelbrot_weave_solution.py:

```
"""
Mandelbrot with Weave
---------------------

The code in this script generates a plot of the Mandelbrot set.

1) Use weave to speed up this code
2) Compare the speed of the weave code with the vectorized-only version.
```

```
Hints:  * The mandelbrot_escape function should be placed in support_code and
             weave.inline should be used to inline a 2-d loop over the data passed in
        * The output data should be pre-allocated and passed in as another array.
        * Recall that weave creates C++ extensions so that std::complex<double> can
             be used.
"""

import time

from numpy import r_, empty, intc, vectorize, newaxis
from pylab import imshow, show, cm, figure
from scipy import weave

support_code = """

static int mandelbrot_escape(std::complex<double> c) {
    std::complex<double> z;
    int i;
    z = c;
    for (i=0; i<1000; i++) {
        z = z*z + c;
        if (std::abs(z) >= 2) break;
    }
    if (i==1000) i=-1;
    return i;
}
"""

code = """
int i,j;
std::complex<double> z;
for (i=0; i<Ny[0]; i++) {
    for (j=0; j<Nx[0]; j++) {
        z = std::complex<double>(X1(j), Y1(i));
        D2(i,j) = mandelbrot_escape(z);
    }
}
"""

# set up the grid of values
x = r_[-2:1:500j]
y = r_[-1.5:1.5:500j]

# compute the escape time values
start = time.time()
d = empty((x.size, y.size), intc)
weave.inline(code, ['x', 'y', 'd'], support_code = support_code)
print "Weave: ", time.time() - start

# plot the Mandelbrot set
figure(1)
imshow(d, cmap=cm.gist_stern)
show()

# Vectorize solution

def mandelbrot_escape(c):
    """ Compute escape time for points near the Mandelbrot set.
    This computes the number of iterations required for the sequence

    .. math::

        z_{n+1} = z_n^2 + c

    to repeat before |z_n| > 2 (which implies that the sequence diverges to
    infinity).  If we iterate 1000 times without the absolute value getting
    large, then we consider the point to be in the Mandelbrot set, and return
    the value -1.

    Parameters
    ----------

    c : complex
        The point on the complex plane being tested.

    Returns
    -------

    i : int
        The escape time for the point, or -1 if the point never escaped.
    """
    z = c
    for i in range(1000):
        z = z**2 + c
        if abs(z) >= 2:
            break
    else:
        i = -1
    return i

# vectorize the function so we can apply it to a complex array
vmand = vectorize(mandelbrot_escape)

start = time.time()
z = x + y[:,newaxis]*1j
d2 = vmand(z)
print "Vectorize: ", time.time() - start

figure(2)
imshow(d2, cmap=cm.gist_stern)

show()
```

2.6.5 Mandelbrot Cython - Solution 1

setup_solution.py:

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext

import numpy

ext = Extension("mandel", ["mandelbrot_solution.pyx"],
    include_dirs = [numpy.get_include()])

setup(ext_modules=[ext],
      cmdclass = {'build_ext': build_ext})
```

2.6.6 Mandelbrot Cython - Solution 2

setup_solution2.py:

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext

import numpy

ext = Extension("mandel", ["mandelbrot_solution2.pyx"],
    include_dirs = [numpy.get_include()])

setup(ext_modules=[ext],
      cmdclass = {'build_ext': build_ext})
```

2.6.7 Mandelbrot Cython - Solution 3

setup_solution3.py:

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext

import numpy

ext = Extension("mandel", ["mandelbrot_solution3.pyx"],
    include_dirs = [numpy.get_include()])

setup(ext_modules=[ext],
      cmdclass = {'build_ext': build_ext})
```

2.7 Ipython - Solutions

2.7.1 Mandelbrot Parallelized - Solution

mandelbrot_parallel_solution.py:

```
"""
Mandelbrot Parallelized
-----------------------

The code in this script generates a plot of the Mandelbrot set.

Use ipython's multi-client engine to parallelize this,
scattering over the y-axis to break computation into chunks
and then gathering the results before plotting.
```

```
Bonus
~~~~~

How fast is the parallel version compared to the regular version on your
system?

"""

import time

from numpy import linspace, ogrid, c_
from pylab import imshow, show, cm
from IPython.kernel import client

mec = client.MultiEngineClient()

code = '''from numpy import vectorize, r_

def mandelbrot_escape(c):
    """ Compute escape time for points near the Mandelbrot set.

    This computes the number of iterations required for the sequence

    .. math::

        z_{n+1} = z_n^2 + c

    to repeat before |z_n| > 2 (which implies that the sequence diverges to
    infinity).  If we iterate n times without the absolute value getting
    large, then we consider the point to be in the Mandelbrot set, and return
    the value -1.

    Parameters
    ----------

    c : complex
        The point on the complex plane being tested.

    Returns
    -------

    i : int
        The escape time for the point, or -1 if the point never escaped.
    """
    z = c
    for i in range(1000):
        z = z**2 + c
        if abs(z) >= 2:
            break
    else:
        i = -1
    return i

# vectorize the function so we can apply it to a complex array
vmand = vectorize(mandelbrot_escape)

# set up the grid of values
x = r_[-2:1:500j]
```

```
z = x + y*1j

# compute the escape time values
d = vmand(z)
'''

# set up the imaginary values we want to test
y = c_[-1.5:1.5:500j]

t = time.time()
# scatter the y values to the engines
mec.scatter('y', y)
# run the code on the engine
mec.execute(code)
# gather the escape times
d = mec.gather('d')
print 'Execution time:', time.time() - t

# plot the Mandelbrot set
imshow(d, cmap=cm.gist_stern)
show()
```

This document was generated on January 18, 2012.