Python for Finance

Exercises and Solutions

Enthought, Inc.

CONTENTS

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*.

This script contains a copy of the inventory exercise solution. It also contains at the top 3 empty functions that should be used to 'refactor' the original code. Refactoring some code means re-organizing it into functions (and those functions into modules) that can be reused later for similar tasks.

1. Move the code from the script part into the functions.
2. Call the functions in order to achieve the same task.

Bonus:

3. The log processing function is an example of a data processing function that can be re-used by other projects. Move it to its own file (data_process.py) and import it here to still be able to run the script.

See *Refactor Inventory - Solution 1*.

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

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

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

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

Write a __repr__ method that prints out an official representation of the person that would produce an identical object if evaluated:

```
Person('eric', 'jones')
```

Bonus:

2. Extend this class in such as way that the code about Homer in the slides on OOP works: for that create the methods eat, take_nap, speak.

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 so that the output includes the 'mass' and 'velocity' argument names. That is, the output should have the form "Particle(mass=2.3, velocity=0.1)" instead of "Particle(2.3, 0.1)".
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 thousandths 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 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 new sqlite database. Create a 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.

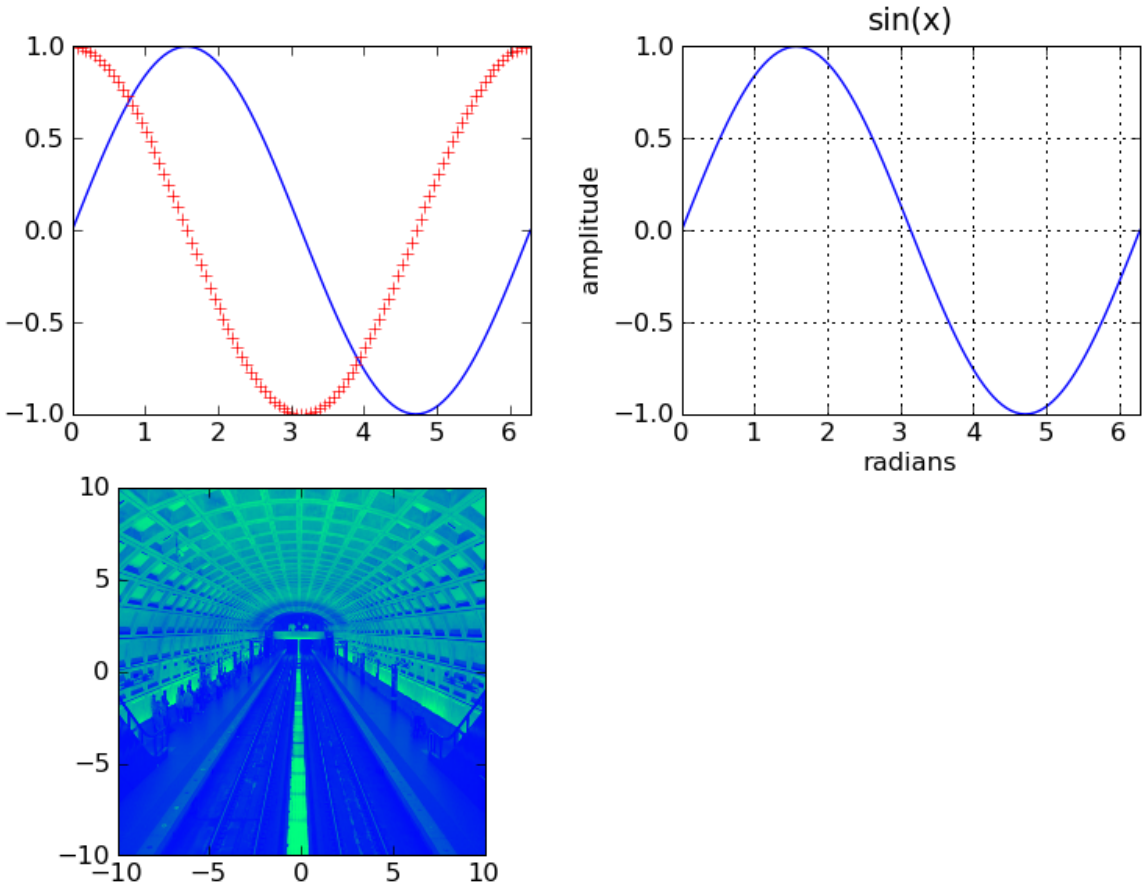
2. Select the records which have an absolute difference between open and close that is greater than 1% (of the open) and sort them in order of that spread.

See *Dow Database - Solution*.

1.3 Numpy Exercises

1.3.1 Plotting

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



'**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 matplotlib.pyplot import imread

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

img = imread('dc_metro.jpg')
```

Tip: If you find that the label of one plot overlaps another plot, try adding a call to *tight_layout()* to your script.

Bonus

4. The *subplot()* function returns an axes object, which can be assigned to the *sharex* and *sharey* keyword arguments of another subplot() function call. E.g.:

   ```
   ax1 = subplot(2,2,1)
   ...
   subplot(2,2,2, sharex=ax1, sharey=ax1)
   ```

   Make this modification to your script, and explore the consequences. Hint: try panning and zooming in the subplots.

See *Plotting - Solution 1*.

1.3.2 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*.

1.3.3 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*

1.3.4 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*.

1.3.5 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*.

1.3.6 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

1. 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.)
2. 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*.

1.3.7 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*.

1.3.8 Structured Array

In this exercise you will read columns of data into a structured array using loadtxt and combine that array to a regular array to analyze the data and learn how the pressure velocity evolves as a function of the shear velocity in sound waves in the Earth.

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*.

1.4 Scipy Exercises

1.4.1 Estimate Volatility

Background

A standard model of stock 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*.

1.4.2 Using fsolve

1. Define a function *funcv(x)* that computes the following

   f0 = x0**2 + x1**2 - 2.0 f1 = x0**2 - x1**2 - 1.0

   where x0 = x[0] and x1 = x[1], and the return value of funcv(x) is (f0, f1).
2. Find x0 and x1 so that f0==0 and f1==0. Hint: See scipy.optimize.fsolve.
3. Create images of f0 and f1 around 0 and plot the guess point and the solution point.

See *Using fsolve - Solution*.

1.4.3 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*.

1.4.4 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*

1.4.5 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 original exponential function using scipy.curve_fit.

See: *Data Fitting - Solution*.

1.4.6 Integrate a Function

Integrate sin using scipy.integrate.quad.

Topics: SciPy's integration library, vectorization.

1. Use scipy.integrate.quad to integrate sin from 0 to pi/4. Print out the result. Hint:

   ```
   from scipy import integrate
   >>> integrate.quad?
   ```

2. Integrate sin from 0 to x where x is a range of values from 0 to 2*pi. Compare this to the exact solution, -cos(x) + cos(0), on a plot. Also plot the error between the two.

   Hint: Use vectorize so that integrate.quad works with arrays as inputs and produces arrays as outputs.

See: *Integrate a Function - Solution*.

1.4.7 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 Advanced Topics Exercises

1.5.1 Pandas Moving Average

In this exercise, you will use Pandas to compute and plot the moving average of a time series. The file 'data.txt' contains measurements from a lake, recorded every half hour. There are missing data points scattered throughout the data set.

Note. The index in the DataFrame created below contains timestamps, but no special time series features will be used, so it is not necessary to have covered Pandas time series before doing this exercise.

0. Use pandas.read_table() to read the data from the file 'data.txt' into a Pandas DataFrame. The columns in the DataFrame are 'temp' (temperature), 'sal' (salinity), 'ph' and 'depth'. Plot the depth. (Part 0 is done for you below.)
1. Use pandas.rolling_mean() to compute and plot the moving average of the temperature using a window of 12 hours. (Note that the sample period of the data is a half hour, so 12 hours corresponds to a window of 24 samples.) Then do the same for windows of 24 and 48 hours.

   Hint: Use the *min_periods* argument to specify that at least 12 samples are needed in a window. What happens if you do not specify *min_periods*?
2. Add a plot of the exponentially weighted moving average to the plot created in part 1. The function to compute this is pandas.ewma(). If x is the input series, the exponentially weighted moving average is the time series y where:

```
y[0] = x[0]
y[k] = alpha * x[k] + (1 - alpha) * y[k - 1]  for k > 0.
```

   So the output at time k is a weighted combination of the input at time k and the output at time k - 1.

   The parameter *alpha* is not given directly. Instead, *span* is given, where:

```
alpha = 2 / (span + 1)
```

   Experiment with different values of *span*. (You can also use the keyword argument *com*, where *alpha = 1 / (com + 1)*.)

See *Pandas Moving Average - Solution*.

1.5.2 Pandas DataFrame

This is a basic fluency exercise. Each part can be solved with just one line of code.

Topics: Pandas DataFrame, pivot, pivot_table, groupby

1. In the code below, read_csv() reads the file "sales.csv" into the DataFrame df1. It looks like this:

```
    Quarter    Name  Product  Service
0         1    Kahn    345.7     90.0
1         1  Porter    291.0      0.0
2         1     Lin    406.5    131.0
3         1   Mason    222.0     14.0
4         2    Kahn    295.0     65.5
5         2  Porter    131.0     19.1
6         2     Lin    319.5     12.0
7         2   Mason    263.1     45.0
8         3    Kahn    195.0      6.7
9         3  Porter    155.9     33.9
10        3     Lin    126.5     89.0
11        3   Mason    140.0    101.5
12        4    Kahn    445.1      8.2
13        4  Porter    308.3     90.0
14        4     Lin    410.0     14.0
15        4   Mason    251.6     63.0
```

   Use the pivot() method to create a new DataFrame whose index is 'Name', whose columns are the Quarters, and whose values are from the 'Product' column. It should look like this:

```
Quarter      1      2      3      4
Name
Kahn     345.7  295.0  195.0  445.1
Lin      406.5  319.5  126.5  410.0
Mason    222.0  263.1  140.0  251.6
Porter   291.0  131.0  155.9  308.3
```

2. Use the pivot_table() method of df1 to generate a DataFrame that looks like this:

```
         Product                         Service
Quarter        1      2      3      4        1     2      3     4
Name
Kahn       345.7  295.0  195.0  445.1       90  65.5    6.7   8.2
Lin        406.5  319.5  126.5  410.0      131  12.0   89.0  14.0
Mason      222.0  263.1  140.0  251.6       14  45.0  101.5  63.0
Porter     291.0  131.0  155.9  308.3        0  19.1   33.9  90.0
```

   Note that no aggregation is performed in that table. Also note that the column index is hierarchical: it has the form (s, q), where s is either 'Product' or 'Service' and q is the Quarter.

3. This part requires the use of the *aggfunc* argument of pivot_table(). From df1 of part 1, use the pivot_table() method to create a DataFrame that looks like this:

```
         Product  Service
Quarter
1         1265.2    235.0
2         1008.6    141.6
3          617.4    231.1
4         1415.0    175.2
```

   The table shows the quarterly sales totals for the 'Product' and 'Service' categories.

4. A slight variation of part 3: from df1 of part 1, use the pivot_table() method to create a DataFrame that looks like this:

```
         Product  Service
Name
Kahn      1280.8    170.4
Lin       1262.5    246.0
Mason      876.7    223.5
Porter     886.2    143.0
```

   It shows the total product and service sales for each person.

5. Modify your answer to part 4 to include the column sums in the result. Hint: look at the 'margins' argument to pivot_table().

6. Add a 'Total' column to result of part 5, so that it looks like:

```
         Product  Service   Total
Name
Kahn      1280.8    170.4  1451.2
Lin       1262.5    246.0  1508.5
Mason      876.7    223.5  1100.2
Porter     886.2    143.0  1029.2
All       4306.2    782.9  5089.1
```

7. Reproduce the answer of part 3 using the groupby() method.

8. Reproduce the answer of part 4 using the groupby() method.

   Your first attempt might produce this DataFrame:

```
         Quarter  Product  Service
Name
Kahn          10   1280.8    170.4
Lin           10   1262.5    246.0
Mason         10    876.7    223.5
Porter        10    886.2    143.0
```

   The 'Quarter' column is not useful; a slight modification can be used to keep just the 'Product' and 'Service' columns.

See *Pandas DataFrame - Solution*.

1.5.3 Wind Statistics

This exercise is an alternative version of the Numpy exercise but this time we will be using pandas for all tasks. The data have been modified to contain some missing values identified by -0.00 or NaN depending on whether the measurement couldn't be trusted or whether no data was collected. Using pandas should make this exercise easier, in particular for the bonus question.

Of course, you should be able to perform all of these operations without using a for loop or other looping construct.

Topics: Pandas, time-series

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

```
Yr Mo Dy   RPT   VAL   ROS   KIL   SHA   BIR   DUB   CLA   MUL   CLO   BEL   MAL
61  1  1 15.04 14.96 13.17  9.29 -0.00  9.87 13.67 10.25 10.83 12.58 18.50 15.04
61  1  2 14.71   NaN 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   NaN  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 'read_table' function from pandas to read the data into a DataFrame. Make sure to convert all missing values (NaN and -0.00 to np.nan).

2. Replace the first 3 columns by a proper datetime index, created manually.
3. Compute how many values are missing for each location over the entire record. They should be ignored in all calculations below. Compute how many non-missing values there are in total.
4. Calculate the mean windspeeds of the windspeeds over all the locations and all the times (a single number for the entire dataset).
5. 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)
6. 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)
7. Find the average windspeed in January for each location. Treat January 1961 and January 1962 both as January.
8. Downsample the record to a yearly, monthly frequency and weekly frequency for each location.
9. Plot linearly and with a candle plot the monthly data for each location.

(Used to be a bonus)

10. Calculate the mean windspeed for each month in the dataset. Treat January 1961 and January 1962 as *different* months.
11. 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.

Notes

This solution has been tested with Pandas version 0.7.3.

The original data from which these were derived 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*.

1.5.4 Modeling Climate Data in Pandas

Topics: Pandas, HDF5, FTP file retrieval.

Introduction

The National Climatic Data Center (NCDC) provides FTP access to extensive historic and current climate data, collected from weather stations across the globe. In this extended exercise, we will download 'Global Surface Summary of the Day' (GSOD) data from the NCDC servers and use Pandas to analyse and plot selected data.

The data that we will analyse are organized by year, and stored under the URL:

> ftp://ftp.ncdc.noaa.gov/pub/data/gsod

The file

ftp://ftp.ncdc.noaa.gov/pub/data/gsod/readme.txt

describes the data format; a copy of this file is stored in the exercise folder under the name 'gsod_readme.txt'.

1. Retrieve GSOD datafiles for 2008 for the following 3 cities.

   Austin, US Cambridge, UK Paris, FR

   Store the files locally. The full URLs for the 3 data files are:

   Austin: ftp://ftp.ncdc.noaa.gov/pub/data/gsod/2008/999999-23907-2008.op.gz Cambridge: ftp://ftp.ncdc.noaa.gov/pub/data/gsod/2008/035715-99999-2008.op.gz Paris: ftp://ftp.ncdc.noaa.gov/pub/data/gsod/2008/071560-99999-2008.op.gz

   Hint: Use the urlretrieve function from the standard library urllib module for this. You can also use ftplib, if you prefer.

   (Note: in case of difficulties accessing the server, the relevant files are also stored in the 'data' subfolder of the exercise folder.)

2. For each of the three cities above:

   (a) Read the datafile into a Pandas DataFrame object with suitable column headings.

   (b) Replace missing values (as described in the readme file) for the numeric columns with NaNs.

   (c) Use the Year, Month and Day information in the input to construct and assign a suitable index for the DataFrame.

   (d) (Optional) Replace the 'FRSHTT' indicators column with six separate boolean columns giving fog, rain, snow, hail, thunder and tornado indicators.

   Hints: pandas.read_fwf reads data in fixed-format columns. You may also find the gzip standard library module useful for dealing with the compressed .gz files.

   Bonus: Extract field heading, column heading and missing data information programmatically from the gsod_readme.txt file, rather than copying and pasting.

3. The Austin data for the start of the year are missing. Retrieve alternative Austin data for 2008 from the URL:

   Austin (2): ftp://ftp.ncdc.noaa.gov/pub/data/gsod/2008/722544-13958-2008.op.gz

   Now:

   (a) Create a scatter plot comparing the two sets of Austin data. (You'll first need to adjust the two datasets to a common set of indices.)

   (b) Compute a correlation coefficient for the two Austin datasets.

   (c) Finally, fill in the missing days in the original Austin dataset using the data from the second Austin dataset.

4. Create a single pandas Panel containing all three datasets.

5. From the panel you created in part 4, extract a DataFrame giving just the mean temperatures for each location. Use this to plot all three temperatures on the same graph.

6. Compute and report mean temperatures for the whole year for each location.

7. Compute and plot rolling means with a 14-day window for each of the locations.

8. Compute monthly total rainfalls for each location, dropping any locations for which no rainfall data are available. Create a bar plot showing these totals.

BONUS:

1. Write a 'CachingClimateDataStore' class that allows easy retrieval of the GSOD data, and stores the corresponding pandas dataframe locally in an HDF5 file called 'gsod_climate_data.h5'.

   For example, your class might be used something like this:

   ```
   >>> store = CachingClimateDataStore()
   >>> # Next line goes to the FTP server, fetches the file, turns
   >>> # it into a dataframe and stores it locally in the HDF5 file.
   >>> paris_climate = store['071560-99999-2008']
   >>> # A second retrieval operation retrieves data directly from the HDF5
   >>> # file.
   >>> paris_climate = store['071560-99999-2008']
   ```

   (Hints: (1) look at the pandas.HDFStore class. (2) you can override item access via the __getitem__ special method.)

2. (Half-day to full-day team project, open-ended!) Use Traits and Chaco (or a UI toolkit of your choice) to write an application that allows the user to:

   - Retrieve data for cities on request, and store the data locally in an HDF5 file.
   - Display plots showing a summary of the data for any particular year and location.
   - Compare two locations (or two years for the same location), showing scatter plots, combined graphs, and correlation coefficients.

See *Modeling Climate Data in Pandas - Solution*.

1.6 Interface With Other Languages Exercises

1.6.1 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*.

1.6.2 Cythonize rankdata

The function rankdata(a) is defined below. (This is a slightly modified version of the function scipy.stats.rankdata.) Given an array *a*, this function returns an array containing the "ranks" of the elements of *a*. If there are no repeated values in *a*, this is simply the numerical order of the elements when sorted, starting with 1. For example:

```
>>> rankdata([25, 15, 10, 20])
array([ 4.,  2.,  1.,  3.]
```

If there are repeated elements (i.e. "ties"), then all the repeated elements are assigned the average rank of the group. For example, in the following, the value 10 occurs at rank 1 and 2, so both are assigned the rank 1.5. Similarly, the value 25 occurs at ranks 4, 5 and 6, so all are assigned the rank 5.0:

```
>>> rankdata([25, 15, 10, 25, 25, 30, 10])
array([ 5. ,  3. ,  1.5,  5. ,  5. ,  7. ,  1.5])
```

The implementation involves a loop over the array *a*. Such loops are relatively slow in Python, so this is a good candidate for speeding up the function by implementing it in Cython. That is the assignment for this exercise.

1. Copy this file to cython_rankdata.pyx. The file setup.py is already configured to build this extension module, so you can run:

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

   If there are no problems in cython_rankdata.pyx, this will create the extension module, which you can import in python. With no changes to the python at all, you may still see a speed improvement in the Cython version. For example:

   ```
   In [2]: from rankdata import rankdata  # Python version

   In [3]: from cython_rankdata import rankdata as cyrankdata  # Cython version

   In [4]: x = np.random.randint(80, size=150)  # Create some test data

   In [5]: %timeit rankdata(x)
   1000 loops, best of 3: 373 us per loop

   In [6]: %timeit cyrankdata(x)
   1000 loops, best of 3: 300 us per loop
   ```

2. Start modifying cython_rankdata.pyx to improve its performance. Here's a first step: add the lines:

   ```
   cimport numpy as np
   import cython
   ```

   after the normal python import of numpy. Then start adding appropriate type declarations to the function rankdata that let Cython generate efficient C code. You shouldn't have to change the algorithm–most of the crucial changes will be type declarations!

3. As you modify cython_rankdata.pyx, run cython on it with the "-a" option, and inspect the HTML file. Ideally, the for-loop should contain *no* yellow lines.

Here's the peformance comparison of the original python version and the Cython solution in cython_rankdata_solution.pyx:

```
In [1]: from rankdata import rankdata

In [2]: from cython_rankdata_solution import rankdata as cyrankdata

In [3]: x = np.random.randint(80, size=150)

In [4]: %timeit rankdata(x)
1000 loops, best of 3: 397 us per loop

In [5]: %timeit cyrankdata(x)
100000 loops, best of 3: 18.4 us per loop
```

That's better than 20 times faster.

Reminders:

- The most efficient indexing of array occurs when the index variable is an unsigned integer.
- Cython provides function decorators to turn off certain safety checks. Two that are relevant here are:

```
@cython.cdivision(True)
@cython.boundscheck(False)
```

See *Cythonize rankdata - Solution*.

1.7 Statistics Exercises

1.7.1 Linear Regression of Mileage Data

The file "mileage_data_1991.txt" contains information from 1991 about several models of cars. The fields in the file are:

MAKE/MODEL: Manufacturer and model name VOL: Cubic feet of cab space HP: Engine horsepower MPG: Average miles per gallon SP: Top speed (mph) WT: Vehicle weight (100 lb)

1. Read the data into an array. (This is already done.)
2. Use scipy.stats.linregress to compute the linear regression line for the weight (WT) and mileage (MPG) of the cars. Print the correlation coefficient.
3. Repeat 2 for the weight and "gallons-per-mile" for the cars.
4. (Optional) For 2 and 3, plot the data with the weight on the x-axis, and plot the regression line. (Use separate figures for 2 and 3.)

See *Linear Regression of Mileage Data - Solution*.

1.7.2 Chi-square test

1. A six-sided die is rolled 50 times, and the number of occurrences of each side are recorded in the following table:

Side	1	2	3	4	5	6
Occurrences	6	1	9	7	14	13

Is the die fair? That is, is each side equally likely?

2. 17 woman and 18 men are asked whether they prefer chocolate or vanilla ice cream. The following table shows the result:

```
+-------+-----------+---------+
| Group | Chocolate | Vanilla |
+-------+-----------+---------+
| Women |     12    |    5    |
+-------+-----------+---------+
| Men   |     10    |    8    |
+-------+-----------+---------+
```

Use the chi-square test to investigate the relation between an individual's sex and the individual's preference for ice cream. Then apply Fisher's exact test (scipy.stats.fisher_exact), and compare the results.

See *Chi-square test - Solution*.

1.7.3 Test for Normal Distribution

The file "rdata.txt" contains a column of numbers.

1. Load the data "rdata.txt" into an array called 'x'.
2. Apply the Anderson-Darling (scipy.stats.anderson) and Kolmogorov-Smirnov (scipy.stats.kstest) tests to 'x', using a 5 percent confidence level. What do you conclude?
3. Apply the Anderson-Darling and Kolmogorov-Smirnov tests to 'dx', where 'dx = numpy.diff(x)', using a five percent confidence level. What do you conclude?

See *Test for Normal Distribution - Solution*.

1.8 Traits Exercises

1.8.1 Trait Particle

Define a Traits-based class for a Particle.

1. Define a Particle class derived from HasTraits. A Particle should have a `mass` and a `velocity` that are both floating point values. It should also have two read-only Property traits, `energy` and `momentum`, given by:

```
energy = 0.5 * mass * velocity **2
momentum = mass * velocity
```

After defining the class, create an instance of it and set/get some its traits to test its behavior.

2. Create a UI that groups the mass and velocity side-by-side at the top of a dialog with the energy and momentum below.

See *Trait Particle - Solution*.

1.8.2 Trait Person

Define a Traits based class for a Person.

Hint: See the demo/traits_examples/configure_ui_1.py

1. Define a person class. A person should have a `first_name` and `last_name` that are both strings, an `age` that is a Float value, and a `gender` that is either 'male' or 'female'. After defining the class, create an instance of it and set/get some its traits to test its behavior. Try setting names to strings and floats and see what happens.
2. Add a static trait listener that prints the age when it changes.
3. Create a UI that groups the name at the top of a dialog with the gender and age below.

See *Trait Person - Solution*.

1.8.3 Trait Function

Traits-based class for calculating a function.

1. Write a Traits-based class that calculates the following signal:

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

   Make *a*, *b*, and *c* floating point traits with default values of 2.0, 0.76, 1.0. The *x* variable should be an array trait with the default value [0, 0.5, 1, 1.5, ..., 5.0]. The *y* value should be a property trait that updates any time *a*, *b*, *c* or *x* changes.

   Once your class is defined, create an instance of the object and print its *y* value.
2. Define a function *printer(value)* that prints an array, and set it to be an external trait listener for the trait *y* on the instance created in 1, so *y*'s value it printed whenever it changes.
3. Create a UI that displays *a*, *b*, and *c* as text boxes.
4. Create a UI that displays *a*, *b*, and *c* as sliders, and *x* and *y* as read-only arrays. Use an ArrayEditor for the display of the arrays, and set the 'width' keyword of the ArrayEditor to a value large enough for the values in the array.

See: *Trait Function - Solution*.

1.9 Chaco Exercises

1.9.1 Function Plot

The module *damped_osc.py* defines DampedOsc, a class that implements the function:

```
y = a * exp(-b * x) * cos(omega * x + phase).
```

It has four Float traits (*a*, *b*, *omega* and *phase*), an Array trait (*x*), and a Property trait (*y*) that is an array computed by the class.

The goal of this exercise is to create a view of a DampedOsc instance that plots the function in a Chaco plot and allows the user to change the parameters *a*, *b*, *omega* and *phase* using sliders.

The code is started for you. This file may be run (from within ipython) and a plot will be created with a slider below it for the parameter *a*. However, the plot will not respond correctly to changes in *a*. The following steps will lead to a working function plotter.

1. The class DampedOscView is started for you below, but it does not work correctly. If you change *a* with the slider, the limits of the vertical axis change (this is done in the method _a_changed()), but the graph of the function does not reflect the new parameter. The problem is that the value of *y*, stored in the plot's ArrayPlotData instance, does not change when *a* changes.

To fix this, add a static listener method, _y_changed(...), that updates the values of *y* in the plot's ArrayPlotData instance whenever the model's value of *y* changes. You can get the plot's ArrayPlotData as self.plot.data, and you can use the ArrayPlotData method set_data(...) to update the value of *y*.

Once you have made this change to the code, when the *a* slider is changed, you will see the vertical axis labels change, but the graph will appear unchanged, because now the change in the vertical axis matches the scale of the graph.

2. DampedOscView does not expose all the parameters of the model. Add a DelegatesTo trait for *b*. Then in the View(), add an Item for *b*. Use a RangeEditor (as in the Item for *a*), but use a range of 0 <= *b* <= 2.

   If you run the script after making the change, you should see that changing *b* with the slider automatically updates the graph. (Why?)

3. Complete the function plotter by repeating the previous step for the parameters *omega* and *phase*. Use the ranges:

```
0 <= omega <= 10
-pi <= phase <= pi
```

See *Function Plot - Solution 1*.

Note

Take a look at *function_plot_solution2.py*, *function_plot_solution3.py* and *function_plot_solution4.py*. These solutions take advantage of different Traits UI features to implement the function plotter.

1.9.2 Plot adjuster

In this problem you will create a Figure containing a combination of a line plot and a scatter plot, both using the same data.

In addition, the plot should be exposed together with Editors that allow you to:

1. select from a sequence of functions to be plotted
2. change the color of the scatter-plot markers
3. change the background color of the plot (ColorTrait)
4. change the marker size (using a RangeEditor)
5. change the marker type (marker_trait)

Bonus

1. add User-Interface elements to allow changing the x-range and the number of data-points

See: *Plot adjuster - Solution*.

1.9.3 Custom tool

In this exercise you will add custom interactivity to a Chaco plot by creating a custom tool that allows a user to add new points by clicking.

1. Create a Chaco Plot that plots sin(x) * x**3 from -14 to 14
2. **Create a Custom Tool that responds to mouse clicks and**

(a) places a point on the plot at each clicked location and

(b) prints the location of the click to standard-out.

Hint: Add the clicked value to the data that is plotted.

Bonus

Add the ability to remove the point nearest the location that is clicked.

See: *Custom tool - 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:

```
"""
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 Refactor Inventory - Solution 1

data_process_solution.py:

```
""" Data processing module containing functions that convert string data into
more manageable list of transactions.
"""

def process_log(log):
    """ Process a warehouse log"""
    # Remove any leading and trailing whitespace from the string::

    warehouse_log = 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)

    return transactions
```

2.1.7 Financial Module - Solution

financial_module_solution.py:

```
"""
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.

"""


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)
```

2.1.8 ASCII Log File - Solution 1

ascii_log_file_solution.py:

```
"""
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.

"""

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]
```

2.1.9 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).
"""

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]
```

2.1.10 Person Class - Solution

person_class_solution.py:

```
"""
Person Class
------------

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

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

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

Write a __repr__ method that prints out an official representation
of the person that would produce an identical object if evaluated::
```

```
    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(%r, %r)' % (self.__class__.__name__, self.first, self.last)

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

2.1.11 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')
```

2.1.12 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__() method so that the output includes
   the 'mass' and 'velocity' argument names.  That is, the
   output should have the form "Particle(mass=2.3, velocity=0.1)"
   instead of "Particle(2.3, 0.1)".

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.
        """
        return "Particle(mass={0!r}, velocity={1!r})".format(
            self.mass, self.velocity)

    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()
```

2.1.13 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 thousandths 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
```

2.1.14 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 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 Numpy - Solutions

2.3.1 Plotting - Solution 1

plotting_bonus_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 matplotlib.pyplot import imread

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

    img = imread('dc_metro.jpg')

Tip: If you find that the label of one plot overlaps another plot, try adding
a call to `tight_layout()` to your script.

Bonus
~~~~~

4. The `subplot()` function returns an axes object, which can be assigned to
   the `sharex` and `sharey` keyword arguments of another subplot() function
   call.  E.g.::

       ax1 = subplot(2,2,1)
       ...
       subplot(2,2,2, sharex=ax1, sharey=ax1)

   Make this modification to your script, and explore the consequences.
   Hint: try panning and zooming in the subplots.

"""


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

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

img = imread('dc_metro.JPG')

close('all')
# 2x2 layout, first plot: sin and cos
ax1 = subplot(2, 2, 1)
plot(x, s, 'b-', x, c, 'r+')
axis('tight')

# 2nd plot: gridlines, labels
subplot(2, 2, 2, sharex=ax1, sharey=ax1)
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, origin='lower')
```

```
tight_layout()

show()


savefig('my_plots.png')
```

2.3.2 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 matplotlib.pyplot 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()
```

2.3.3 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
```

2.3.4 Filter Image - Solution

filter_image_solution.py:

```
"""
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.

"""

from scipy.misc.pilutil import imread
from matplotlib.pyplot 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()
```

2.3.5 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 matplotlib.pyplot 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()
```

2.3.6 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
~~~~~

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

2. 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.

"""

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 1."
print "  mean:", monthly_means
print

# Bonus 2.
# 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 'Bonus 2. 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.
```

2.3.7 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 matplotlib.pyplot 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()
```

2.3.8 Structured Array - Solution

structured_array_solution.py:

```
"""
Structured Array
----------------

In this exercise you will read columns of data into a structured array using
loadtxt and combine that array to a regular array to analyze the data and learn
how the pressure velocity evolves as a function of the shear velocity in sound
waves in the Earth.

    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 matplotlib.pyplot 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()
```

2.4 Scipy - Solutions

2.4.1 Estimate Volatility - Solution

estimate_volatility_solution.py:

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

A standard model of stock 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 matplotlib.pyplot 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()
```

2.4.2 Using fsolve - Solution

solve_function_solution.py:

```
"""
Using fsolve
------------

1. Define a function `funcv(x)` that computes the following

       f0 = x0**2 + x1**2 - 2.0
       f1 = x0**2 - x1**2 - 1.0

    where x0 = x[0] and x1 = x[1], and the return value of
    funcv(x) is (f0, f1).

2. Find x0 and x1 so that f0==0 and f1==0.
   Hint: See scipy.optimize.fsolve.

3. Create images of f0 and f1 around 0 and
   plot the guess point and the solution point.
```

```
"""

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

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

# 1. Define the function.

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()
```

2.4.3 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)
```

2.4.4 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)
```

2.4.5 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 matplotlib.pyplot 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()
```

2.4.6 Integrate a Function - Solution

integrate_function_solution.py:

```
"""
Integrate sin using scipy.integrate.quad.

Topics: SciPy's integration library, vectorization.

1. Use scipy.integrate.quad to integrate sin from 0 to pi/4.
   Print out the result. Hint::

       from scipy import integrate
       >>> integrate.quad?

2. Integrate sin from 0 to x where x is a range of values from 0 to 2*pi.
   Compare this to the exact solution, -cos(x) + cos(0), on a plot.
   Also plot the error between the two.

   Hint: Use vectorize so that integrate.quad works with arrays as inputs and
   produces arrays as outputs.
"""
from numpy import linspace, vectorize, sin, cos, pi
from scipy import integrate
from matplotlib.pyplot import plot, legend, show, subplot, xlabel, ylabel, \
                              title

# A. integrate sin from 0->pi/2
result, error = integrate.quad(sin,0, pi/2)
print "integral(sin 0->pi/2):", result

# B. Integrate sin from 0 to x where x is a range of
#    values from 0, 2*pi

x = linspace(0, 2*pi, 101)

# 1. quad needs to be vectorized before you can call it with an array.
vquad = vectorize(integrate.quad)

# 2. Now calculate the integral using the vectorized function.
approx, error_est = vquad(sin, 0, x)

# 3. Evaluate the actual integral value for x.
exact = -cos(x) + cos(0)

# 4. Plot the comparison.
subplot(1,2,1)
plot(x, approx, label="Approx")
```

```
plot(x, exact, label="Exact")
xlabel('x')
ylabel('integral(sin)')
title('Integral of sin from 0 to x')
legend()

subplot(1,2,2)
plot(x, exact-approx)
title('Error in approximation')
show()
```

2.4.7 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 matplotlib.pyplot 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 Advanced Topics - Solutions

2.5.1 Pandas Moving Average - Solution

pandas_moving_average_solution.py:

```
"""
Pandas Moving Average
---------------------

In this exercise, you will use Pandas to compute and plot the moving
average of a time series.  The file 'data.txt' contains measurements
from a lake, recorded every half hour.  There are missing data points
scattered throughout the data set.

*Note.*  The index in the DataFrame created below contains timestamps,
but no special time series features will be used, so it is not necessary
to have covered Pandas time series before doing this exercise.

0. Use pandas.read_table() to read the data from the file 'data.txt'
   into a Pandas DataFrame.  The columns in the DataFrame are 'temp'
   (temperature), 'sal' (salinity), 'ph' and 'depth'.  Plot the depth.
   (Part 0 is done for you below.)

1. Use pandas.rolling_mean() to compute and plot the moving average of the
   temperature using a window of 12 hours.  (Note that the sample period of the
   data is a half hour, so 12 hours corresponds to a window of 24 samples.)
   Then do the same for windows of 24 and 48 hours.

   *Hint:* Use the `min_periods` argument to specify that at least 12
   samples are needed in a window.  What happens if you do not
   specify `min_periods`?

2. Add a plot of the exponentially weighted moving average to the plot
   created in part 1.  The function to compute this is pandas.ewma().
   If `x` is the input series, the exponentially weighted moving
   average is the time series `y` where::

       y[0] = x[0]
       y[k] = alpha * x[k] + (1 - alpha) * y[k - 1]  for k > 0.

   So the output at time `k` is a weighted combination of the input
   at time `k` and the output at time `k` - 1.

   The parameter `alpha` is not given directly.  Instead, `span` is
   given, where::

       alpha = 2 / (span + 1)

   Experiment with different values of `span`.  (You can also use the
```

```
   keyword argument 'com', where 'alpha = 1 / (com + 1)'.)

See :ref:'pandas-moving-average-solution'.

"""

from datetime import datetime
import pandas as pd
import matplotlib.pyplot as plt


# 0.

# To parse the time field of the file, we use the following
# timestamp format:
fmt = "%Y%j+%H%M"

df = pd.read_table('data.txt', sep='\s+', skiprows=22, skip_footer=1,
                   names=['time', 'temp', 'sal', 'ph', 'depth'],
                   parse_dates=True,
                   date_parser=lambda s: datetime.strptime(s, fmt),
                   index_col=0)

temp = df['temp']

temp.plot(label='Raw data')


# 1.

# 12 hour moving average (each period is a half hour).
ma1 = pd.rolling_mean(temp, 24, min_periods=12)
ma1.plot(label='12 hour moving average')

# 24 hour moving average.
ma2 = pd.rolling_mean(temp, 48, min_periods=12)
ma2.plot(label='24 hour moving average')

# 48 hour moving average.
ma3 = pd.rolling_mean(temp, 96, min_periods=12)
ma3.plot(label='48 hour moving average')


# 2.

# Exponentially weighted moving average
ewma1 = pd.ewma(temp, span=80)
ewma1.plot(label='Exponentially weighted', style='--')

# ---
plt.legend(loc='best')
plt.ylabel('Temperature')
plt.show()
```

2.5.2 Pandas DataFrame - Solution

pandas_dataframe_solution.py:

```
"""
Pandas DataFrame
----------------

This is a basic fluency exercise.  Each part can be solved with just one line
of code.

Topics: Pandas DataFrame, pivot, pivot_table, groupby

1. In the code below, read_csv() reads the file "sales.csv" into the DataFrame
   df1.  It looks like this::

           Quarter    Name  Product  Service
      0          1    Kahn    345.7     90.0
      1          1  Porter    291.0      0.0
      2          1     Lin    406.5    131.0
      3          1   Mason    222.0     14.0
      4          2    Kahn    295.0     65.5
      5          2  Porter    131.0     19.1
      6          2     Lin    319.5     12.0
      7          2   Mason    263.1     45.0
      8          3    Kahn    195.0      6.7
      9          3  Porter    155.9     33.9
      10         3     Lin    126.5     89.0
      11         3   Mason    140.0    101.5
      12         4    Kahn    445.1      8.2
      13         4  Porter    308.3     90.0
      14         4     Lin    410.0     14.0
      15         4   Mason    251.6     63.0

   Use the pivot() method to create a new DataFrame whose index is 'Name',
   whose columns are the Quarters, and whose values are from the 'Product'
   column.  It should look like this::

      Quarter      1      2      3      4
      Name
      Kahn     345.7  295.0  195.0  445.1
      Lin      406.5  319.5  126.5  410.0
      Mason    222.0  263.1  140.0  251.6
      Porter   291.0  131.0  155.9  308.3

2. Use the pivot_table() method of df1 to generate a DataFrame that looks
   like this::

               Product                      Service
      Quarter        1      2      3      4       1     2      3     4
      Name
      Kahn       345.7  295.0  195.0  445.1      90  65.5    6.7   8.2
      Lin        406.5  319.5  126.5  410.0     131  12.0   89.0  14.0
      Mason      222.0  263.1  140.0  251.6      14  45.0  101.5  63.0
      Porter     291.0  131.0  155.9  308.3       0  19.1   33.9  90.0

   Note that no aggregation is performed in that table.  Also note that
   the column index is hierarchical: it has the form (s, q), where
   s is either 'Product' or 'Service' and q is the Quarter.

3. This part requires the use of the `aggfunc` argument of pivot_table().
   From df1 of part 1, use the pivot_table() method to create a DataFrame
```

```
   that looks like this::

                Product  Service
        Quarter
        1         1265.2    235.0
        2         1008.6    141.6
        3          617.4    231.1
        4         1415.0    175.2

   The table shows the quarterly sales totals for the 'Product' and
   'Service' categories.

4. A slight variation of part 3: from df1 of part 1, use the pivot_table()
   method to create a DataFrame that looks like this::

                Product  Service
        Name
        Kahn     1280.8    170.4
        Lin      1262.5    246.0
        Mason     876.7    223.5
        Porter    886.2    143.0

   It shows the total product and service sales for each person.

5. Modify your answer to part 4 to include the column sums in the result.
   Hint: look at the 'margins' argument to pivot_table().

6. Add a 'Total' column to result of part 5, so that it looks like::

                Product  Service   Total
        Name
        Kahn     1280.8    170.4  1451.2
        Lin      1262.5    246.0  1508.5
        Mason     876.7    223.5  1100.2
        Porter    886.2    143.0  1029.2
        All      4306.2    782.9  5089.1

7. Reproduce the answer of part 3 using the groupby() method.

8. Reproduce the answer of part 4 using the groupby() method.

   Your first attempt might produce this DataFrame::

                Quarter  Product  Service
        Name
        Kahn         10   1280.8    170.4
        Lin          10   1262.5    246.0
        Mason        10    876.7    223.5
        Porter       10    886.2    143.0

   The 'Quarter' column is not useful; a slight modification can be used
   to keep just the 'Product' and 'Service' columns.


See :ref:`pandas-dataframe-solution`.
"""

import pandas as pd
```

```
df1 = pd.read_csv('sales.csv')


# 1

print "1.", "-" * 65
print df1.pivot(index='Name', columns='Quarter', values='Product')
print


# 2

print "2.", "-" * 65
print df1.pivot_table(rows='Name', cols=['Quarter'])
print


# 3

print "3.", "-" * 65
print df1.pivot_table(values=['Product', 'Service'], rows='Quarter', aggfunc='sum')
print

# 4

df2 = df1.pivot_table(values=['Product', 'Service'], rows='Name', aggfunc='sum')

print "4.", "-" * 65
print df2
print


# 5

df2 = df1.pivot_table(values=['Product', 'Service'], rows='Name', aggfunc='sum', margins=True)

print "5.", "-" * 65
print df2
print


# 6

df2['Total'] = df2['Product'] + df2['Service']

print "6.", "-" * 65
print df2
print


# 7

print "7.", "-" * 65
print df1.groupby('Quarter').sum()
print
```

```
# 8

print "8.", "-" * 65
# Use groupby(), and then keep only the 'Product' and 'Service' columns
# before applying the sum() method.
print df1.groupby('Name')['Product', 'Service'].sum()
print

# ...or drop the 'Quarter' column before performing the groupby operation:
# print df1.drop('Quarter', axis=1).groupby('Name').sum()
```

2.5.3 Wind Statistics - Solution

pandas_wind_statistics_solution.py:

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

This exercise is an alternative version of the Numpy exercise but this time we
will be using pandas for all tasks. The data have been modified to contain some
missing values identified by -0.00 or NaN depending on whether the measurement
couldn't be trusted or whether no data was collected. Using pandas should make
this exercise easier, in particular for the bonus question.


Of course, you should be able to perform all of these operations without using a
for loop or other looping construct.


Topics: Pandas, time-series

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

        Yr Mo Dy   RPT   VAL   ROS   KIL   SHA   BIR   DUB   CLA   MUL   CLO   BEL   MAL
        61  1  1 15.04 14.96 13.17  9.29 -0.00  9.87 13.67 10.25 10.83 12.58 18.50 15.04
        61  1  2 14.71   NaN 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   NaN  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 'read_table' function from pandas to read the data into
   a DataFrame. Make sure to convert all missing values (NaN and -0.00 to
   np.nan).

2. Replace the first 3 columns by a proper datetime index, created manually.

3. Compute how many values are missing for each location over the entire record.
   They should be ignored in all calculations below. Compute how many
   non-missing values there are in total.

4. Calculate the mean windspeeds of the windspeeds over all the locations and
   all the times (a single number for the entire dataset).

5. 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)

6. 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)

7. Find the average windspeed in January for each location.  Treat
   January 1961 and January 1962 both as January.

8. Downsample the record to a yearly, monthly frequency and weekly frequency for
   each location.

9. Plot linearly and with a candle plot the monthly data for each location.

(Used to be a bonus)

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

11. 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.

Notes
~~~~~

This solution has been tested with Pandas version 0.7.3.

The original data from which these were derived 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.

"""

from pandas import read_table
from datetime import datetime

# 1.
wind_data_df = read_table('wind.data', sep = '\s*', header = 1)

# 2.
# Extract the data part
data_df = wind_data_df.ix[:,3:]

# the fully generic way
index = []
for row in range(len(data_df)):
    year = int(1900+wind_data_df.ix[row,0])
    month = int(wind_data_df.ix[row,1])
    day = int(wind_data_df.ix[row,2])
    index.append(datetime(year, month, day))
data_df.index = index

print "Data:", data_df
```

```
# Non-missing values at each location
print "3. Number of non-missing values for each location:"
print data_df.count()
non_null_count = data_df.count().sum()
print "There are {0} non-missing values in the entire dataset".format(non_null_count)
print

print '4. Mean over all values'
total = data_df.sum().sum()
print '  mean:', total/non_null_count
print

print '5. Statistics over all days at each location'
print '  min:', data_df.min()
print '  max:', data_df.max()
print '  mean:', data_df.mean()
print '  standard deviation:', data_df.std()
print

print '6. Statistics over all locations for each day'
print '  min:', data_df.min(axis=1)
print '  max:', data_df.max(axis=1)
print '  mean:', data_df.mean(axis=1)
print '  standard deviation:', data_df.std(axis=1)
print

monthly_grouped = data_df.groupby(lambda d: d.month)
monthly_means = monthly_grouped.mean()
print '7. Mean wind speed for January in each location'
print monthly_means.ix[1]
print

# Downsample the data to yearly, monthly and weekly data
print "8. Downsampled data:"
# To avoid loosing the first few days, identify which day of the week the first
# day corresponds to
yearly_group = data_df.groupby(lambda x: x.year)
print "Yearly:", yearly_group.mean()

monthly_group = data_df.groupby(lambda x: (x.year,x.month))
print "Monthly:", monthly_group.mean()

# For a given entry in the index, identify which week it belongs to (multiple of
# 7).
which_week = lambda x: (x-data_df.index[0]).days / 7
weekly_group = data_df.groupby(which_week)
print "Weekly data:", weekly_group.mean()
print

# 9. Plots
monthly_data = monthly_group.mean()
from matplotlib import pyplot
monthly_data.plot()
# Force this plot to happen in a separate figure
pyplot.figure()
monthly_data.boxplot()
pyplot.show()
```

```
# 10. This is just another way to group records:
unique_monthly_grouped = data_df.groupby(lambda d: (d.month, d.year))
print '10. Mean wind speed for each month in each location'
print unique_monthly_grouped.mean()
print


# 11. Weekly stats over the first year
first_year = data_df.ix[:52*7,:]
weekly_first_year = first_year.groupby(which_week)
stats = weekly_first_year.apply(lambda x: x.describe())
import pandas
pandas.set_printoptions(max_rows=500, max_columns = 15, notebook_repr_html=False)
print stats
```

2.5.4 Modeling Climate Data in Pandas - Solution

pandas_climate_data_solution.py:

```
"""
Modeling Climate Data in Pandas
-------------------------------

Topics: Pandas, HDF5, FTP file retrieval.

Introduction
~~~~~~~~~~~~

The National Climatic Data Center (NCDC) provides FTP access to extensive
historic and current climate data, collected from weather stations across the
globe.  In this extended exercise, we will download 'Global Surface Summary of
the Day' (GSOD) data from the NCDC servers and use Pandas to analyse and plot
selected data.

The data that we will analyse are organized by year, and stored under the URL:

    ftp://ftp.ncdc.noaa.gov/pub/data/gsod

The file

    ftp://ftp.ncdc.noaa.gov/pub/data/gsod/readme.txt

describes the data format; a copy of this file is stored in the exercise
folder under the name 'gsod_readme.txt'.


1. Retrieve GSOD datafiles for 2008 for the following 3 cities.

    Austin, US
    Cambridge, UK
    Paris, FR

   Store the files locally.  The full URLs for the 3 data files are:

      Austin:     ftp://ftp.ncdc.noaa.gov/pub/data/gsod/2008/999999-23907-2008.op.gz
      Cambridge:  ftp://ftp.ncdc.noaa.gov/pub/data/gsod/2008/035715-99999-2008.op.gz
      Paris:      ftp://ftp.ncdc.noaa.gov/pub/data/gsod/2008/071560-99999-2008.op.gz
```

```
   Hint: Use the urlretrieve function from the standard library urllib module
   for this.  You can also use ftplib, if you prefer.

   (Note: in case of difficulties accessing the server, the relevant files are
    also stored in the 'data' subfolder of the exercise folder.)

2. For each of the three cities above:

     (a) Read the datafile into a Pandas DataFrame object with suitable column
         headings.

     (b) Replace missing values (as described in the readme file) for the
         numeric columns with NaNs.

     (c) Use the Year, Month and Day information in the input to construct and
         assign a suitable index for the DataFrame.

     (d) (Optional) Replace the 'FRSHTT' indicators column with six separate
         boolean columns giving fog, rain, snow, hail, thunder and tornado
         indicators.

   Hints: pandas.read_fwf reads data in fixed-format columns.  You may also
   find the gzip standard library module useful for dealing with the compressed
   .gz files.

   Bonus: Extract field heading, column heading and missing data information
   programmatically from the gsod_readme.txt file, rather than copying and
   pasting.

3. The Austin data for the start of the year are missing.  Retrieve alternative
   Austin data for 2008 from the URL:

      Austin (2): ftp://ftp.ncdc.noaa.gov/pub/data/gsod/2008/722544-13958-2008.op.gz

   Now:

     (a) Create a scatter plot comparing the two sets of Austin data.  (You'll
         first need to adjust the two datasets to a common set of indices.)

     (b) Compute a correlation coefficient for the two Austin datasets.

     (c) Finally, fill in the missing days in the original Austin dataset using
         the data from the second Austin dataset.

4. Create a single pandas Panel containing all three datasets.

5. From the panel you created in part 4, extract a DataFrame giving just
   the mean temperatures for each location.  Use this to plot all three
   temperatures on the same graph.

6. Compute and report mean temperatures for the whole year for each location.

7. Compute and plot rolling means with a 14-day window for each of the
   locations.

8. Compute monthly total rainfalls for each location, dropping any locations
   for which no rainfall data are available.  Create a bar plot showing these
   totals.
```

```
BONUS:

1. Write a 'CachingClimateDataStore' class that allows easy retrieval of the
   GSOD data, and stores the corresponding pandas dataframe locally in an HDF5
   file.

   For example, your class might be used something like this:

      >>> store = CachingClimateDataStore()
      >>> # Next line goes to the FTP server, fetches the file, turns
      >>> # it into a dataframe and stores it locally.
      >>> paris_climate = store['071560-99999-2008']
      >>> # A second retrieval operation goes straight to the local .h5 file.
      >>> paris_climate = store['071560-99999-2008']

   (Hints: (1) look at the pandas.HDFStore class.  (2) you can override item
   access via the __getitem__ special method.)

2. (Half day to full day team project, open-ended!)  Use Traits and Chaco (or a
   UI toolkit of your choice) to write an application that allows the user to:

     - retrieve data for cities on request

     - display plots showing a summary of the data for any particular year and
       location

     - compare two locations (or two years for the same location), showing
       scatter plots, combined graphs, and correlation coefficients.

See :ref:`pandas-climate-data-solution`.

"""

# Standard library imports.
import contextlib
import cStringIO
import datetime
import ftplib
import gzip

# 3rd party library imports.
from matplotlib import pyplot
import numpy
import pandas
import tables


def read_column_descriptions():
    """
    Retrieve field names and positions from the 'gsod_readme.txt' file.

    """
    with open('gsod_readme.txt') as f:
        readme = f.read()

    # Extract portion of the readme that describes the fields.
    readme = readme[readme.index('FIELD'):]
```

```
    readme = readme[:readme.index('*****')]

    # Find column indices for this portion of the README.
    position = readme.index('POSITION')
    field_type = readme.index('TYPE')
    description = readme.index('DESCRIPTION')

    # Extract lines; throw away the header.
    readme_lines = readme.splitlines()[1:]

    # Extract chunks: each chunk is a list of rows giving
    # information about one field of the climate data file.
    rows = [
        i for i, line in enumerate(readme_lines)
        if line[:position].strip()
    ]
    chunks = [
        readme_lines[start_row:end_row]
        for start_row, end_row in zip(rows, rows[1:] + [len(readme_lines)])
    ]

    # Extract field names.
    fields = [chunk[0][:position].strip() for chunk in chunks]
    # Field names in the readme.txt are not unique: names 'Count' and 'Flag'
    # occur multiple times.  We prefix each occurrence of 'Count' or 'Flag'
    # with the field name from the preceding column.
    fields = [
        fields[i - 1] + field if field in ['Count', 'Flag'] else field
        for i, field in enumerate(fields)
    ]

    # Extract field positions.
    positions = []
    for chunk in chunks:
        start, end = chunk[0][position:field_type].split('-')
        # Columns in readme.txt are 1-based with an *inclusive* end column.
        # Adjust for 0-based values with an *exclusive* end column.
        positions.append((int(start) - 1, int(end)))

    # Extract descriptions, then information about missing values.
    missing = {}
    descriptions = [
        ''.join([line[description:] + '\n' for line in chunk])
        for chunk in chunks
    ]
    for field, description in zip(fields, descriptions):
        words = description.split()
        if 'Missing' in words:
            index = words.index('Missing')
            assert words[index + 1] == '='
            missing[field] = float(words[index + 2])

    return fields, positions, missing


def read_raw_data(file):
    """
    Read raw climate data from the given open file or file-like object,
```

```
    returning a pandas DataFrame object.

    """
    fields, positions, missing = read_column_descriptions()

    # We convert the MAXFlag and MINFlag columns to booleans directly on
    # reading.
    field_converters = {
        'MAXFlag': lambda s: s == '*',
        'MINFlag': lambda s: s == '*',
    }

    # Use read_fwf for reading fixed-width columns.
    climate_data = pandas.read_fwf(
        file,
        colspecs=positions,
        names=fields,
        skiprows=1,
        converters=field_converters,
    )

    # Remove 'YEAR' and 'MODA' columns, and use them to set the index.
    year = climate_data.pop('YEAR')
    month, day = divmod(climate_data.pop('MODA'), 100)
    climate_data.index = map(datetime.datetime, year, month, day)

    # Replace missing values in numeric columns.
    for field, missing_value in missing.items():
        series = climate_data[field]
        series[series == missing_value] = numpy.NaN

    # Replace the indicators 'FRSHTT' column with 6 separate boolean columns.
    indicators = climate_data.pop('FRSHTT')
    climate_data['FOG'] = indicators // 10 ** 5 == 1
    climate_data['RAIN'] = indicators // 10 ** 4 % 10 == 1
    climate_data['SNOW'] = indicators // 10 ** 3 % 10 == 1
    climate_data['HAIL'] = indicators // 10 ** 2 % 10 == 1
    climate_data['THUNDER'] = indicators // 10 % 10 == 1
    climate_data['TORNADO'] = indicators % 10 == 1
    return climate_data


class RemoteClimateDataStore(object):
    """
    Dict-like class allowing read-only access to climate data FTP site,
    returning pandas DataFrames.

    """
    def __init__(self, site='ftp.ncdc.noaa.gov', directory='pub/data/gsod'):
        self.site = site
        self.directory = directory

    def __getitem__(self, location):
        year = location.split('-')[-1]
        dirname = '{}/{}'.format(self.directory, year)
        filename = '{}.op.gz'.format(location)

        # Retrieving an FTP file the hard way.  See also urllib.urlretrieve.
```

```
        connection = ftplib.FTP(self.site)
        with contextlib.closing(connection):
            connection.login()
            connection.cwd(dirname)
            compressed_output = cStringIO.StringIO()
            connection.retrbinary(
                'retr {}'.format(filename),
                compressed_output.write,
            )
            compressed_output.seek(0)
            with gzip.GzipFile(fileobj=compressed_output) as z:
                return read_raw_data(z)


class CachingClimateDataStore(object):
    """
    Dict-like class representing store for raw dataframes corresponding
    to climate data files.

    """
    def __init__(self, local_store=None, remote_store=None):
        if local_store is None:
            local_store = pandas.HDFStore('gsod_climate_data.h5')
        if remote_store is None:
            remote_store = RemoteClimateDataStore()
        self.local_store = local_store
        self.remote_store = remote_store

    def __getitem__(self, location):
        try:
            df = self.local_store[location]
        # Pandas 0.8 gives a NodeError;  0.10 gives a KeyError.  Catch both
        # to be on the safe side.
        except (tables.NodeError, KeyError):
            df = self.remote_store[location]
            self.local_store[location] = df
        return df


def main():
    # 1. Retrieve climate data files for 2008 for Austin, Cambridge and Paris.
    # 2. For each city, read the data into a Pandas DataFrame object.

    store = CachingClimateDataStore()
    austin_climate = store['999999-23907-2008']
    cambridge_climate = store['035715-99999-2008']
    paris_climate = store['071560-99999-2008']
    alternative_austin_climate = store['722544-13958-2008']

    # Plotting the two Austin temperature data on the same plot.
    austin_climate['TEMP'].plot()
    alternative_austin_climate['TEMP'].plot()
    pyplot.show()

    # Creating a scatter plot for the two datasets.  Requires matching
    # the indices first, using the 'align' method.
    austin1, austin2 = austin_climate['TEMP'].align(
        alternative_austin_climate['TEMP'],
```

```
        join='inner',
    )
    pyplot.figure()
    pyplot.scatter(austin1, austin2)
    pyplot.show()

    # Computing the correlation.
    print "Correlation coefficient between the two Austin temperature sets: ",
    print austin_climate['TEMP'].corr(alternative_austin_climate['TEMP'])

    # Joining the Austin data into a single dataframe.
    austin_climate = austin_climate.combine_first(alternative_austin_climate)

    # 3. Create a single pandas Panel containing all three datasets.
    climate_panel = pandas.Panel(dict(
            Cambridge=cambridge_climate,
            Austin=austin_climate,
            Paris=paris_climate,
    ))

    # Extract a DataFrame giving temperatures only for the three locations.
    temp_df = climate_panel.minor_xs('TEMP')

    # ... and use this to plot all three temperatures on the same graph.
    temp_df.plot()
    pyplot.show()

    # Find mean temperatures for the year in Cambridge, Austin and Paris.
    print "Mean temperatures"
    print temp_df.mean()

    # Compute and plot rolling means with a 14-day window for the temperature.
    rolling_means = pandas.rolling_mean(temp_df, window=10)
    rolling_means.plot()
    pyplot.show()

    # Compute monthly total rainfalls in Austin and Paris.
    rainfall = climate_panel.minor_xs('PRCP')
    monthly_rainfall = rainfall.groupby(lambda dt: dt.month).agg(numpy.sum)
    # Drop columns containing no values.
    monthly_rainfall = monthly_rainfall.dropna(axis=1, how='all')

    # Plot bar plot.
    monthly_rainfall.plot(kind='bar')
    pyplot.show()


if __name__ == '__main__':
    main()
```

2.6 Interface With Other Languages - Solutions

2.6.1 Mandelbrot Cython - Solution 1

mandelbrot_solution.pyx:

```
"""
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 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.

"""

from numpy import empty

cdef int mandelbrot_escape(double x, double y, int n):
    """ Mandelbrot set escape time algorithm in real and complex components
    """
    cdef double z_x = x
    cdef double z_y = y
    cdef int i
    for i in range(n):
        z_x, z_y = z_x**2 - z_y**2 + x, 2*z_x*z_y + y
        if z_x**2 + z_y**2 >= 4.0:
            break
    else:
        i = -1
    return i

def generate_mandelbrot(xs, ys, int n):
    """ Generate a mandelbrot set """
    cdef int i,j
    cdef int N = len(xs)
```

```
    cdef int M = len(ys)
    d = empty(dtype=int, shape=(len(xs), len(ys)))
    for j in range(M):
        for i in range(N):
            d[j,i] = mandelbrot_escape(xs[i], ys[j], n)
    return d
```

2.6.2 Mandelbrot Cython - Solution 2

mandelbrot_solution2.pyx:

```
"""
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 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.

"""

from numpy import empty
cimport numpy as np

cdef int mandelbrot_escape(double x, double y, int n):
    """ Mandelbrot set escape time algorithm in real and complex components
    """
    cdef double z_x = x
    cdef double z_y = y
```

```
    cdef int i
    for i in range(n):
        z_x, z_y = z_x**2 - z_y**2 + x, 2*z_x*z_y + y
        if z_x**2 + z_y**2 >= 4.0:
            break
    else:
        i = -1
    return i

def generate_mandelbrot(np.ndarray[double, ndim=1] xs, np.ndarray[double, ndim=1] ys, int n):
    """ Generate a mandelbrot set """
    cdef int i,j
    cdef int N = len(xs)
    cdef int M = len(ys)

    cdef np.ndarray[int, ndim=2] d = empty(dtype='i', shape=(N, M))
    for j in range(M):
        for i in range(N):
            d[j,i] = mandelbrot_escape(xs[i], ys[j], n)
    return d
```

2.6.3 Mandelbrot Cython - Solution 3

mandelbrot_solution3.pyx:

```
#cython: boundscheck=False
"""
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 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.

This code uses complex numbers requiring a recent version of Cythnon > 0.11.2
"""


from numpy import empty
cimport numpy as np

cdef int mandelbrot_escape(double complex c, int n):
    """ Mandelbrot set escape time algorithm in real and complex components
    """
    cdef double complex z
    cdef int i
    z = c
    for i in range(n):
        z = z*z + c
        if z.real*z.real + z.imag*z.imag >= 4.0:
           break
    else:
        i = -1
    return i

def generate_mandelbrot(np.ndarray[double, ndim=1] xs, np.ndarray[double, ndim=1] ys, int n):
    """ Generate a mandelbrot set """
    cdef unsigned int i,j
    cdef unsigned int N = len(xs)
    cdef unsigned int M = len(ys)
    cdef double complex z

    cdef np.ndarray[int, ndim=2] d = empty(dtype='i', shape=(M, N))
    for j in range(M):
        for i in range(N):
            z = xs[i] + ys[j]*1j
            d[j,i] = mandelbrot_escape(z, n)
    return d
```

2.6.4 Cythonize rankdata - Solution

cython_rankdata_solution.pyx:

```
import numpy as np
cimport numpy as np
cimport cython


@cython.boundscheck(False)
@cython.cdivision(True)
def rankdata(a):
    """
    Ranks the data, dealing with ties appropriately.

    Equal values are assigned a rank that is the average of the ranks that
    would have been otherwise assigned to all of the values within that set.
```

```
    Ranks begin at 1, not 0.

    Parameters
    ----------
    a : array_like
        This array is first flattened.

    Returns
    -------
    rankdata : ndarray
         An array of length equal to the size of `a`, containing rank scores.

    Examples
    --------
    >>> rankdata([0, 2, 2, 3])
    array([ 1. ,  2.5,  2.5,  4. ])

    """
    # Type declarations.   Note that these 'cdef' statements are the *only*
    # changes that we've made in the function!
    cdef unsigned int i, j, n, sumranks, dupcount
    cdef np.ndarray[np.int_t, ndim=1] ivec
    cdef np.ndarray[np.float64_t, ndim=1] svec
    cdef np.ndarray[np.float64_t, ndim=1] ranks
    cdef double averank

    # Convert the array to floating point, and flatten.
    a = np.ravel(np.asarray(a, dtype=np.float64))
    n = len(a)

    # Sort, in two steps: use argsort to get the sorted indices, and then
    # use fancy indexing to create the sorted array.
    ivec = np.argsort(a)
    svec = a[ivec]

    # Create the array of ranks, taking into account repeated values.
    sumranks = 0
    dupcount = 0
    ranks = np.zeros(n, float)
    for i in xrange(n):
        sumranks += i
        dupcount += 1
        if i == n - 1 or svec[i] != svec[i + 1]:
            averank = sumranks / float(dupcount) + 1
            for j in xrange(i - dupcount + 1, i + 1):
                ranks[ivec[j]] = averank
            sumranks = 0
            dupcount = 0
    return ranks
```

2.7 Statistics - Solutions

2.7.1 Linear Regression of Mileage Data - Solution

mileage_solution.py:

```
"""
Linear Regression of Mileage Data
---------------------------------

The file "mileage_data_1991.txt" contains information from 1991 about
several models of cars.  The fields in the file are:

MAKE/MODEL: Manufacturer and model name
VOL: Cubic feet of cab space
HP: Engine horsepower
MPG: Average miles per gallon
SP: Top speed (mph)
WT: Vehicle weight (100 lb)

1. Read the data into an array. (This is already done.)

2. Use scipy.stats.linregress to compute the linear regression line
   for the weight (WT) and mileage (MPG) of the cars.  Print the correlation
   coefficient.

3. Repeat 2 for the weight and "gallons-per-mile" for the cars.

4. (Optional) For 2 and 3, plot the data with the weight on the x-axis, and
   plot the regression line. (Use separate figures for 2 and 3.)

See :ref:`mileage-solution`.
"""


from numpy import loadtxt, array
from matplotlib.pyplot import plot, show, xlim, xlabel, ylabel, figure
from scipy.stats import linregress

VOLUME = 0
HP = 1
MPG = 2
SP = 3
WT = 4

# Skip the first row, which is the headers, and don't try to read the
# first column, which is the make and model.
data = loadtxt('mileage_data_1991.txt', skiprows=1, usecols=(1,2,3,4,5))

# For convenience, pull out the WT and MPG columns.
wt = data[:,WT]
mpg = data[:,MPG]

gpm = 1.0/mpg

# Compute the linear regressions.
m1, b1, r1, p1, se1 = linregress(wt, mpg)
m2, b2, r2, p2, se2 = linregress(wt, gpm)

print "MPG: r =", r1
print "GPM: r =", r2

wt_lim = array([wt.min(), wt.max()])
```

```
figure(1)
plot(wt, mpg, 'bo')
plot(wt_lim, m1*wt_lim + b1, 'r')
xlim(15,60)
xlabel('Weight (100 lb)')
ylabel('MPG')

figure(2)
plot(wt, gpm, 'bo')
plot(wt_lim, m2*wt_lim + b2, 'r')
xlim(15,60)
xlabel('Weight (100 lb)')
ylabel('GPM')
show()
```

2.7.2 Chi-square test - Solution

chi_square_solution.py:

```
"""
Chi-square test
---------------

1. A six-sided die is rolled 50 times, and the number of occurrences
   of each side are recorded in the following table::

      +-------------+-----+-------+------+-----+------+------+
      | Side        |  1  |   2   |   3  |  4  |   5  |   6  |
      +-------------+-----+-------+------+-----+------+------+
      | Occurrences |  6  |   1   |   9  |  7  |  14  |  13  |
      +-------------+-----+-------+------+-----+------+------+

   Is the die fair?   That is, is each side equally likely?


2. 17 woman and 18 men are asked whether they prefer chocolate or
   vanilla ice cream.  The following table shows the result::

       +-------+-----------+---------+
       | Group | Chocolate | Vanilla |
       +-------+-----------+---------+
       | Women |    12     |    5    |
       +-------+-----------+---------+
       | Men   |    10     |    8    |
       +-------+-----------+---------+

   Use the chi-square test to investigate the relation between an
   individual's sex and the individual's preference for ice cream.
   Then apply Fisher's exact test (scipy.stats.fisher_exact), and
   compare the results.

See :ref:`chi-square-solution`.
"""

import numpy as np
from scipy.stats import chisquare, chi2_contingency, fisher_exact
```

```
# 1. Six-sided die.

print "H0: Each side of the die is equally probable."

num = np.array([6, 1, 9, 7, 14, 13])

chi2, p = chisquare(num)

print "p = %6.4f" % p
if p < 0.05:
    print "Reject H0."
else:
    print "Do not reject H0."


# 2. Ice cream preference.

print
print "H0: Preference for ice cream flavor is independent of sex."

table = np.array([[12, 5], [10, 8]])

chi2, p, dof, ex = chi2_contingency(table)

print "Chi-squared test: p = %6.4f" % p
if p < 0.05:
    print "Reject H0."
else:
    print "Do not reject H0."

odds_ratio, pvalue = fisher_exact(table)
print "Fisher exact test: p = %6.4f" % pvalue
if pvalue < 0.05:
    print "Reject H0."
else:
    print "Do not reject H0."
```

2.7.3 Test for Normal Distribution - Solution

normal_test_solution.py:

```
"""
Test for Normal Distribution
----------------------------

The file "rdata.txt" contains a column of numbers.

1. Load the data "rdata.txt" into an array called 'x'.

2. Apply the Anderson-Darling (scipy.stats.anderson) and Kolmogorov-Smirnov
   (scipy.stats.kstest) tests to 'x', using a 5 percent confidence level.
   What do you conclude?

3. Apply the Anderson-Darling and Kolmogorov-Smirnov tests to 'dx', where
   'dx = numpy.diff(x)', using a five percent confidence level.  What do you
   conclude?
```

```
See :ref:`normal-test-solution`.
"""
from numpy import loadtxt, diff
from scipy.stats import anderson, kstest, zscore

x = loadtxt('rdata.txt')

print "Testing x"
a, c, s = anderson(x, 'norm')
print "anderson:",
print a, c[2],
if a > c[2]:
    print "not normal"
else:
    print "no conclusion"

kstat, kp = kstest(zscore(x), 'norm')
print "ks:",
print kp,
if kp < 0.05:
    print "not normal"
else:
    print "no conclusion"

print
print "Testing dx"
dx = diff(x)

a1, c1, s1 = anderson(dx, 'norm')
print "anderson:",
print a1, c1[2],
if a1 > c1[2]:
    print "not normal"
else:
    print "no conclusion"

kstat1, kp1 = kstest(zscore(dx), 'norm')
print "ks:",
print kp1,
if kp1 < 0.05:
    print "not normal"
else:
    print "no conclusion"
```

2.8 Traits - Solutions

2.8.1 Trait Particle - Solution

trait_particle_solution.py:

```
"""
Trait Particle
--------------

Define a Traits-based class for a Particle.
```

```
1. Define a Particle class derived from HasTraits. A Particle
   should have a :attr:`mass` and a :attr:`velocity` that are both
   floating point values.  It should also have two read-only
   Property traits, :attr:`energy` and :attr:`momentum`, given
   by::

       energy = 0.5 * mass * velocity **2
       momentum = mass * velocity

   After defining the class, create an instance of it and set/get
   some its traits to test its behavior.

2. Create a UI that groups the mass and velocity side-by-side at
   the top of a dialog with the energy and momentum below.

"""

from traits.api import HasTraits, Float, Property
from traitsui.api import View, VGroup, HGroup, Item
from traitsui.menu import OKCancelButtons


class Particle(HasTraits):

    # The mass of the particle.
    mass = Float

    # The velocity of the particle.
    velocity = Float

    # The energy of the particle.
    energy = Property(Float, depends_on=['mass', 'velocity'])

    # The momentum of the particle.
    momentum = Property(Float, depends_on=['mass', 'velocity'])

    view = \
        View(
            VGroup(
                HGroup(
                    Item('mass'),
                    Item('velocity'),
                ),
                HGroup(
                    Item('energy', style='readonly', width=80),
                    Item('momentum', style='readonly', width=80),
                ),
            ),
            buttons=OKCancelButtons,
        )


    def _get_energy(self):
        return 0.5 * self.mass * self.velocity**2

    def _get_momentum(self):
        return self.mass * self.velocity
```

```
if __name__ == "__main__":
    p = Particle(mass=2, velocity=1.5)
    print "energy is", p.energy
    p.configure_traits()
    print "energy is", p.energy
```

2.8.2 Trait Person - Solution

trait_person_solution.py:

```
"""
Trait Person
------------

Define a Traits based class for a Person.

Hint: See the demo/traits_examples/configure_ui_1.py

1. Define a person class. A person should have a
   :attr:`first_name` and :attr:`last_name` that are both strings,
   an :attr:`age` that is a Float value, and a :attr:`gender` that is
   either 'male' or 'female'.  After defining the class,
   create an instance of it and set/get some its traits
   to test its behavior.  Try setting names to strings
   and floats and see what happens.

2. Add a static trait listener that prints the age when
   it changes.

3. Create a UI that groups the name at the top of a dialog
   with the gender and age below.

See :ref:`trait-person-solution`.
"""
from traitsui.api import View, VGroup, HGroup, Group, Item
from traits.api import HasTraits, Float, Str, Enum

class Person(HasTraits):
    first_name = Str('Joe')
    last_name = Str('Doe')
    age = Float
    gender = Enum('male','female')

    view = View(Group(
                      HGroup(
                             Item('first_name', label='First', springy=True),
                             Item('last_name', label='Last', springy=True),
                             show_border=True,
                             label="Name"
                      ),
                      VGroup('gender','age'
                      ),
                )
            )

    def _age_changed(self, old, new):
```

```
        print 'new age:', self.age

person = Person()
person.edit_traits()
```

2.8.3 Trait Function - Solution

trait_function_solution.py:

```
"""
Trait Function
--------------

Traits-based class for calculating a function.

1. Write a Traits-based class that calculates the following signal::

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

   Make *a*, *b*, and *c* floating point traits with default
   values of 2.0, 0.76, 1.0. The *x* variable should be an array
   trait with the default value [0, 0.5, 1, 1.5, ..., 5.0].
   The *y* value should be a property trait that updates any time
   *a*, *b*, *c* or *x* changes.

   Once your class is defined, create an instance of the object and
   print its *y* value.

2. Define a function 'printer(value)' that prints an array, and set
   it to be an external trait listener for the trait *y* on the
   instance created in 1, so *y*'s value it printed whenever it
   changes.

3. Create a UI that displays *a*, *b*, and *c* as text boxes.

4. Create a UI that displays *a*, *b*, and *c* as sliders,
   and *x* and *y* as read-only arrays.  Use an ArrayEditor for the
   display of the arrays, and set the 'width' keyword of the
   ArrayEditor to a value large enough for the values in the array.

"""

from traitsui.api import View, Item
from traits.api import HasTraits, Float, Property, Array
from numpy import linspace, exp, set_printoptions, get_printoptions
from traitsui.api import RangeEditor, ArrayEditor


# 1. Create class that updates y whenever a, b,c or x change.

class Exponential(HasTraits):

    # Amplitude.
    a = Float(2.0)

    # Exponential decay constant.
    b = Float(0.76)
```

```
    # Offset.
    c = Float(1.0)

    # Values at which to evaluate the function.
    x = Array

    # The values of the functions at `x`.
    y = Property(depends_on=['a', 'b', 'c', 'x'])

    def _get_y(self):
        y = self.a * exp(-self.b * self.x) + self.c
        return y

    def _x_default(self):
        return linspace(0.0, 5.0, 6)


func = Exponential()
print func.y


# 2. Define a function `printer(value)` that prints an array, and set
#    it to be an external trait listener for the trait *y*.

def printer(value):
    opt = get_printoptions()
    set_printoptions(precision=2)
    print 'new value:', value
    set_printoptions(**opt)


func.on_trait_change(printer, name='y')
func.a = 2.1


# 3. Create a UI that displays a, b, and c as text boxes.
simple_view = View('a', 'b', 'c')
func.edit_traits(view=simple_view)

# 4. Create a UI that displays a, b, and c as sliders, and x and y
#    using ArrayEditors.
slider_view = View(Item('a', editor=RangeEditor(low=0.0, high=10.0)),
                   Item('b', editor=RangeEditor(low=0.0, high=10.0)),
                   Item('c', editor=RangeEditor(low=0.0, high=10.0)),
                   Item('x', style='readonly', editor=ArrayEditor(width=120)),
                   Item('y', style='readonly', editor=ArrayEditor(width=120)),
                   resizable=True,
                )
func.edit_traits(view=slider_view)
```

2.9 Chaco - Solutions

2.9.1 Function Plot - Solution 1

function_plot_solution.py:

```
from math import pi

from traits.api import HasTraits, Instance, DelegatesTo
from traitsui.api import View, Item, RangeEditor, UItem, Group
from enable.api import ComponentEditor
from chaco.api import Plot, ArrayPlotData

# Local import
from damped_osc import DampedOsc


class DampedOscView(HasTraits):
    """
    Access the model traits using DelegatesTo().
    """

    model = Instance(DampedOsc)

    x = DelegatesTo('model')
    y = DelegatesTo('model')
    a = DelegatesTo('model')
    b = DelegatesTo('model')
    omega = DelegatesTo('model')
    phase = DelegatesTo('model')

    plot = Instance(Plot)

    traits_view = \
        View(
            Group(
                UItem('plot', editor=ComponentEditor(), style='custom'),
            ),
            Item('a', label='a', editor=RangeEditor(low=0.0, high=10.0)),
            Item('b', label='b', editor=RangeEditor(low=0.0, high=2.0)),
            Item('omega', label='omega',
                 editor=RangeEditor(low=0.0, high=10.0)),
            Item('phase', label='phase',
                 editor=RangeEditor(low=-pi, high=pi,
                                    low_label='-pi', high_label="pi")),
            resizable=True,
            width=600,
            height=550,
            title="a * exp(-b*x) * cos(omega*x + phase)",
        )

    def _plot_default(self):
        data = ArrayPlotData(x=self.x, y=self.y)
        plot = Plot(data)
        plot.plot(('x', 'y'), style='line', color='green')
        plot.value_range.set_bounds(-self.a, self.a)
        return plot

    def _y_changed(self):
        # Get the plot's ArrayPlotData object.
        data = self.plot.data
        # Update the value of y in the ArrayPlotData.
        data.set_data('y', self.y)
```

```
    def _a_changed(self):
        self.plot.value_range.set_bounds(-self.a, self.a)


if __name__ == "__main__":
    osc = DampedOsc()
    osc_viewer = DampedOscView(model=osc)
    # We use edit_traits(), so this script should be run from within ipython.
    osc_viewer.edit_traits()
```

2.9.2 Plot adjuster - Solution

plot_adjuster_solution.py:

```
"""
Plot adjuster
-------------

In this problem you will create a Figure containing a combination of
a line plot and a scatter plot, both using the same data.

In addition, the plot should be exposed together with Editors
that allow you to:

1. select from a sequence of functions to be plotted
2. change the color of the scatter-plot markers
3. change the background color of the plot (ColorTrait)
4. change the marker size (using a RangeEditor)
5. change the marker type (marker_trait)

Bonus
~~~~~
1. add User-Interface elements to allow changing the x-range and the
   number of data-points

See: :ref:`plot-adjuster-solution`.

"""

from numpy import linspace, cos, sin
from scipy.special import jn
from functools import partial
j0 = partial(jn, 0)
j1 = partial(jn, 1)
j2 = partial(jn, 2)

from enable.api import ColorTrait, ComponentEditor
from chaco.api import ArrayPlotData, Plot, marker_trait
from traits.api import Enum, HasTraits, Instance, Int, on_trait_change
from traitsui.api import Item, View, RangeEditor, HGroup, spring


class Adjuster(HasTraits):
    plot = Instance(Plot)
    color = ColorTrait("blue")
    marker = marker_trait
    marker_size = Int(4)
```

```
    function = Enum("j0", "j1", "j2", "cos", "sin")
    # Fixme:  This should probably be
    # function = Enum(j0, j1, j2, cos, sin)
    # but I don't know how to create the Editor to display
    # the right text in the drop down.

    xleft = Int(-14)
    xright = Int(14)
    num_points = Int(100)
    traits_view = View(
        Item('color', label="Color", style="custom"),
        Item('marker', label="Marker"),
        Item('marker_size', label="Size", editor=RangeEditor(low=1, high=20)),
        Item('function', label="Y data"),
        Item('plot', editor=ComponentEditor(), show_label=False),
        HGroup(spring, Item('xleft', label='min'),
               spring, Item('num_points', label='N'),
               spring, Item('xright', label='max'),
               spring
               ),
        width=800, height=650, resizable=True,
        title="Select All")

    def _plot_default(self):
        x = linspace(self.xleft, self.xright, self.num_points)
        y = j0(x)
        # Create the data and the PlotData object
        plotdata = ArrayPlotData(x=x, y=y)

        # Create a Plot and associate it with the PlotData
        plot = Plot(plotdata)
        plot.plot(('x', 'y'), type='line', color='red')
        # Create a line plot in the Plot
        self.renderer = plot.plot(("x", "y"), type="scatter", color="blue")[0]
        return plot

    def _function_changed(self, old, new):
        func = eval(self.function)
        xdata = self.plot.data.get_data("x")
        self.plot.data.set_data("y", func(xdata))

    def _color_changed(self):
        self.renderer.color = self.color

    def _marker_changed(self):
        self.renderer.marker = self.marker

    def _marker_size_changed(self):
        self.renderer.marker_size = self.marker_size

    @on_trait_change('xleft, xright, num_points')
    def _xdata_update(self):
        xdata = linspace(self.xleft, self.xright, self.num_points)
        self.plot.data.set_data("x", xdata)
        func = eval(self.function)
        ydata = func(xdata)
        self.plot.data.set_data("y", ydata)
```

```
#============================================================================
# demo object that is used by the demo.py application.
#============================================================================
demo = Adjuster()


if __name__ == "__main__":
    demo.configure_traits()
```

2.9.3 Custom tool - Solution

custom_tool_solution.py:

```
"""
Custom tool
-----------

In this exercise you will add custom interactivity to a Chaco plot by
creating a custom tool that allows a user to add new points by clicking.

1. Create a Chaco Plot that plots sin(x) * x**3 from -14 to 14
2. Create a Custom Tool that responds to mouse clicks and
    a) places a point on the plot at each clicked location and
    b) prints the location of the click to standard-out.

   Hint:  Add the clicked value to the data that is plotted.

Bonus
~~~~~

Add the ability to remove the point nearest the location that is
clicked.
"""

from numpy import linspace, sin, r_

from chaco.api import ArrayPlotData, Plot
from enable.api import BaseTool, ComponentEditor
from traits.api import HasTraits, Instance
from traitsui.api import Item, View


class CustomTool(BaseTool):

    def normal_left_down(self, event):
        # Map the screen-space coordinates back into data space
        new_x, new_y = self.component.map_data((event.x, event.y))
        print "Data:", new_x, new_y

        # Get a reference to the PlotData  (self.component is a Plot)
        plotdata = self.component.data

        # Get the index (x) and value (y) datasources
        x_ary = plotdata.get_data("x")
        y_ary = plotdata.get_data("y")

        # Set the new data arrays
```

```
        plotdata.set_data("x", r_[x_ary, new_x])
        plotdata.set_data("y", r_[y_ary, new_y])


class ScatterPlot(HasTraits):

    plot = Instance(Plot)

    traits_view = View(Item('plot', editor=ComponentEditor(),
                            show_label=False),
                       width=800, height=600, resizable=True,
                       title="Custom Tool")

    def _plot_default(self):
        # Create the data and the PlotData object
        x = linspace(-14, 14, 100)
        y = sin(x) * x**3
        plotdata = ArrayPlotData(x=x, y=y)
        # Create a Plot and associate it with the PlotData
        plot = Plot(plotdata)
        # Create a scatter plot in the Plot
        plot.plot(("x", "y"), type="scatter", color="blue")
        # Add our custom tool to the plot
        plot.tools.append(CustomTool(plot))
        return plot


demo = ScatterPlot()
if __name__ == "__main__":
    demo.configure_traits()
```