Python Tools for Science

Release 1.0

Bartosz Telenczuk

February 09, 2010

CONTENTS

Contents:

CHAPTER

ONE

NUMPY EXAMPLES

Author Bartosz Telenczuk

Contents

1.1 Numpy array type

Importing Numpy (avoid using wildcard imports – PEP8)

```
>>> import numpy as np
```

Create a basic array out of a list:

```
>>> a = np.array( [ 10, 20, 30, 40 ] )
>>> a
array([10, 20, 30, 40])
```

Arrays are standard Python objects with methods and properties. Some important properties are:

`ndarray.ndim` the number of dimensions of an array

`ndarray.shape` a tuple containing a number of elements in each dimmension of the array.

`ndarray.size` the total number of elements

`ndarray.dtype` an object describing the datatype of the elements in the array

```
>>> a.ndim
1
>>> a.shape
(4,)
>>> a.size
4
>>> a.dtype
dtype('int64')
```

1.2 Creating arrays

Create a two-dimmensional array from a nested list:

```
>>> a = np.array([[1, 2], [3, 4]])
>>> print a
[[1 2]
 [3 4]]
>>> a.ndim
2
>>> a.shape
(2, 2)
```

The type of the array can be explicitly specified at creation time:

```
>>> c =np. array( [ [1,2], [3,4] ], dtype=np.complex )
>>> c
array([[ 1.+0.j,  2.+0.j],
       [ 3.+0.j,  4.+0.j]])
```

There are also numerous functions for creating special arrays:

- an array of 0 to 3:

  ```
  >>> np.arange(4)
  array([0, 1, 2, 3])
  ```

- an array of 3 evenly spaced samples from a given range:

  ```
  >>> np.linspace(-np.pi, np.pi ,3)
  array([-3.14159265,  0.        ,  3.14159265])
  ```

- an empty array of a given shape:

  ```
  >>> np.empty((2,2))
  array([[  1.97626258e-323,   2.13112375e-314],
         [  2.12354999e-314,   2.78136356e-309]])
  ```

1.3 Slicing and indexing

You can use slicing and indexing on arrays the same way you do it with lists:

```
>>> a = np.arange(10)
>>> a[2]
2
>>> a[-2]
8
>>> a[1:3]
array([1, 2])
```

For multidimensional arrays use simply two indices:

```
>>> b = np.array([[1, 2], [3 ,4]])
>>> b[:, -1]
array([2, 4])
```

You can also use more fancy indexing:

- indexing with arrays of indices:

  ```
  >>> a = np.arange(12)**2
  >>> i = np.array( [ 1,1,3,8,5 ] )
  >>> a[i]
  array([ 1,  1,  9, 64, 25])
  >>> j = np.array( [ [ 3, 4], [ 9, 7 ] ] )
  >>> a[j]
  array([[ 9, 16],
         [81, 49]])
  ```

- indexing with boolean arrays:

  ```
  >>> a = np.arange(12)
  >>> b = a > 4
  >>> a[b]
  array([ 5,  6,  7,  8,  9, 10, 11])
  ```

Fancy indexing can be also used in assignments:

```
>>> a[b]=0
>>> a
array([0, 1, 2, 3, 4, 0, 0, 0, 0, 0, 0, 0])
```

Note: Always prefer slices over indices: slices are more efficient than fancy indexing.

1.4 Array operations

Arithmetic operators on arrays apply *elementwise*.

```
>>> a = np.array([20, 30, 40, 50])
>>> b = np.arange(4)
>>> a-b
array([20, 29, 38, 47])
>>> b**2
array([0, 1, 4, 9])
>>> 10*np.sin(a)
array([ 9.12945251, -9.88031624,  7.4511316 , -2.62374854])
```

```
>>> a<35
array([ True,  True, False, False], dtype=bool)
```

The product operator performs elementwise multiplication as well. The matrix product can be computed using the `np.dot` function:

```
>>> a * b
array([  0,  30,  80, 150])
>>> np.dot(a, b)
260
```

1.5 Shape manipulations

Change a shape of an array in-place using `reshape` or create a new array with new dimmensions with `resize`:

```
>>> a = np.arange(0, 1, 0.1).reshape((5,2))
>>> a
array([[ 0. ,  0.1],
       [ 0.2,  0.3],
       [ 0.4,  0.5],
       [ 0.6,  0.7],
       [ 0.8,  0.9]])
```

Flatten an array:

```
>>> b = a.ravel()
>>> b
array([ 0. ,  0.1,  0.2,  0.3,  0.4,  0.5,  0.6,  0.7,  0.8,  0.9])
```

Stack two arrays along different dimmensions:

```
>>> a  = np.zeros((2, 2))
>>> b  = np.ones((2, 2))
>>> np.vstack((a, b))
array([[ 0.,  0.],
       [ 0.,  0.],
       [ 1.,  1.],
       [ 1.,  1.]])
```

```
>>> np.hstack((a, b))
array([[ 0.,  0.,  1.,  1.],
       [ 0.,  0.,  1.,  1.]])
```

1.6 Reading/writing arrays from/to a file

Store pickled data in a compressed file:

```
>>> a = np.arange(5)
>>> np.save("test", a)
>>> b = np.load("test.npy")
>>> b
array([0, 1, 2, 3, 4])
```

It is also possible to store several files:

```
>>> b = np.random.randint(0, 10, (2,2))
>>> np.savez("test", a=a, b=b)
>>> npz = np.load("test.npz")
>>> (b==npz['b']).all()
True
```

To export data for use outside python you can use either text (`savetxt`) or binary format (`ndarray.tofile`):

```
>>> np.savetxt("data.txt", b)
```

```
8.000000000000000000e+00 0.000000000000000000e+00
3.000000000000000000e+00 3.000000000000000000e+00
```

1.7 Record arrays

A Record Array allows access to its data using named fields. In other words, instead of referring to the first dimension of a matrix x as x[0], one might name that dimension ‘space’, and use x['space'] instead. Imagine your data being a spreadsheet, then the field names would be the column headings.

```
>>> types = [('pressure',np.float32),('temperature',np.float32),('time',np.float32)]
>>> measurement = np.array([(1005, 20.1, 1), (1007, 21.2, 2), (1005, 21.1, 3)], types)
>>> measurement['pressure']
array([ 1005.,  1007.,  1005.], dtype=float32)
```

You can still view the dat as a 3x3 array:

```
>>> meas_mat = measurement.view((np.float32, 3))
>>> meas_mat
array([[  1.00500000e+03,   2.01000004e+01,   1.00000000e+00],
       [  1.00700000e+03,   2.12000008e+01,   2.00000000e+00],
       [  1.00500000e+03,   2.11000004e+01,   3.00000000e+00]], dtype=float32)
```

1.8 Other functions

1.9 Common idioms

- Check if two arrays are the same

```
>>> a = np.arange(0, 1, 0.1)
>>> b = np.arange(10)*0.1
>>> (b==a).all()
True
```

- Filter elements of an array:

```
>>> a = np.random.randn(10)
>>> a
array([ 0.66250904,  0.67524634, -0.94029827, -0.95658428, -0.33060682,
        0.87412791,  2.00254961,  0.01086208, -0.86924706,  1.4249841 ])
>>> a[a>0]
array([ 0.66250904,  0.67524634,  0.87412791,  2.00254961,  0.01086208,
        1.4249841 ])
```

- Count the positive elements

```
>>> np.sum(a>0)
6
```

CHAPTER

TWO

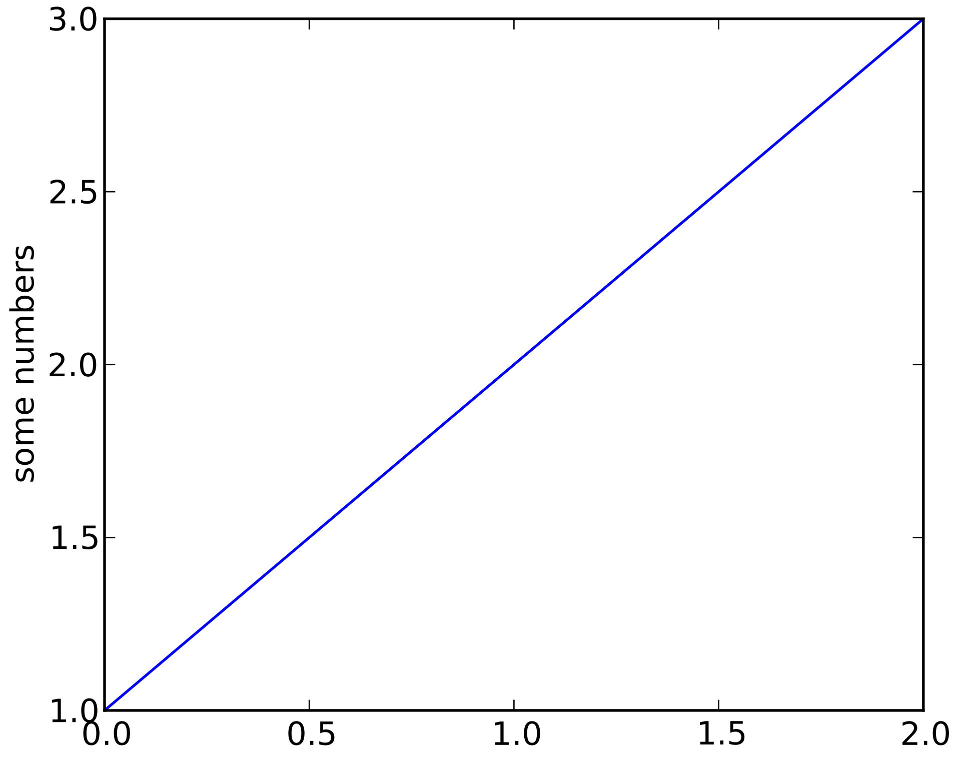
MATPLOTLIB EXAMPLES

This document is based on matplotlib documentation.

2.1 Simple plots

Matplotlib allows one to make nice visualizations of data using only few lines of code:

```
import matplotlib.pyplot as plt
plt.plot([1,2,3])
plt.ylabel('some numbers')
plt.show()
```

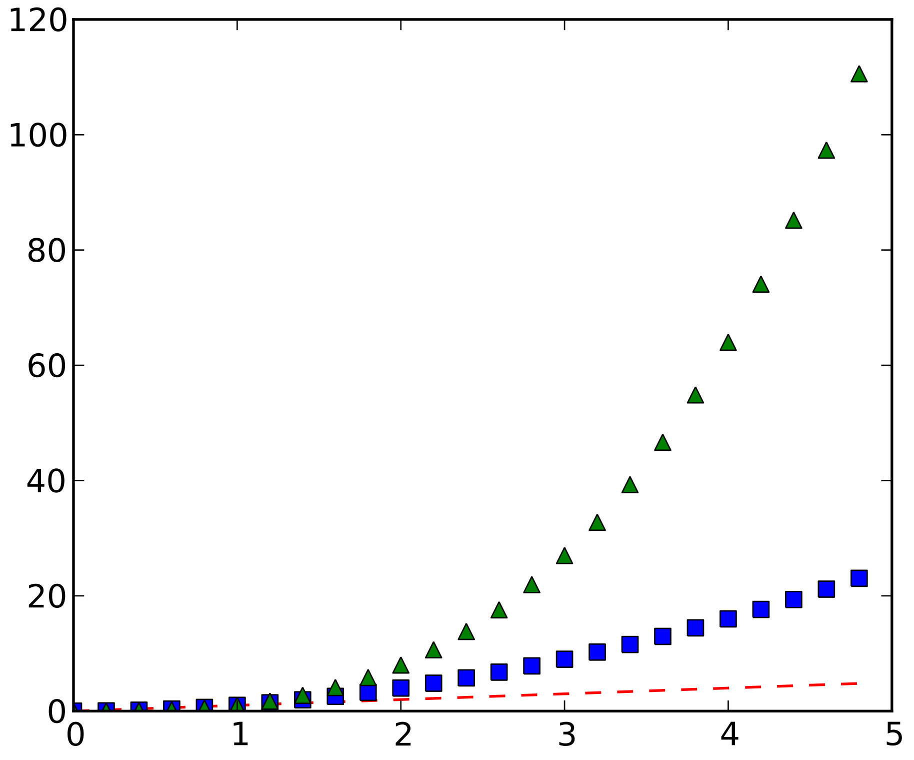


It is also easy to generate XY graphs like this:

```
plt.plot([1,3, 5], [0, 2, -1])
```

Naturally, matplotlib can be combined with numpy. For example, you can plot three different functions using different line styles:

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

# evenly sampled time at 200ms intervals
t = np.arange(0., 5., 0.2)

# red dashes, blue squares and green triangles
plt.plot(t, t, 'r--', t, t**2, 'bs', t, t**3, 'g^')
```



2.2 Setting plot properties

Most of the plot elements are customizable. There are several ways to set desired attributes:

- using keyword args:

  ```
  plt.plot(x, y, linewidth=2.0)
  ```

- using the setter methods of the graphical primitive:

```
line, = plt.plot(x, y, '-')
line.set_antialiased(False) # turn off antialising
```

- using the `setp()` command (especially useful when changing mutliple objects simulataneously):

  ```
  lines = plt.plot(x1, y1, x2, y2)
  plt.setp(lines, color='r', linewidth=2.0)
  ```

You can find out what properties the objects have:

```
>>> lines = plt.plot([1,2,3])
>>> plt.setp(lines)
alpha: float
animated: [True | False]
antialiased or aa: [True | False]
...snip
```

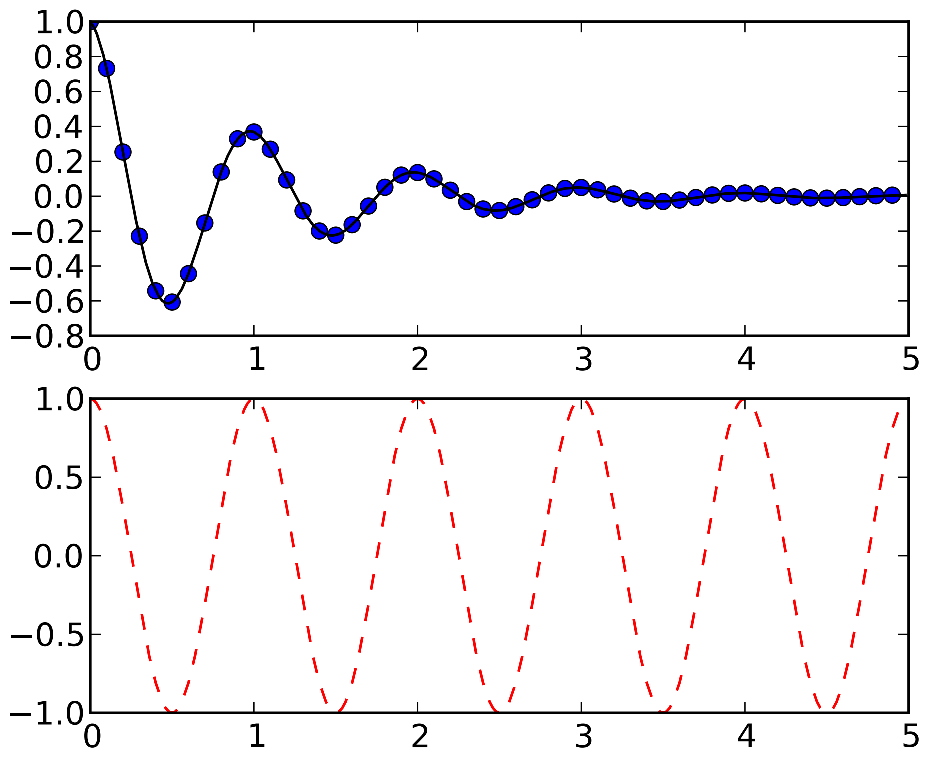
2.3 Working with multiple figures and axes

It is also straigthforward to divide the figure into several axes:

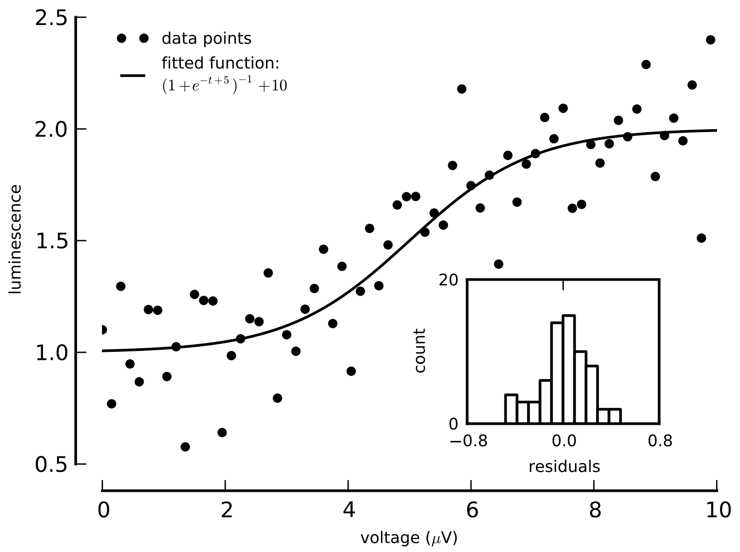
```
import numpy as np
import matplotlib.pyplot as plt

def f(t):
    return np.exp(-t) * np.cos(2*np.pi*t)

t1 = np.arange(0.0, 5.0, 0.1)
t2 = np.arange(0.0, 5.0, 0.02)

plt.figure(1)
plt.subplot(211)
plt.plot(t1, f(t1), 'bo', t2, f(t2), 'k')

plt.subplot(212)
plt.plot(t2, np.cos(2*np.pi*t2), 'r--')
```

2.4 Preparing publication-quality figures

```
from matplotlib import rcParams

#set plot attributes
fig_width = 4  # width in inches
fig_height = 3  # height in inches
fig_size =  [fig_width,fig_height]
params = {'backend': 'Agg',
          'axes.labelsize': 6,
          'axes.titlesize': 8,
          'text.fontsize': 8,
          'legend.fontsize': 6,
          'xtick.labelsize': 8,
```



```
          'ytick.labelsize': 8,
          'figure.figsize': fig_size,
          'savefig.dpi' : 600,
          'font.family': 'sans-serif'}
rcParams.update(params)


import numpy as np
import matplotlib.pyplot as plt

def sigmoid(x):
    return 1./(1+np.exp(-(x-5)))+1

np.random.seed(1234)
t = np.arange(0, 10., 0.15)
y = sigmoid(t) + 0.2*np.random.randn(len(t))
residuals = y - sigmoid(t)

t_fitted = np.linspace(0, 10, 100)

#adjust subplots position
fig = plt.figure()

ax1 = plt.axes((0.145, 0.15, 0.8, 0.775))
plt.plot(t, y, 'k.', label="data points")
plt.plot(t_fitted, sigmoid(t_fitted), 'k-',
         label="fitted function:\n$({1+e^{-t+5}})^{-1}+10$")

#set axis limits
ax1.set_xlim((0, 10.))
ax1.set_ylim((0.5, 2.5))

#hide right and top axes
ax1.spines['top'].set_visible(False)
ax1.spines['right'].set_visible(False)
ax1.spines['bottom'].set_position(('outward', 10))
ax1.spines['left'].set_position(('outward', 10))
ax1.yaxis.set_ticks_position('left')
ax1.xaxis.set_ticks_position('bottom')

#set labels
plt.xlabel(r'voltage ($\mu$V)')
plt.ylabel('luminescence')

#add legend
leg = plt.legend(loc="upper left")
leg.draw_frame(False)

#make inset
ax_inset = plt.axes((0.62, 0.22, 0.25, 0.25))
plt.hist(residuals, fc='w')
plt.xticks([-0.8, 0., 0.8], size=6)
plt.yticks([0, 20.], size=6)
plt.xlabel("residuals", size=6)
plt.ylabel("count", size=6)


#export to svg
plt.savefig('power_vs_synchrony.svg')
```



CHAPTER

THREE

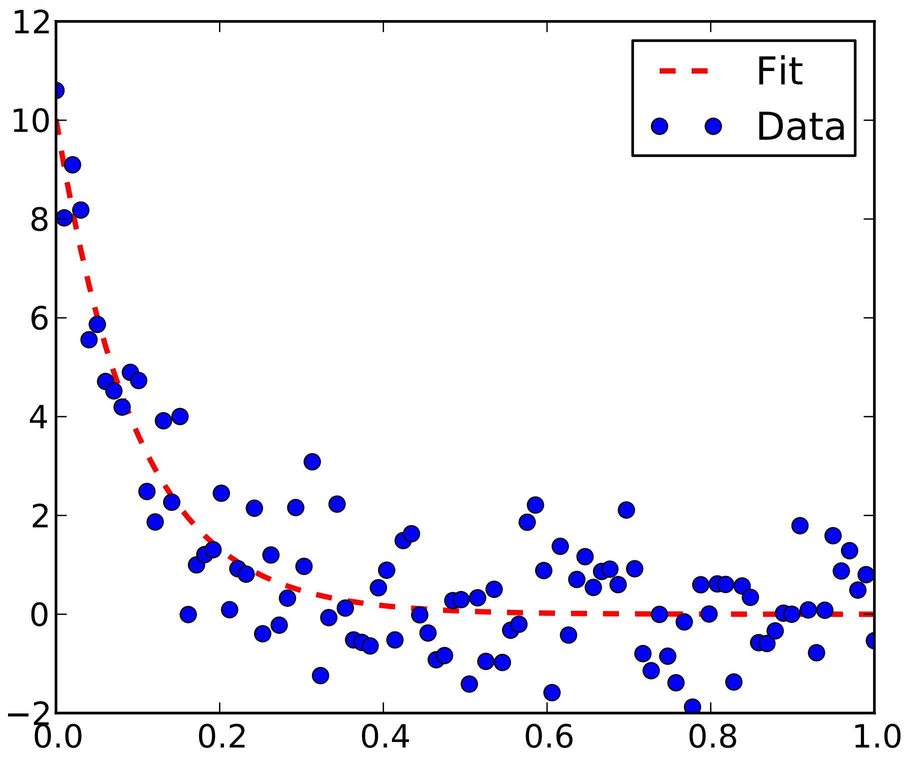
SCIPY

Many of the examples can be found in Scipy Cookbook

3.1 Curve-fitting

SciPy includes a `scipy.optimize.leastsq` function which can be used to perform least squares fits.

```
from scipy import optimize
import numpy as np
import matplotlib.pyplot as plt

def fitfunc(p, x):
    """Function to be fitted.

        Arguments:

            - x - independent variable
            - p - tuple of parameters
    """
    return np.exp(-x/p[0])/p[1]

# simulate some data
n = 100
a, b = 0.1, 0.1
x = np.linspace(0, 1., n)
y = np.exp(-x/a)/b

# add noise
y = y + np.random.randn(n)

# Define an error function (standard form)
errfunc = lambda p, x, y: (y - fitfunc(p, x))

# Initial values for fit parameters
pinit = np.array([2, 2])

out = optimize.leastsq(errfunc, pinit,args=(x, y),full_output = 1)
plt.plot(x, fitfunc(out[0], x),'r--',lw=2,label="Fit")
plt.plot(x, y,'o',label="Data")

plt.legend()
plt.show()
```



3.2 scipy.weave

```
from scipy import weave
import numpy as np

def f_blitz(a,b,c):

    code = r"""

    for(int i=0;i<Na[0];i++) {
    c(i) = a(i)*b(i);
    }

    """

    weave.inline(code,['a','b','c'], type_converters=weave.converters.blitz)

a = 2*np.ones(10)
b = np.arange(0,10)
c = np.zeros(10)

f_blitz(a,b,c)
print c
```

```
[  0.   2.   4.   6.   8.  10.  12.  14.  16.  18.]
```

CHAPTER

FOUR

DATA SERIALIZATION

4.1 pickle

You can store (almost) any Python object in a file:

```
import cPickle
import numpy as np

class my_data(object): pass

data1 = my_data()
data1.data = np.random.randn(100)
data1.params = {"sigma":1., "mu":0. }

fname = "my_data.p"
cPickle.dump(data1, file(fname, 'w'))

data2 = cPickle.load(file(fname))
```

Pickles are a good solution for temporary storage of intermediate analysis results. However, they should not be used for pernament data serialization or data sharing:

- pickles can not be easily exchange between different programming enviroments,
- pickles are unsecure:

  ```
  import pickle
  pickle.loads("cos\nsystem\n(S'ls ~'\ntR.") # This will run: ls ~
  ```

  More information in Why Python Pickle is Insecure

4.2 CSV and Data frame

You can store your data using comma-seperated files (CSV). To find out, how to use it read the documentation of standard library (csv module)

A very nice use case of CSV is a DataFrame class which implements R-like tables. For more information, see Scipy cookbook

4.3 NumPy (record) arrays

See *Reading/writing arrays from/to a file*

4.4 Databases in Python

4.4.1 SQLite

Pysql ite is a wrapper to C library that provides a lightweight disk-based database that doesn't require a separate server process. SQLlite is fitted especially well for developing custom file format, which can replace *ad hoc* binary files. It is platform independent and can be easily shared just by sending the file.

Other application of SQLlite include:

- internal or temporary databases,
- dataset analysis tool,
- sandbox for learning SQL.

Create a new database in a file:

```
>>> import sqlite3
>>> conn = sqlite3.connect('/tmp/example.sqlite')
```

Now you can create a cursor object to execute SQL commands on the database:

```
>>> c = conn.cursor()
```

The first command will create a new table:

```
>>> c = c.execute("""CREATE TABLE morphologies(
...                         name VARCHAR(128) PRIMARY KEY,
...                      age INTEGER,
...                      branch_points INTEGER,
...                      length REAL,
...                      surface_area REAL
...                 )""");
```

You can insert new data into the table:

```
>>> c = c.execute("""insert into morphologies
...                 values ('S1_pyr_A121',5,100,100.,35.14)""")

# Save (commit) the changes
>>> conn.commit()
```

It is easy to insert some data from a python list into the new database:

```
>>> for t in [('S1_basket_A122', 4, 30, 10.1, 45.0),
...           ('CA3_pyr_B3', 8, 3, 10.1, 25.0),
...           ('V1_chand_C4', 1, 5, 1., 4.)
...      ]:
...     c = c.execute('insert into morphologies values (?,?,?,?,?)', t)
...
```

Now you can retrieve data fulfilling specified conditons:

```
>>> c = c.execute('select * from morphologies order by age')
>>> for row in c:
...     print row
...
(u'V1_chand_C4', 1, 5, 1.0, 4.0)
(u'S1_basket_A122', 4, 30, 10.1, 45.0)
(u'S1_pyr_A121', 5, 100, 100.0, 35.140000000000001)
(u'CA3_pyr_B3', 8, 3, 10.1, 25.0)
```

We are done:

```
>>> c.close()
>>> conn.close()
```

4.4.2 STORM

Accessing databases via SQL queries is not very pythonic: it requires learning SQL syntax and the queries are basically strings which are then interpreted by the database engine. That is where Storm comes to rescue. It is an object-relational mapper (ORM) for Python. It allows to map SQL tables and its fields to Python objects. This way you can implement easily Python objects which are semi-automatically serialised into a SQL database. This is a simple usage case:

```
import storm.locals as sls


# first define an object describing the SQL table structure
class Cell(object):
    __storm_table__ = "morphologies"
    name = sls.Unicode(primary=True)
    age = sls.Int()
    branch_points = sls.Int()
    length = sls.Float()
    surface_area = sls.Float()


db_target = "/tmp/example.sqlite"

# next read your database (here using SQLite engine)
db = sls.create_database("sqlite:" + db_target)
store = sls.Store(db)

# now you can add new rows...
new_cell = Cell()
new_cell.name = u"CA1_pyr_A123"
new_cell.age = 3
new_cell.branch_points = 5
new_cell.length = 15
new_cell.surface_area = 19.2

store.add(new_cell)
store.commit()

# ... search the existing ones
```

```
mytest = store.find(Cell, Cell.name == u"CA1_pyr_A123").one()
print mytest.age
```

```
3
```

Lets clean up:

```
>>> import os
>>> os.remove(db_target)
```

4.5 PyTables

PyTables is a package for managing hierarchical datasets and designed to efficiently and easily cope with extremely large amounts of data. PyTables is built on top of the HDF5 library which is a standarised file format with interfaces in C, C++, Fortran, Java.

PyTables is especially suited for efficient storage and access of large datasets. Moreover, it allows to store inhomogenous datasets, combine them with metadata and define hierarchical relationships between them. Last but not least PyTables provides also a very nice NumPy "flavour" and highly-interactive interface to the data (supported by IPython). All of the features makes PyTable the package of choice for many scientific applications.

Given all of the features you might think using PyTables is difficult. On the contrary, first import the module (and numpy if you want to use numpy array):

```
>>> import tables
>>> import numpy as np
```

The data is stored in HDF5 files:

```
>>> h5file = tables.openFile("/tmp/example.h5", mode = "w", title = "Test file")
```

now it is time to organize your data by creating groups:

```
>>> group = h5file.createGroup("/", 'detectors', 'Detector information')
```

Finally, you can generate some data and add the newly generated data to the file:

```
>>> pressure = np.array([25.0, 36.0, 49.0])
>>> h5file.createArray(group, 'pressure', pressure,
...                        "Pressure measurements")
/detectors/pressure (Array(3,)) 'Pressure measurements'
  atom := Float64Atom(shape=(), dflt=0.0)
  maindim := 0
  flavor := 'numpy'
  byteorder := 'little'
  chunkshape := None
```

Do not forget to flush!!

```
>>> h5file.flush()
```

The access to the data is very easy:

```
>>> h5file.root.detectors.pressure.read()
array([ 25.,  36.,  49.])
```

No file names, fread and binary conversions!

If you are finished, close the file:

```
>>> h5file.close()
```

PyTables allows for much more. If you want to find out about its features and usage, visit the website (or simply ask Francesc).

CHAPTER

FIVE

EXERCISES

Contents

5.1 Solving heat equation

Note: Example taken from “Scipy Cookbook” orginally implemented and written by Prabhu Ramachandran (please do not look at the solution before trying ;)

The example we will consider is a very simple (read, trivial) case of solving the 2D Laplace equation using an iterative finite difference scheme (four point averaging, Gauss-Seidel or Gauss-Jordan). This type of partial differential equation (PDE) describes for example heat distribution or electric potential inside a conductor. The formal specification of the problem is as follows. We are required to solve for some unknown function u(x,y) such that $\nabla u^2 = 0$ with a boundary condition specified. For convenience the domain of interest is considered to be a rectangle and the boundary values at the sides of this rectangle are given.

It can be shown that this problem can be solved using a simple four point averaging scheme as follows. Discretise the domain into an (nx x ny) grid of points. Then the function u can be represented as a 2 dimensional array - u(nx, ny). The values of u along the sides of the rectangle are given. The solution can be obtained by iterating in the following manner.

```
for i in range(1, nx-1):
    for j in range(1, ny-1):
        u[i,j] = ((u[i-1, j] + u[i+1, j])*dy**2 +
                  (u[i, j-1] + u[i, j+1])*dx**2)/(2.0*(dx**2 + dy**2))
```

Where dx and dy are the lengths along the x and y axis of the discretised domain.

5.1.1 Exercises

1. Solve the Laplace equation with boundary condtions of your choice. The code implementing the method can be found on wiki (`laplace.py`). Try to choose interesting boundary condtions (other than constant).

2. Plot the solution as an image and 3D plot using matplotlib. Add necessary colorbars and labels. Try using different colormaps.
3. Optimize the solution using numpy array operations. To do that subclass `LaplaceSolver` and override `LaplaceSolver.timeStep` method.
4. Profile your code. Compare the evaluation time of the solution based on Python for loops and numpy arrays.
5. Check how the evaluation times scale with the grid size: plot the evaluation time as a function of the grid size for the for-loop-based and numpy-based solutions.
6. Repeat steps 4 and 5 for C-based solution implemented using `scipy.weave.inline` (you can download the method from Scipy Coobook page).

5.2 Clustering webspace

Note: Motivated by "Programming Collective inteligence" by Toby Segaran (Chapter 3)

In this exercise you will use data gathered from Internet to group blogs (or other feeds) in common topics. To this end, you will use an algorithm called **hierachical clustering** which assigns a set of observations into subsets (called clusters) so that observations in the same cluster are similar.

Hierarchical clustering starts with an assignment where each of the observations (in this case blogs) form a seperate cluster. Next, when the algorithm proceeds clusters similar to each other (seperated by a small distance) are joined together and new clusters are formed. These new clusters are connected again and the procedure is repeated until only one cluster is left. As a result a hierchy of clusters is obtained which can be presented in a form of tree called **dendogram**.

5.2.1 Exercises

1. Use provided functions (in module `feedgenerator.py`) to download the blog entries from Internet and count the words:
   - `parseblogs(blog_list)` – takes as an argument an iterator with the feed URLs and returns a dictionay whose keys are the titles of the blogs and values contain the word counts. Word counts are stored in another dictionary whose keys are the words found in an given blog and items are the counts.
   - `feedlist.txt` – a file with sample feed URLs (in plain text format)
2. Inspect the dictionary returned by `parseblogs`. Make sure that you understand the data strucutre. If in doubt, please ask!

   Optional: Print the array in a table with column and row headers.
3. Convert the dictionary to an array whose rows represent blogs and columns different words. Please note that some words won't appear in all the blogs, so that the word-count dictionary won't contain all possible keys. In such a case the entries in the array corresponding to such a word are 0 for all the blogs where the word does not appear.
4. Store the resulting array (together with words and blog titles) in a file. You can use plain text output (CSV), binary numpy export (such as `save` or `savez`) or pickle.
5. In a seperate module read the file generated in the last exercise. *Optional*: Choose only those words in the array which appear at least in 10% of the blogs but not more than 90% (avoid using loops).

6. Calculate pairwise distances between all blogs. Here the distances are defined as a correlation coefficient between the word counts corresponding to each blog. *Hint*: use `scipy.spatial.distance.pdist` function. Note that it returns only distances between two different blogs and discards reversed distances (because distance A-B is equal to distance B-A).
7. Cluster the data using `scipy.cluster.hierarchy.linkage`. You can take default options, but if you have some time left you can play with different linkage algorithms and check how they influence the result.
8. Draw a dendrogram of the clustering results (use `scipy.cluster.hierarchy.matplotlib`). Are blogs clustered into groups of common topics? Can you recognize the topics?

5.2.2 Optional exercises

1. If you subscribe feeds in an application which can export a subscription list to an OPML file, use `feedgenerator.getbloglist` function to parse the OPML file. It returns a list of URLs which can be used together with `parseblogs`. Run the cluster algorithm on you own feeds and find common topics.
2. Insted of clustering the blogs, you can also cluster words to find out which ones occur most frequently together.