PHYS4038/MLiS and ASI/MPAGS

Scientific Programming in

Python

mpags-python.github.io

Steven Bamford

University of Nottingham
UK | CHINA | MALAYSIA
1) Only using `scipy.special.yn`, plot the zeroth-order Bessel function of the second kind, and use `fsolve` to find the value of the third root, i.e. the third lowest value of $x$ where $Y_0(x) = 0$.

2) Use `quad` to find the integral of $Y_0(x)$ between $x=0$ and the third root.

3) Use `scipy.stats.norm.rvs` to create 100 samples from a Normal distribution for some mean and sigma. Then use `plt.hist` to create a 10-bin histogram of the samples (see the return values). Determine the centre of each bin.

4) Create a function $f((m, s), a, x, y)$ which returns the sum of the squared residuals between the values in $y$ and a Gaussian with mean $m$, sigma $s$ and amplitude $a$, evaluated at $x$.

5) Use function you created in (5) with `scipy.optimize.minimize` to fit a Gaussian to the histogram created in (4). Plot and compare with `scipy.stats.norm.fit`.

Any questions?

- skype (spbamford): [https://join.skype.com/KpW5oCLNNiJt](https://join.skype.com/KpW5oCLNNiJt)
- email steven.bamford@nottingham.ac.uk
An introduction to scientific programming with Python

Session 6:
Data handling
Databases

• Python has tools for accessing most (all?) databases
  • e.g. MySQL, SQLite, MongoDB, Postgres, …

• Allow one to work with huge datasets
• Data can be at remote locations
• Robust and fast

• May require knowledge of DB-specific language
• But often provide Pythonic interface
Databases

• SQLite
  • Lightweight
  • No server
  • Just uses files (convenient, but less powerful)
  • Standard python module: sqlite3
Databases

- MySQL
  - Widely used
  - Need MySQL server installed
  - Official: mysql-connector-python
  - mysqlclient, pymysql, SQLAlchemy
Databases

- MongoDB
  - NoSQL database
  - Need Mongo database server
  - Official: pymongo
Databases

- Python has tools for accessing most (all?) databases
  - e.g. MySQL, SQLite, MongoDB, Postgres, ...

- Allow one to work with huge datasets
- Data can be at remote locations
- Fast random read and write
- Atomic transactions
- Concurrent connections
Databases

• **DB pros and cons**

  • Allow one to work with huge datasets
  • Data can be at remote locations
  • Fast random read and write
  • Concurrent, atomic transactions

• However, most databases are designed for webserver use
  • typically not optimised for data analysis
  • write once, multiple sequential reads
Python Data Analysis Library
  • http://pandas.pydata.org

Easy-to-use data structures
  • DataFrame (more friendly recarray)
  • Handles missing data (more friendly masked array)
  • read and write various data formats
  • data-alignment
    • tries to be helpful, though not always intuitive
  • Easy to combine data tables
  • Surprisingly fast!

Notebook demo…
# Arrays implement the Numpy API
```python
import dask.array as da
x = da.random.random(size=(10000, 10000), chunks=(1000, 1000))
x + x.T - x.mean(axis=0)
```

# Dataframes implement the Pandas API
```python
import dask.dataframe as dd
df = dd.read_csv('s3://.../2018-**.csv')
df.groupby(df.account_id).balance.sum()
```

# Dask-ML implements the Scikit-Learn API
```python
from dask_ml.linear_model \
import LogisticRegression
lr = LogisticRegression()
lr.fit(train, test)
```
• typically for dealing with very large datasets
• distributed computing on a cluster
• need to setup infrastructure
PyTables / h5py

- [http://pytables.github.io](http://pytables.github.io)
- For creating, storing and analysing datasets
  - from simple, small tables to complex, huge datasets
  - standard HDF5 file format
  - incredibly fast – even faster with indexing
  - uses on the fly block compression
  - designed for modern systems
    - fast multi-code CPU; large, slow memory
  - "in-kernel" – data and algorithm are sent to CPU in optimal way
  - "out-of-core" – avoids loading whole dataset into memory
PyTables / h5py

- Can store many things in one HDF5 file (like FITS)
- Tree structure
- Everything in a group (starting with root group, '/')
- Data stored in leaves
- Arrays (e.g. n-dimensional images)

```python
>>> from tables import *

>>> h5file = openFile("test.h5", mode = "w")
>>> x = h5file.createArray("/", "x", arange(1000))
>>> y = h5file.createArray("/", "y", sqrt(arange(1000)))
>>> h5file.close()
```
PyTables

• Tables (columns with different formats) – better to use Pandas!
  • described by a class
  • accessed by a row iterator

```python
>>> class MyTable(IsDescription):
    z = Float32Col()
>>> table = h5file.createTable('/', 'mytable', MyTable)
>>> row = table.row
>>> for i in xrange(1000):
    row['z'] = i**(3.0/2.0)
    row.append()
>>> table.flush()
>>> z = table.cols.z
```
• **Expr** enables in-kernel & out-of-core operations

```python
>>> r = h5file.createArray("/", "r", np.zeros(1000))
>>> xyz = Expr("x*y*z")
>>> xyz.setOutput(r)
>>> xyz.eval()
/r (Array(1000,)) 
    atom := Float64Atom(shape=(), dflt=0.0)
    maindim := 0
    flavor := 'numpy'
    byteorder := 'little'
    chunkshape := None
>>> r.read(0, 10)
array([  0.        ,   1.        ,   7.99999986,   26.9999989 ,
           64.        ,  124.99999917,  216.00000085,  343.00001259,
          511.99999124,   729.        ])```
• **where** enables in-kernel selections

```python
>>> r_bigish = [row['z'] for row in table.where('(z > 1000) & (z <= 2000)')]

>>> for big in table.where('z > 10000;'):
...     print('A big z is {}').format(big['z'])
```

• There is also a **where** in **Expr**
Multiprocessing

- Python includes modules for writing "parallel" programs:
  - threaded – limited by the Global Interpreter Lock
  - multiprocessing – generally more useful

```python
from multiprocessing import Pool

def f(x):
    return x**x

pool = Pool(processes=4)  # start 4 worker processes

z = range(10)
print(pool.map(f, z))  # apply f to each element of z in parallel
```
from multiprocessing import Process
from time import sleep

def f(name):
    print('Hello {}, I am going to sleep now'.format(name))
    sleep(3)
    print('OK, finished sleeping')

if __name__ == '__main__':
    p = Process(target=f, args=(lock, 'Steven'))
    p.start()  # start additional process
    sleep(1)   # carry on doing stuff
    print 'Wow, how lazy is that function!
    p.join()   # wait for process to complete

$ python thinking.py
Hello Steven, I am going to sleep now
Wow, how lazy is that function!
OK, finished sleeping

(Really, should use a lock to avoid writing output to screen at same time)
Coursework submission

- Submission and feedback via your GitHub repository
- Mandatory for MLiS, optional for MPAGS
- Create a branch called sub2
- Should contain your draft code
Questions (especially about coursework code)

Any questions?

• shout and wave
• skype (spbamford)
  • https://join.skype.com/KpW5oCLNNiJt
• email steven.bamford@nottingham.ac.uk