Martti Louhivuori

Python in High-Performance Computing

CSC – IT Center for Science Ltd, Finland
April 21-22, 2016 @ University of Oslo

```
import sys, os
try:
    from Bio.PDB import PDBParser
    __biopython_installed__ = True
except ImportError:
    __biopython_installed__ = False

__default_bfactor__ = 0.0     # default B-factor
__default_occupancy__ = 1.0   # default occupancy level
__default_segid__ = ''        # empty segment ID

class EOF(Exception):
    def __init__(self): pass

class FileCrawler:
    """
    Crawl through a file reading back and forth without loading
    anything to memory.
    """
    def __init__(self, filename):
        try:
            self.__fp__ = open(filename)
        except IOError:
            raise ValueError, "Couldn't open file '%s' for reading." % filename
        self.tell = self.__fp__.tell
        self.seek = self.__fp__.seek
    def prevline(self):
        try:
            self.prev()
```

EXERCISE ASSIGNMENTS

Practicalities

Computing servers

We will be using the computing cluster Abel for the exercises. Temporary user accounts will be handed out. Follow the instructions given at the lecture on how to log in to Abel.

Python is available on Abel as a loadable module that includes all the extra libraries (numpy, mpi4py etc.) needed in the course. Use the **module** command to load it for use:

```
% module load python2
```

If possible, it is recommended to start an interactive session, but jobs can also be individually submitted to the queue.

General exercise instructions

Simple exercises can be carried out directly using the interactive interpreter (if in an interactive session). For more complex ones it is recommended to write the program into a **.py** file. Still, it may be useful to keep an interactive interpreter open for testing!

Directory **exercises/** contains input data and skeleton codes that can be used as starting points for the exercises. Example solutions are provided in the **answers/** directory.

Some exercises contain references to functions/modules which are not addressed in actual lectures. In these cases Python's interactive help (and google) are useful, e.g.

```
>>> help(numpy)
```

It is not necessary to complete all the exercises, instead you may leave some for further study at home. Also, some Bonus exercises are provided in the end of exercise sheet.

Exercise 1: NumPy semantics

1. Investigate the behavior of the statements below by looking at the values of the arrays **a** and **b** after assignments:

```
a = np.arange(5)
b = a
b[2] = -1
b = a[:]
b[1] = -1
b = a.copy()
b[0] = -1
```

2. Generate a 1D NumPy array containing all numbers from -2.0 to 2.0 with a spacing of 0.2. Use optional start and step arguments of **np.arange()** function.

   Generate another 1D NumPy array containing 11 equally spaced values between 0.5 and 1.5. Extract every second element of the array.

3. Create a 4x4 array with arbitrary values.

   Extract every element from the second row.

   Extract every element from the third column.

   Assign a value of 0.21 to upper left 2x2 subarray.

4. Read an 2D array from the file `contour.dat`. You can use the function **np.loadtxt()**:

```
data = np.loadtxt('contour.dat')
```

Split the array into four subblocks (upper left, upper right, lower left, lower right) using **np.split()**. Construct then the full array again with **np.concatenate()**. You can visualize the various arrays with matplotlib's **imshow()**, i.e.

```
import pylab
pylab.ion()
pylab.imshow(data)
```

Exercise 2: Vectorisation

1. Derivatives can be calculated numerically with the finite-difference method as:

$$f'(x_i) = \frac{f(x_i + \Delta x) - f(x_i - \Delta x)}{2\,\Delta x}$$

   Construct 1D Numpy array containing the values of x_i in the interval $[0,\pi/2]$ with spacing $\Delta x=0.1$. Evaluate numerically the derivative of **sin** in this interval (excluding the end points) using the above formula. Try to avoid **for** loops. Compare the result to function **cos** in the same interval.

2. A simple method for evaluating integrals numerically is by the middle Riemann sum

$$S = \sum_{i=1}^{n} f(x'_i)\Delta x$$

   with $x'_i = (x_i + x_{i-1})/2$. Use the same interval as in the first exercise and investigate how much the Riemann sum of **sin** differs from 1.0. Avoid **for** loops. Investigate also how the results changes with the choice of Δx.

   You may use matplotlib for investigating the data

```
import pylab
pylab.ion()
pylab.plot(x, y)
```

Exercise 3: NumPy tools

1. File `xy_coordinates.dat` contains a list of (x,y) value pairs. Read the data with **numpy.loadtxt**() and fit a second order polynomial to data using **numpy.polyfit**().

2. Generate a 10x10 array whose elements are uniformly distributed random numbers using **numpy.random** module.

   Calculate the mean and standard deviation of the array using **numpy.mean**() and **numpy.std**().

   Choose some other randvom distribution and calculate its mean and standard deviation.

3. Construct two symmetric 2x2 matrices **A** and **B**.
   (hint: a symmetric matrix can be constructed easily as $\mathbf{A}_{sym} = \mathbf{A} + \mathbf{A}^T$)

   Calculate the matrix product **C**=**A***B** using **numpy.dot**().

   Calculate the eigenvalues of matrix **C** with **numpy.linalg.eigvals**().

Exercise 4: Simple plotting

1. Plot to the same graph **sin** and **cos** functions in the interval [-π/2, -π/2]. Use Θ as x-label and insert also legends. Save the figure in .png format.
2. The file `csc_usage.dat` contains the usage of CSC servers by different disciplines. Plot a pie chart about the resource usage.
3. The file `contour.dat` contains cross section of electron density of benzene molecule as a 2D data. Make a contour plot of the data. Try both contour lines and filled contours (matplotlib functions **contour** and **contourf**). Use **numpy.loadtxt** for reading the data.

Exercise 5: Parallel computing with MPI

1. Create a parallel Python program which prints out the number of processes and the rank of each process.
2. Read the Fasta sequence of Zika virus (PDB: 5IRE) and send the sequence for chain C to another process. You can use the simple parser provided in `fasta.py` for reading the sequence from the file `5ire.fasta.txt`.
3. Read the atom coordinates of the Zika virus (`5ire.pdb`) using the simple parser provided in `pdb.py`. Calculate the center of coordinates in parallel by distributing the coordinates to multiple processes that each calculate their part sending back the results to rank 0.
   Note: remember to weight the sub-results with the correct number of atoms!
4. Continue the previous exercise by shifting origo to the center of coordinates and then broadcasting the final coordinates to all processes.

Exercise 6: Multiprocessing

1. Calculate the square of all numbers 1..10 using a separate Process to calculate each number. Use a shared memory Array to store the results. Remember to lock the shared array before manipulating it.

2. Repeat exercise 6.1, but now using a data manager instead of shared memory.

3. Read the Fasta sequence of Zika virus (PDB: 5IRE) and send the sequence for chain C to another process using a pipe. Calculate the frequency of Trypthophan (W) in the chain and send back the result to the master process.

   You can use the simple parser provided in `fasta.py` for reading the sequence from `5ire.fasta.txt`.

4. Read the atom coordinates of the Zika virus (`5ire.pdb`) using the simple parser provided in `pdb.py`. Calculate the center of coordinates in parallel by distributing the coordinates in chunks of hundred to a queue that is processed by multiple processes. Store the results for each chunk in a separate queue.

   Combine the results in the master process and figure out how to deal with any coordinates left out of the neat chunks of hundred coordinates each.

5. Repeat exercise 6.4, but now using a pool of workers instead of queues. Use asynchronous mode to maximise throughput.

Exercise 7: Extending Python with C

1. Compile the ”Hello World” extension contained in the file `hello.c` into a shared library `hello.so`. (Use the provided script `include_paths.py` for finding out proper –I options)

   Import the extension in Python and execute the **world** function.

2. Create a C-extension for evaluating second derivative with finite differences. Input is provided by one dimensional NumPy array.
   The core of the C-extension is provided in the file `fidi.c`

   Compare the performance of a pure NumPy version (`pyfidi.py`).

BONUS EXERCISES

Exercise B1: Game of Life

1. Game of life is a cellular automaton devised by John Conway in 70's:
   http://en.wikipedia.org/wiki/Conway's_Game_of_Life

   The game consists of two dimensional orthogonal grid of cells. Cells are in two possible states, **alive** or **dead**. Each cell interacts with its eight neighbours, and at each time step the following transitions occur:

   - Any live cell with fewer than two live neighbours dies, as if caused by underpopulation

   - Any live cell with more than three live neighbours dies, as if by overcrowding

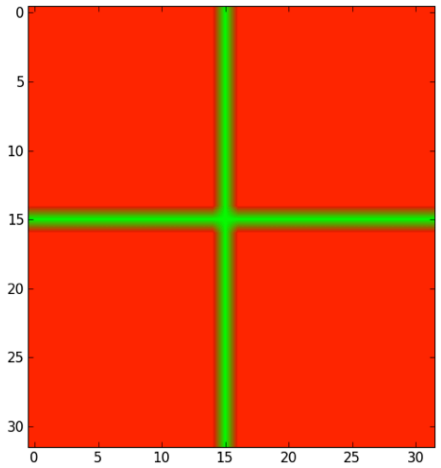
   - Any live cell with two or three live neighbours lives on to the next generation

   - Any dead cell with exactly three live neighbours becomes a live cell

   The initial pattern constitutes the seed of the system, and the system is left to evolve according to rules. Deads and births happen simultaneously.

Exercise B1: Game of Life (cont.)

Implement the Game of Life using Numpy, and visualize the evolution with Matplotlib (e.g. **imshow**). Try first 32x32 square grid and cross-shaped initial pattern:



Try also other grids and initial patterns (e.g. random pattern). Try to avoid **for** loops.

Exercise B2: Parallel Game of Life

1. Use MPI to parallelise the Game of Life from exercise B1 by distributing the grid along one dimension to different processors.
   Hint: you need to have ghost layers to store data from adjacent processors