Sebastian von Alfthan
Jussi Enkovaara
Martti Louhivuori

Exercises for Python in HPC

January 25-27, 2016
PRACE Advanced Training Centre
CSC – IT Center for Science Ltd, Finland

```
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 use either the classroom computers or CSC's computing servers (taito, sisu) for the exercises. To log in to one of the computing servers (e.g. Taito), use the provided tnrgXX username and password, e.g.

```
% ssh -X trng10@vtaito.csc.fi
```

For editing files you can use e.g. nano editor:

```
nano example.py
```

Also other popular editors (vim, emacs) are available.

Python environment at CSC

Python is available on all computing servers (taito, sisu). Use the **module** command to load it for use:

```
% module load python (Sisu) or module load python-env (Taito)
```

The default version is 2.6 and has the following modules available:

- Numpy
- Scipy
- Matplotlib
- Mpi4py

The classroom computers have a similar environment.

General exercise instructions

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

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: Playing with the interactive interpreter

1. Start the interpreter by typing “python” on the command prompt
2. Try to execute some simple statements and expressions e.g.

```
print “Hello! ”
1j**2
1 / 2
my_tuple = (1, 2, 3)
my_tuple[0] = 1
2.3**4.5
```

3. Many Python modules and functions provide interactive help with a help command. Try to execute the following commands

```
import math
help(math)
help(math.sin)
```

Exercise 2: Python syntax and code structure:

1. Are the following valid statements?

```
names = ['Antti', 'Jussi']
x2y = 22
3time = 33
my-name = “Jussi”
next = my-name
open = 13
in = “first”
```

2. Are the following pieces of code valid Python?

ex2_1.py

```
numbers = [4, 5, 6, 9, 11]
sum = 0
for n in numbers:
    sum += n
  print "Sum is now", sum
```

ex2_2.py

```
x = 11
test(x)

def test(a):
    if a < 0:
      print "negative number"
```

Exercise 3: Working with lists and dictionaries

1. Create a list with the following fruit names:
   “pineapple”, “strawberry”, “banana”

   Append “orange” to the list. List objects have a **sort()** function, use that for sorting the list alphabetically (e.g. fruits.sort()). What is now the first item of the list?

   Remove the first item from the list.

2. Create a list of integers up to 10 using the **range()** function.

   Use slicing to extract first all the even numbers and then all the odd numbers.

   Next, extract the last three odd integers of the original list.

   Combine the even and odd integer lists to reproduce the original list of all integers up to 10.

3. Create a two dimensional list of (x,y) value pairs, i.e. arbitrary long list whose elements are two element lists.

4. Create a dictionary whose keys are the fruits “pineapple”, “strawberry”, and “banana”. As values use numbers representing e.g. prices.

   Add “orange” to the dictionary and then remove “banana” from the dictionary. Investigate the contents of dictionary and pay attention to the order of key-value pairs.

Exercise 4: Control structures

1. Write a **for** loop which iterates over the list of fruit names and prints out the name.

   Write a **for** loop which determines the squares of the odd integers up to 10. Use the **range()** function

   Write a **for** loop which sums up to the prices of the fruits of the Exercise 3.4.

2. Fibonacci numbers are a sequence of integers defined by the recurrence relation

   F[n] = F[n-1] + F[n-2]

   with the initial values F[0]=0, F[1]=1. Create a list of Fibonacci numbers F[n] < 100 using a **while** loop.

3. Write a control structure which checks whether an integer is negative, zero, or belongs to the prime numbers 3,5,7,11,17 and perform e.g. corresponding **print** statement.

   Keyword **in** can be used for checking whether a value belongs to a list:

```
>>> 2 in [1,2,3,4]
True
```

4. Go back to the two dimensional list of (x,y) pairs of Exercise 3.3. Sort the list according to y values. (Hint: you may need to create a temporary list).

   Create a new list containing only the sorted y values.

   Next, create a new list consisting of sums of the (x,y) pairs

   Finally, create a new list consisting of sums of the (x,y) pairs where both x and y are positive.

5. Take the fruit price dictionary of Ex. 3.4. Try to access fruit that is not in the dictionary, e.g. **print fruits['kiwi']**. What exception do you get?

   Try to catch the exception and handle it e.g. with your own print statement. Hint: for this exercise it is better to use a .py file and not the interactive interpreter.

6. Function **raw_input()** can be used for obtaining input from terminal. Write an infinite **while** loop which asks user for input and prints it out. Quit the loop when user enters **q**.

   Hit CTRL-D when asked for input, what happens? Try to handle the corresponding exception.

Exercise 5: Modules and functions

1. Standard Python module **sys** has list **argv** which contains the command line arguments to he script.

   ex5_1.py

   ```
   import sys
   print sys.argv
   ```

   Investigate the content of **sys.argv** with different command line arguments, *e.g.*
   "`python ex5_1.py 2.2`", "`python ex5_1.py 1.2  foo`", …

2. Write a function that calculates the area of a circle based on the radius given as command line argument.

   Note that **sys.argv** contains the arguments as strings, use explicit type conversion with **float()** in order to obtain floating point numbers.

3. Write a function which calculates the arithmetic mean from a list of numbers.

4. Write a function that converts a polar coordinate representation (r, ϕ) into cartesian representation (x,y). Write also a function which does the reverse transformation. The important formulas are:
   $x = r \cos(\phi)$ $y = r \sin(\phi)$ $\phi = \operatorname{atan}(y/x)$ $r2 = x2 + y2$

   Use the **math** module.

   Implement the coordinate transformation functions in their own module **polar.py** and use them from a main script.

Exercise 6: Working with files

1. The file “exercise6_1.dat” contains list of (x, y) value pairs. Read the values into two lists **x** and **y**.
2. The file “exercise6_2.txt” contains output from a simulation run where the geometry of CO molecule is optimized. One part of the output is the free energy during geometry optimization. Free energies are in the lines of type:

   Free Energy: -16.47537

   Read the free energies from the file and print out how much each energy differs from the final value.
3. The file “CH4.pdb” contains the coordinates of methane molecule in a PDB format. The file consists of header followed by record lines which contain the following fields:

   record name(=ATOM), atom serial number, atom name, x-,y-,z-coordinates, occupancy and temperature factor.

   Convert the file into XYZ format: first line contains the number of atoms, second line is title string, and the following lines contain the atomic symbols and x-, y-, z-coordinates, all separated by white space. Write the coordinates with 6 decimals.

Exercise 7: Steering a simulation with Python

1. Write a Python program which does loop over different values of x (use e.g. a self-specified list). At each iteration, write an “input file” of the form (here x=4.5):

input.dat

```
#!/bin/bash
x=4.500000
res=`echo "$x^2" | bc`
echo "Power 2 of $x is $res"
```

2. As a “simulation program” use **bash** and execute the input file at each iteration with the help of **os** module:

```
>>> os.system(‘bash input.dat > output.dat')
```

3. Read (x,y) pairs from the “output file” **output.dat** and save them for later processing.

Exercise 8: Object oriented programming and classes

1. Define a class for storing information about an element. Store the following information:
   name, symbol, atomic number, atomic weight

   Use e.g. the following data for creating instances of a few elements:

Element	symbol	atomic number	atomic weight
Hydrogen	H	1	1.01
Iron	Fe	26	55.85
Silver	Ag	47	107.87

   Store the instances as values of dictionary whose keys are the element names.

2. Define a class Circle that requires the radius as input data when creating an instance. Define methods for obtaining the area and circumference of the circle.

3. Define a class Sphere, inheriting from Circle. Define methods for obtaining the volume and area of the sphere.

   Raise an exception if one tries to obtain circumference of a sphere.

Exercise 9: Simple NumPy usage

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 numbers from -2 to 2 in increments 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 **exercise9_4.dat.** You can use the function **np.loadtxt()**:

```
data = np.loadtxt('exercise9_4.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.imshow(data)
```

Exercise 10: Numerical computations with NumPy

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.

Exercise 11: NumPy tools

1. File "exercise11_1.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 random 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 12: 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_data.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 13: Using SciPy

1. The integrate module (**scipy.integrate**) contains tools for numerical integration. Use the module for evaluating the integrals

   $$\int_{1.0}^{3.5}(1+x^2)dx \qquad \text{and} \qquad \int_0^{\infty} e^{-x^2}\,dx$$

   Try to give the function both as samples (use **simps**) and as a function object (use **quad**).

2. Try to find the minimum of the function

   $$f(x) = (x+4)(x+1)(x-1)(x-3)$$

   using the **scipy.optimize** module. Try e.g. downhill simplex algorithm (**fmin**) and simulated annealing (**anneal**) with different initial guesses (try first 4 and -4).

Exercise 14: 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)

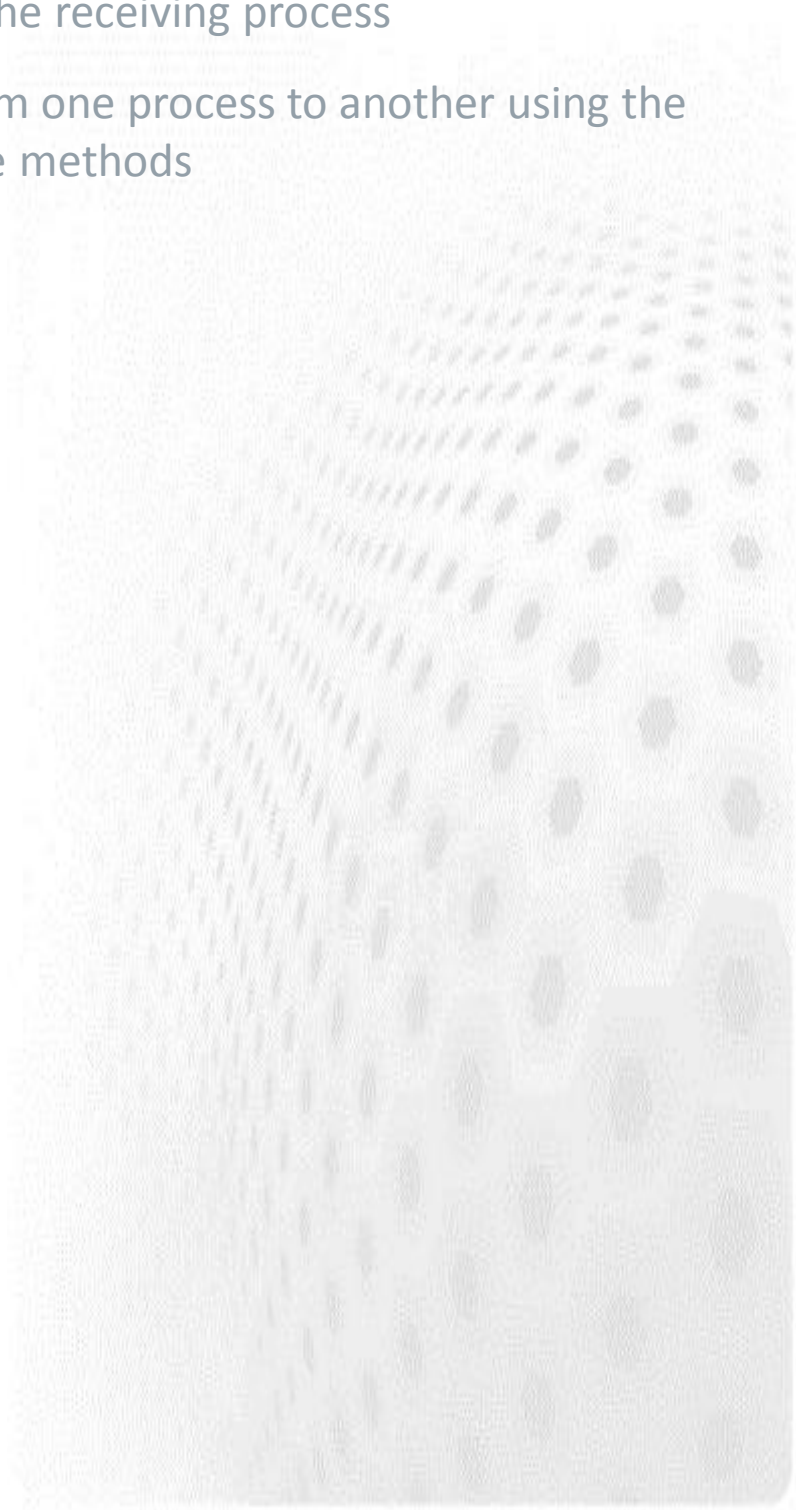
   Import the extension in Python and execute the **world** function.

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**

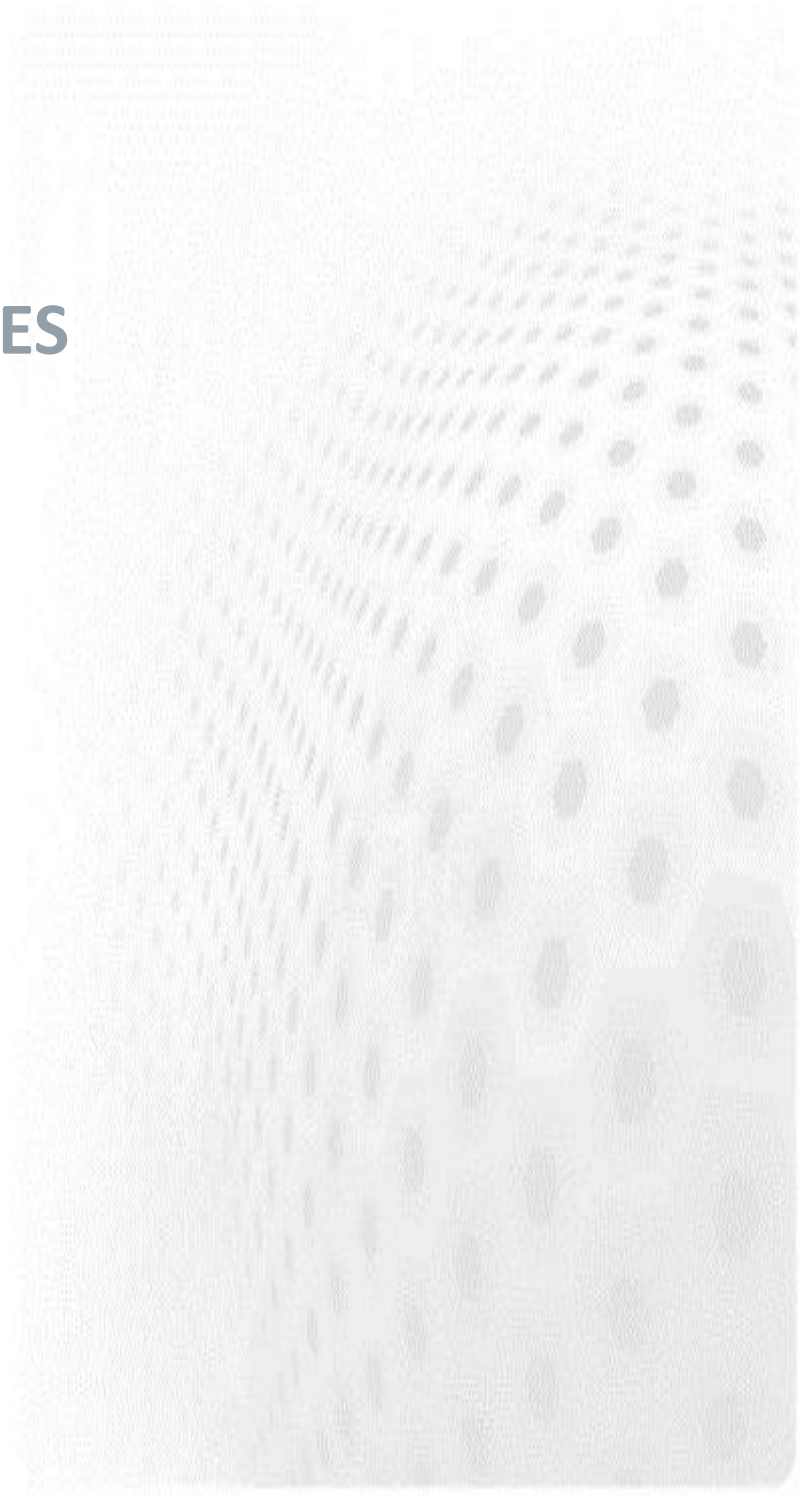
   Compare the performance of pure NumPy version (fidi.py or your implementation in the solution of Poisson equation).

Exercise 15: Parallel computing with Python

1. Create a parallel Python program which prints out the number of processes and rank of each process
2. Send a dictionary from one process to another and print out part of the contents in the receiving process
3. Send a NumPy array from one process to another using the uppercase Send/Receive methods



BONUS EXERCISES



Exercise B1: Working with lists and dictionaries

1. Create a new “fruits” dictionary where the values are also dictionaries containing key-value pairs for color and weight, e.g.

```
>>> fruits['apple'] = {'color':'green',
                       'weight': 120}
```

   Change the color of “apple” from green to red.

2. It is often useful idiom to create empty lists or dictionaries and add contents little by little.

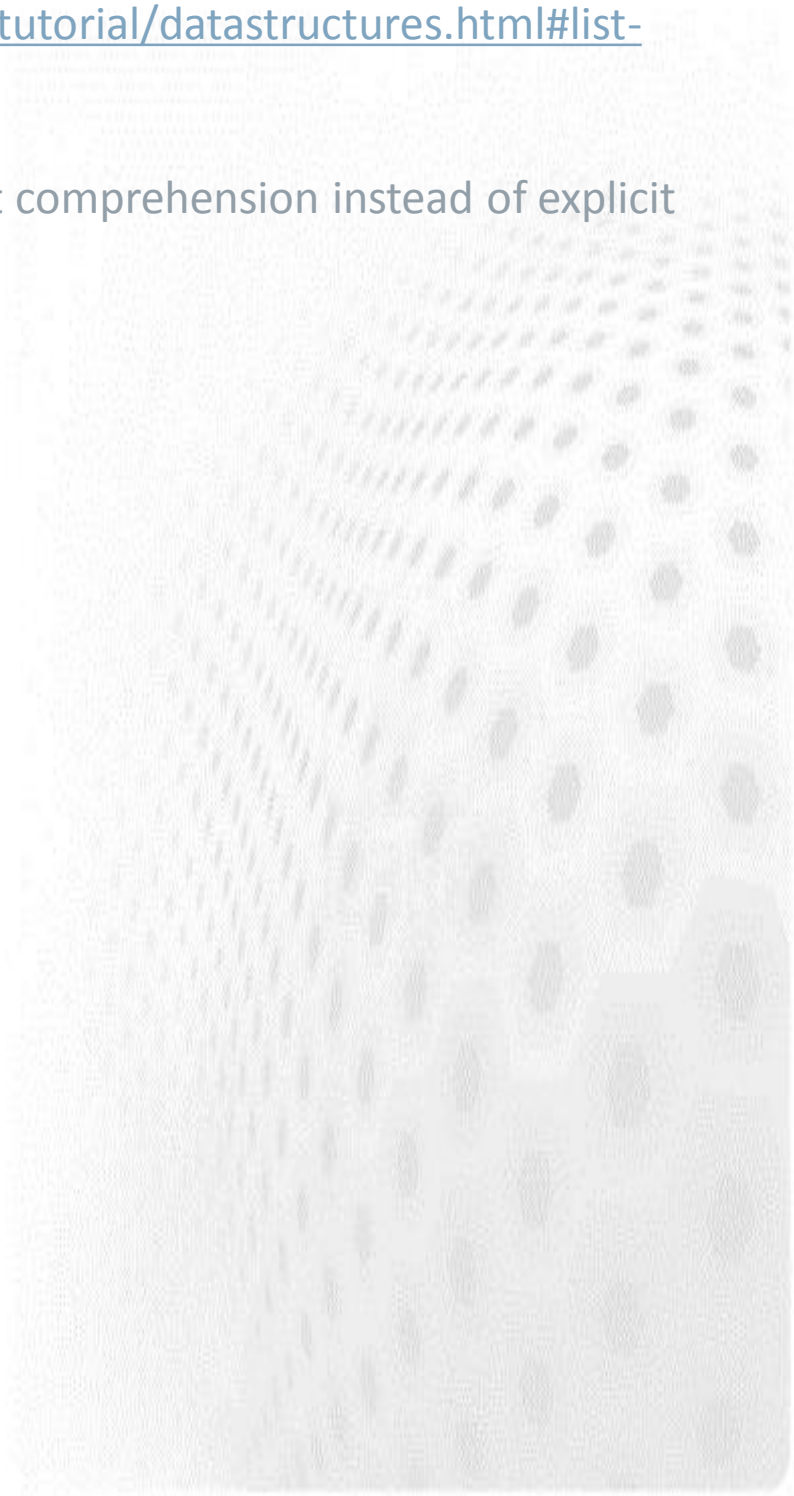
   Create first an empty dictionary for a mid-term grades of students. Then, add a key-value pairs where the keys are student names and the values are empty lists. Finally, add values to the lists and investigate the contents of the dictionary.

Exercise B2: More on control structures

1. List comprehension is useful Python idiom which can be sometimes used instead of explicit **for** loops

   Familiarize yourself with list comprehensions e.g. at http://docs.python.org/tutorial/datastructures.html#list-comprehensions

   Do Exercise 4.4 using list comprehension instead of explicit for loops.



Exercise B3: Solving Poisson equation with Python

1. Poisson equation in one dimension is

$$\frac{d^2\phi}{dx^2} = -\rho$$

where ϕ is the potential and ρ is the source. The equation can be discretized on uniform grid, and the second derivative can be approximated by the finite differences as

$$\frac{d^2\phi(x_i)}{dx^2} = \frac{\phi(x_{i+1}) - 2 * \phi(x_i) + \phi(x_{i-1})}{h^2}$$

where h is the spacing between grid points.

Using the finite difference representation, the Poisson equation can be written as

$$\phi(x_{i+1}) - 2 * \phi(x_i) + \phi(x_{i-1}) = -h^2\rho(x_i)$$

The potential can be calculated iteratively by making an initial guess and then solving the above equation for ϕ(xi) repeatedly until the differences between two successive iterations are small (this is the so called Jacobi method).

Use

$$\rho(x) = 2x^2 - 1$$

as a source together with boundary conditions ϕ(0)=0 and ϕ(1)=0 and solve the Poisson equation in the interval [0,1].

Exercise B3: Solving Poisson equation with Python

Compare the numerical solution to the analytic solution:

$$\phi(x) = \frac{-x}{3} + \frac{x^2}{2} - \frac{x^4}{6}$$

Note! In reality, pure Python is not the best solution for this kind of problems.

Exercise B4: Working with files

1. The file “exercise_b4_1.txt” contains a short piece of text. Determine the frequency of words in the file, i.e. how many times each word appears. Print out the ten most frequent words

   Read the file line by line and use the **split()** function for separating a line into words.

   The frequencies are stored most conveniently into a dictionary. The dictionary method **setdefault** can be useful here. For sorting, convert the dictionary into a list of (key, value) pairs with the **items()** function:

```
>>> words = {"foo" : 1, "bar" : 2}
>>> words.items()
[('foo', 1), ('bar', 2)]
```

Exercise B5: Object-oriented programming with Python

1. Define a class for three dimensional (x,y,z) vector. use operator overloading for implementing element-wise addition and multiplication with + and * operators.

   Define functions calculating dot and cross products of two vectors

2. The file “solvents.dat” contains some information about different solvents. Define a class for storing the information. Define the following methods for the class:

   read - given a file object, reads a single line from the file and parses the information for solvent

   mass - given volume, return the mass of solvent

   Construct a list of solvents, sort it according to density and print out the names.

Exercise B6: Numerical computations with NumPy

1. Bonus exercise B3 introduced the 1D Poisson equation. Solve the exercise B3 using NumPy instead of pure Python and do not use any **for** loops. Note: you can work on this exercise even if Ex. B3 was skipped.

2. The Poisson equation of can be solved more efficiently with the conjugate gradient method, which is a general method for the solution of linear systems of type:

   $$\mathbf{Ax} = \mathbf{b}.$$

   Interpret the Poisson equation as a linear system and write a function which evaluates the second order derivative (i.e. the matrix – vector product **Ax**). You can assume that the boundary values are zero.

   Solve the Poisson equation with the conjugate gradient method and compare its performance to the Jacobi method.

   See next page for the pseudo-code for the conjugate gradient method.

Conjugate gradient algorithm

$\mathbf{r}_0 := \mathbf{b} - \mathbf{A}\mathbf{x}_0$
$\mathbf{p}_0 := \mathbf{r}_0$
$k := 0$
repeat
  $\alpha_k := \frac{\mathbf{r}_k^{\mathrm{T}}\mathbf{r}_k}{\mathbf{p}_k^{\mathrm{T}}\mathbf{A}\mathbf{p}_k}$
  $\mathbf{x}_{k+1} := \mathbf{x}_k + \alpha_k\mathbf{p}_k$
  $\mathbf{r}_{k+1} := \mathbf{r}_k - \alpha_k\mathbf{A}\mathbf{p}_k$
  if $\mathbf{r}_{k+1}$ is sufficiently small **then** exit loop **end if**
  $\beta_k := \frac{\mathbf{r}_{k+1}^{\mathrm{T}}\mathbf{r}_{k+1}}{\mathbf{r}_k^{\mathrm{T}}\mathbf{r}_k}$
  $\mathbf{p}_{k+1} := \mathbf{r}_{k+1} + \beta_k\mathbf{p}_k$
  $k := k + 1$
end repeat
The result is $\mathbf{x}_{k+1}$

Exercise B7: 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

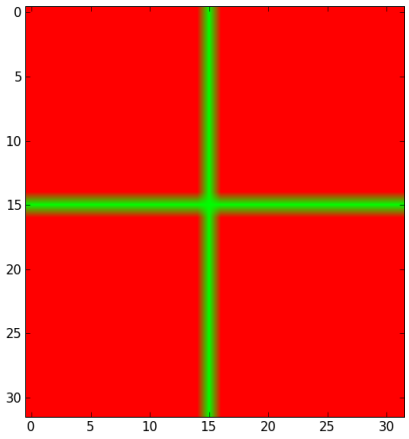
   - Any live cell with two or three live neighbours lives on to the next generation

   - 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 B7: 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 B8: Advanced SciPy and matplotlib

1. Solve the Poisson equation using Scipy. Define a sparse linear operator which evaluates matrix–vector product **Ax** and e.g. Scipy's conjugate gradient solver.
2. The file “atomization_energies.dat” contains atomization energies for a set of molecules, calculated with different approximations. Make a plot where the molecules are in x-axis and different energies in the y-axis. Use the molecule names as tick marks for the x-axis
3. Game of Life can be interpreted also as an convolution problem. Look for the convolution formulation (e.g. with Google) and use SciPy for solving the Game of Life.

   No solution provided! The exercise should be possible to solve with ~10 lines of code.

Exercise B9: Parallel computing with Python

1. Try to parallelize Game of Life with mpi4py by distributing the grid along one dimension to different processors.