Send in your final programs to both Rosa Bulo (r.e.bulo@uu.nl) and Felipe Zapata (f.zapata@vu.nl) as a tar file named with your last name. The tar-file contains a directory for each question. Working together is *not* allowed. Send your solutions in before the end of Saturday December 10.

Question 1

Create one big python script that is able to read in atomic positions from a file, and computes the total energy for a system of particles with those positions. The script-file contains several functions and a main part. A template (*computeLJenergy.py*) is provided in the subdirectory Q1. In all cases, try to optimize performance within the constraints of the assignments.

a. Create a function called `computeDist()` that takes two points in 3D space as argument (in the form of two lists), and returns the distance between the points.

b. Create a function `computeEnergy()` that takes a distance as argument and returns an energy. The function evaluates the mathematical expression for a Lennard-Jones energy in Eq. (1).

$$E_{LJ}(r) = 4\varepsilon\left(\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right) \qquad (1)$$

with an ε value of 1.0 and a σ value of 1.0.

c. Create a function called `getTotalEnergy()` that takes a set of points in 3D space as argument (a list of lists) and computes the sum of the energies of all pairs. This function should call `computeDist` () and `computeEnergy()` under the hood.

d. Create an alternative function to the one above (`getTotalEnergy()`). This function is called `getTotalEnergyFromMatrix()` and takes a square numpy array of floats as argument. The element *ij* of the input array contains the distance between particle *i* and particle *j*. The function then computes the sum of the energies of all pairs. *This function does NOT call any of the above functions.* The input array should contain zeros on the diagonal and in the lower left triangle to avoid double counting.

e. A textfile is supplied (positionfile.txt) with hidden on certain lines the coordinates of helium atoms. Write a script that takes the name of the file as a command line argument. Inside the script, use the shell command `grep` to inside a python script to select the relevant lines from the file, and store the positions in a list of lists. Print the results to the screen.

f. Continue in the main script and compute the total Lennard-Jones energy for the atoms of which you stored the positions, using the function `getTotalEnergy()`. Print the total energy to screen.

g. Continue the main script further, and store all interatomic distances (for the atoms of which you stored the positions) inside a matrix (a numpy array). The matrix should contain a very large number on the diagonal and in the lower left triangle (so that the energy contribution equals zero). Use the function `getTotalEnergyFromMatrix()` from question 1d to compute the total energy. Print the total energy to screen.

h. The supplied textfile (positionfile.txt) contains a jumble of words. Loop over the words and count how many times each word is used. Print the two most frequently used words to screen.

Note: To loop over the key-value pairs of a dictionary in Python3 you can use the dictionary method `items()`.

Question 2

Tensors are very common in scientific computing. They represent multidimensional arrays of arbitrary dimension (or shape): 1-dimension is a vector, 2-dimension a matrix, etc. A fundamental requirement of multidimensional array representations is the possibility to manipulate the array elements. In this exercise you will use object oriented programming to implement methods to operate with tensors.

In the template provided, called *oop_question.py,* you will find the *partial* implementation of the classes: Tensor, Vector, Matrix and Array3D. Representing a general tensor, a 1-dimension array, a 2-dimensional array and a 3-dimensional array, respectively. Each class has 6 methods:

- **__init__(***self, xs, shape***)**. This method takes three arguments: *self*, a list of numbers and a tuple containing the array shape. The elements are simply the numbers to store in the object and the shape is a tuple with the size of each dimension, for example: (25) for 1-dimensional vector, (3, 4) for a matrix and (3, 4, 8) for a tridimensional array.
- ***__str__(self).*** This method formats the tensor in a human readable format. It is called every time that you want to print a Tensor.
- ***map(****self, f****)***. It takes a function *f* and applies it uniformly over the entire data in the tensor. For example, for a matrix:

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} \xrightarrow{map(f)} \begin{bmatrix} f(a_{00}) & f(a_{01}) & f(a_{02}) \\ f(a_{10}) & f(a_{11}) & f(a_{12}) \\ f(a_{20}) & f(a_{21}) & f(a_{22}) \end{bmatrix}$$

- ***reduce(****self, f, axis=0****).*** It takes a function f and accumulates with it along an axis. For example, for a matrix:

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} \xrightarrow{reduce(+,axis=0)} \begin{bmatrix} a_{00}+a_{01}+a_{02} \\ a_{10}+a_{11}+a_{12} \\ a_{20}+a_{21}+a_{22} \end{bmatrix}$$

Where the function f is a binary operator (i.e. it takes two arguments), the function (+) in this case.

Notice that the resulting Tensor dimension has been reduced from 3x3 to 3. The previous application of *reduce* is equivalent to

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} \xrightarrow{apply\ f} \begin{bmatrix} f(a_{00}, f(a_{01}, a_{02})) \\ f(a_{10}, f(a_{11}, a_{12})) \\ f(a_{20}, f(a_{21}, a_{22})) \end{bmatrix}$$

- ***mask(****self, predicate****).*** It takes a special function (called predicate) that takes a value and check whether it fulfills a condition returning a Boolean. Using the predicate it transform all the values in the Tensor to True or False. For example, for a matrix and the (>3) predicate:

$$\begin{bmatrix} 2 & 8 \\ 4 & -5 \end{bmatrix} \xrightarrow{mask(>3)} \begin{bmatrix} False & True \\ True & False \end{bmatrix}$$

- ***filter(****self, predicate****).*** This function also takes a predicate and return always a 1-D dimensional containing the elements for which the predicate is True. Using the previous example:

$$\begin{bmatrix} 2 & 8 \\ 4 & -5 \end{bmatrix} \xrightarrow{filter(>3)} [8\ 4]$$

Your first task is to implement the **__init__** method for Tensor class. The remaining 5 methods should be abstract. Hint: What is the implementation of an abstract method?

The Vector, Matrix and Array3D are *subclasses* of the Tensor class, see the *oop_question.py* file, therefore they should implement the 6 aforementioned methods: *__init__, __str__, map, reduce, mask and filter*. Your main assignment in the present exercise is to implement those 6 methods for the subclasses of Tensor: Vector, Matrix and array3D.

In the *oop_question.py* file there are further examples and explanations about the expected functionality of these methods. First implement the methods for the *Vector* class, then try to reuse some of the code for the implementation of the methods in higher dimensions. Hint: You can use the built-in function map and filter functions from python.

Notes:

- The *filter* method must be implemented using the *mask* method.
- You are expected to use inheritance to implement the **__init__** method for the classes: Vector, Matrix and Array3D.
- *You are* ***not allowed*** *to use* ***Numpy***.