Aims

- practice writing loops and conditionals
- learn how to read from a file
- learn how to work with matrices

Problem 1 - Addresses

The aim here is to practice reading and writing files, as well as using dictionaries and tuples. Write your solutions in a file **sol.py**.

You are given an input file 'addresses.txt' which contains details (names, age, address, job) about several persons. Have a look at the file first and then write the following functions:

1. Write a function **loadFile(filePath)** which takes a string variable filePath, loads the file contents in a dictionary called persDetails and returns the dictionary. In general, a dictonary is a collection of (key, value) pairs. In your case, the key will be a string representing the person's name and the value will be a 3-element tuple (age, address, job). At the end of the script (outside the function), run the following code:

```
persDetails = loadFile('addresses.txt')
print(persDetails)
```

   This should show (not necessary in this order as dictionaries are not ordered):

```
{'Radu': (24, 'Str_Corvin_nr_7', 'programator'), 'Maria': (40, '
  Str_Corvin_nr_7', 'economist'), 'Enescu': (30, 'Str_Muzicienilor_nr_1'
  , 'pianist'), 'Ana': (25, 'Str_Maslinilor_nr_3', 'lingvist'), 'George'
  : (18, 'Str_Grozavesti_nr_7', 'student'), 'Ionel': (20, '
  Str_Bratianu_nr_4', 'consultant')}
```

2. Write a function **getAge(persDetails, name)** which takes the previously loaded dictionary persDetails and the name of a person and returns the corresponding age of that person. For example, getAge(persDetails, 'Ionel') = 20

3. Write a similar function **getAddress(persDetails, name)** which works like getAge but returns the corresponding address of that person. For example, getAddress(persDetails, 'Ionel') = 'Str_Bratianu_nr_4'

4. Write a function **getNamesFromAge(persDetails, age)** which takes the dictionary persDetails and an age (integer) and returns a list of the names of persons that have this age. For example, getNamesFromAge(persDetails, 24) = ['Radu', 'Ana']

5. Write a function **addToDict(persDetails, name, age, address, job)** which takes a dictionary persDetails, a new name, age, address and job and adds them to the dictionary. You might find the function **dict.keys()** useful. The function should return the updated dictionary. Run the following code to check if the result worked:

```
persDetails = addToDict(persDetails, 'Adriana', 25, 'Str_Maniu_nr_3', '
    medic')
# should show entry with Adriana
print(persDetails)
```

6. Write a function **removeFromDict(persDetails, name)** which takes a dictionary persDetals and a name and removes the entry with the corresponding name from the dictionary. Do not use del or pop. Do this by constructing a new dictionary newDict with all the entries apart from the removed one and return it. You can test your code as follows:

```python
persDetails = removeFromDict(persDetails, 'Enescu')
# Should not show Enescu entry
print(persDetails)
```

7. Write a function **writeToFile(persDetails, filePath)** which takes the dictionary and saves it to a file called filePath. Save one dictionary entry on each line. On each line, show the name, age, address and job, separated by a whitespace. Run the function call writeToFile(persDetails, 'new_addresses.txt') and check the file outputs.

8. You can use this code for calling all functions:

```python
persDetails = loadFile('addresses.txt')
print(persDetails)

ageIonel = getAge(persDetails, 'Ionel')
print('ageIonel ', ageIonel)
addressIonel = getAddress(persDetails, 'Ionel')
print('addressIonel ', addressIonel)

names24 = getNamesFromAge(persDetails, 24)
print('names24', names24)
persDetails = addToDict(persDetails, 'Adriana', 25, 'Str_Maniu_nr_3', '
    medic')
# should show entry with Adriana
print(persDetails)

persDetails = removeFromDict(persDetails, 'Enescu')
# Show not show Enescu entry
print(persDetails)

writeToFile(persDetails, 'new_addresses.txt')
```

Problem 2 - Matrices

The aim here is to practice using matrices in Python using numpy. You need to have a working version of numpy to proceed with this problem. We can define a 1-dimensional numpy vector in python as follows:

```python
import numpy as np # place this at the beginning of the file

m = np.zeros(10, float) # vector has 10 entries, all elements are initialised
   to zero
m[0] = 4 # assign the value of 4 to the first element
print(m[0])
```

We can also convert a list of numbers into a (numpy) array as follows:

```python
n = np.array([1,2,3,7,8])
print(n) # show the value of n
```

For declaring a 2-dimensional numpy matrix and accessing an element we can use the following syntax:

```
m = np.zeros((3,3), float) # create a 3x3 matrix where all elements are
    initialised to zero, float is the type of date being stored in the matrix
m[0,1] = 7 # assign the value of 7 to element at position (0,1)
print(m)
```

The following example creates the matrix $\begin{bmatrix} 1 & 2 & 6 \\ 3 & 4 & 7 \end{bmatrix}$ and stores it in variable m:

```
m = np.zeros((3,2), float)
m[0,1] = 1
m[0,2] = 2
m[0,3] = 6
m[1,0] = 3
m[1,1] = 4
m[1,2] = 7
print(m) # try printing the matrix
```

- Following the examples above, create two matrices and store them in variables m1 and m2:

$$m1 = \begin{bmatrix} 1 & 2 & 6 & 9 \\ 3 & 4 & 7 & 1 \end{bmatrix}$$

$$m2 = \begin{bmatrix} 1 & 2 & 3 & 1 \\ 5 & 6 & 7 & 0 \end{bmatrix}$$

- Write a function **printMatrix(m)** that takes a 4x2 matrix m and loops through each element of m and prints it on the screen. Do not use any other higher-level functions such as print(). You need to loop through each element m[i,j] (requires two nested for-loops, one until 4 and the other until 2) and print it, along with an extra space character. At the end of each row, don't forget to print a newline character.

- Write a function **addMatrix(m,n)** that takes two numpy matrices m and n and returns m+n. Do not use the high-level operator '+'. Instead, create a new numpy matrix p and loop over every element p[i,j] of the matrix p (requires two nested for-loops) and assign it the correct value. You can assume m,n have size 4x2. Test your function using the matrixes m1 and m2 defined above. For example:

$$addMatrix(m1, m2) = \begin{bmatrix} 2 & 4 & 9 & 10 \\ 8 & 10 & 14 & 1 \end{bmatrix}$$

- Write a function **transposeMatrix(m)** which computes and returns m^T, the transpose of matrix m. The transpose of a matrix m with dimensions AxB is a matrix m^T with dimensions BxA, such that $m^T[i, j] = m[j, i]$ (elements are flipped around the diagonal). You can assume m has size 4x2. For example, for matrix m1 defined above, it's transpose is:

$$transposeMatrix(m1) = \begin{bmatrix} 1 & 3 \\ 2 & 4 \\ 6 & 7 \\ 9 & 1 \end{bmatrix}$$

- Write a function **slideMatrix(m)** which returns a new matrix n, where all elements are slided to the right by one position. Moreover, the elements on the last column are placed on the first column. For example:

$$slideMatrix(m1) = \begin{bmatrix} 9 & 1 & 2 & 6 \\ 1 & 3 & 4 & 7 \end{bmatrix}$$

- HARDER (you can skip this if too difficult): Given two matrices $M : A \times B$ (i.e. M has A rows and B columns) and $N : B \times C$ we can multiply then them to get $P = MN$, where P is an $A \times C$ matrix defined as

$$P[i,j] = \sum_{k=0}^{B-1} M[i,k] * N[k,j], \forall i \in [0 \ldots (A-1)], j \in [0 \ldots (C-1)]$$

  For example, if we consider $m1 = \begin{bmatrix} 1 & 2 & 6 & 9 \\ 3 & 4 & 7 & 1 \end{bmatrix}$ and $m3 = \begin{bmatrix} 1 & 3 \\ 0 & 4 \\ -2 & 0 \\ 9 & 1 \end{bmatrix}$, then

$$matrixMultiply(m1, m3) = \begin{bmatrix} 70 & 24 \\ -2 & 26 \end{bmatrix}$$

  Write a function **multiplyMatrix(m,n)** which takes as input a matrix 2x4 matrix m, a 4x2 matrix n and returns a new matrix p=m*n, the result of multiplying matrices m and n.

- Modify all the functions above to accept matrices of any shape (so far we only assumed 4x2 matrices). If we are given a matrix m, we can find it's shape using function **m.shape**. We can then use this to create a new matrix n of the same size as

```
shapeM = m.shape
n = np.zeros(shapeM, float) # create a matrix n of the same size as m
```

  Use this strategy to change your previous functions to accept matrices of any shape. Test all your functions for new matrices of different sizes.

Problem 3 - Triangles and Magic Squares

Implement all the problems from the **magic_squares.pdf** specification. The specification has been originally written for Java, but we can easily adapt it for Python (or C++). I have written the function stubs in the file called **magic.py**. Fill in the functions with the required code according to the instructions given in the pdf speccification.