Python for Engineers 0509-1820, Semester 1 2020-2021

Homework 9

OOP, NUMPY & Image Processing

General instructions

- Read the questions **carefully** and make sure your programs work according to the requirements.
- The homework needs to be done individually!
- Read the submission rules on the course web page. All the questions need to be submitted together in the file ex9_012345678.py attached to the homework, after changing the number 012345678 with your ID number (9 digits, including check digit).
- How to write the solution: in this homework, you need to complete the code in the attached outline file.
- **You are not allowed to change the names of the classes, functions, methods and variables that already appear in the attached outline file.**
- **You are not allowed to erase the instructions that appear in the outline file.**
- Some questions are checked automatically. **You thus need to ensure that the output is exactly fits the requirements (even for spaces).**
- Check your code: in order to ensure correctness of your programs and their robustness in the presence of faulty input, for each question run your program with a variety of different inputs, those that are given as examples in the question and additional ones of your choice (check that the output is correct and that the program does not crash).
- **Unless stated otherwise, you can suppose that the input received by the functions is correct.**
- Final submission date: see course web page.

Question 1:

Roman numerals are a numeral system that originated in ancient Rome and remained the usual way of writing numbers throughout Europe well into the Late Middle Ages. Numbers in this system are represented by combinations of letters from the Latin alphabet.

You can have more information here: https://en.wikipedia.org/wiki/Roman_numerals

We want to implement a class in which we can represent the numbers in the roman format and do basic arithmetic operations on them.

Basic roman numerals definition goes as follows:
'I': 1, 'V': 5, 'X': 10, 'L': 50, 'C': 100, 'D': 500, 'M': 1000

1) Write a class called "Roman" which initializes either with an integer which holds the integer value of the numeral or a string which holds the roman numeral.
   The class has 3 attributes:
   Start by implementing the constructor:
   def __init__(self, input_value):
   The argument input_value is either an integer or a roman format string. The constructor initializes the three attributes:

   int_value: stores the integer value (int)

   roman_value: stores the value in the roman format (string)

   is_neg: stores whether the input is negative (less than a zero) or not (bool).

   We have already defined in the template the functions of "get_roman_from_int (self)" and "get_int_from_roman (self)" to covert the int value to roman and save in the attribute and vice-versa.

   You may assume that the input is legal and may only contain an integer or an upper-case string that contains a valid roman format. In addition, assume that the input is not zero, as zero cannot be represented in Roman format.

2) Now we want to implement the functions __repr__ and __str __ to enable a pretty representation of Roman objects. The desired format is:

- The method of __repr__:
  >>>Roman('V')
  V
  >>>Roman(50)
  L
  >>>Roman('XL')
  XL
- The method of __str__
  >>>print(Roman('L'))
  The integer value is 50 and the Roman Numeral is denoted by 'L'
  >>>print(Roman(50))

```
The integer value is 50 and the Roman Numeral is denoted by 'L'
>>>print(Roman('XL'))
The integer value is 40 and the Roman Numeral is denoted by 'XL'
>>> print(Roman('-V'))
The integer value is -5 and the Roman Numeral is denoted by '-V'
>>> print(Roman(-5))
The integer value is -5 and the Roman Numeral is denoted by '-V'
```

Pay attention to the case of negative number that is presented in the last two examples.

3) We would like to support the negation operator. For that purpose, you should implement the __neg__ method. The following example demonstrates negation operator:

```
>>>-Roman(50)
-L
>>>-Roman(V)
-V
```

4) Now we need to write appropriate methods for addition (__add__) and comparison (__lt__ and __gt__) of roman numerals (only left side operations). We should be able to do the operations with 2 roman numerals or a roman numeral and an integer. For e.g.:

Method: __add__

```
>>> Roman(5) + Roman(6)
XI
>>> Roman('V') + 6
XI
>>> Roman(5) + Roman(-6)
-I
>>> Roman('V') + (-6)
-I
```

Method: __lt__ and __gt__

```
>>> Roman(5) < Roman(6)
True
>>> Roman(6) < Roman(5)
False
>>> Roman(5) > 6
False
>>> 5 < Roman(6)
True
```

Pay attention to the special case of a zero sum. Since zero cannot be represented by roman numerals, You should raise a you should raise a ValueError with a message of your choice if the sum is zero.

5) Since we are working with only integers, we will implement the floor division method '//' instead of normal division '/'. (Again, only left side operations and not right-hand side).
Special cases:
   a. You can assume that there will be no zero division.
   b. You should raise a ValueError with a message of your choice if the floor division value is zero.

- The method is __floordiv__
  It will work like this:
  >>> Roman(6) // Roman(5)
  I
  >>> Roman('VI') // 5
  I
  >>> Roman(6) // Roman('-V')
  -II
  >>> Roman(6) // -5
  -II
  ~~>>> 6// 5~~ #this example was removed because it's incorrect
  ~~2~~
  >> Roman(2) // Roman(3) # this should raise a ValueError

Question 2 (numpy)

The Ministry of Health wanted to check on the achievements of a new physical training program. For several consecutive months, data on the training candidates (trainees) was collected. The data is collected in *csv* tables (a table contained in a *.csv* file), so that the first column contains the names of the trainees and the first row contains the names of the months. Each row contains data collected at the end of each month. That is, the second column will show the data for each trainee at the end of the first month, the second column at the end of the second month, and so on (see the sample file attached to this homework, *weight_input.csv*, where the weights of the four candidates were measured in kilograms. The first sample month is August).

1. Write a function called *load_training_data* that accepts the path to the *csv* file and returns dictionary with three string keys mapped to the following objects:
   - *"data"*: mapped to the data of the table, in a numpy matrix (without the row of months and the column of candidates)
   - *"column_names"*: mapped to the list of month names according to the order of the columns (from the first line of the table, without the first character), of type *numpy.array* of strings.
   - *"row_names"*: mapped list of trainees according to the order of rows (from the first column of the table, without the first character), of type *numpy.array* of strings.

   Example of execution on the file attached to this homework:

	August	September	October	November	December	January
Orit	84	81.3	82.8	80.1	77.4	75.2
Miki	79.6	75.2	75	74.3	72.8	71.4
Roni	67.5	66.5	65.3	65.9	65.6	64

Assaf	110.7	108.2	104.1	101	98.3	95.5

```
>>> data_dict = load_training_data("weight_input.csv")
>>> print(data_dict["data"])
[[  84.     81.3    82.8    80.1    77.4    75.2]
 [  79.6    75.2    75.     74.3    72.8    71.4]
 [  67.5    66.5    65.3    65.9    65.6    64. ]
 [ 110.7   108.2   104.1   101.     98.3    95.5]]
>>> print(data_dict["column_names"])
['August' 'September' 'October' 'November' 'December' 'January']
>>> print(data_dict["row_names"])
['Orit' 'Miki' 'Roni' 'Assaf']
```

In Sections 2-5 hereunder:

- **The functions receive as parameters the output dictionary of the function of Section 1:, with the keys: *data, row_names, column_names*.**
- **You are not allowed to use loops.**

2. Write a function *get_highest_weight_loss_trainee* that returns the name of the trainee whose weight loss was greatest from the beginning of the program to the end (you may assume there is one). That is, the difference between the start weight and the end weight is greatest. You can use (but it is not required) the *numpy.argmax* function that returns the index where the maximum value is.

   Example of execution on the given input table:

```
>>> get_highest_weight_loss_trainee(data_dict)
'Assaf'
```

3. Let us define the "monthly difference" as the difference for a particular trainee from one month to the previous month. Write a function *get_diff_data* that returns the difference matrix. Each column i contains the difference between the month $(i+1)^{th}$ month and the ith month, so the number of columns in the differences matrix is equal to the number of columns in the data matrix reduced by 1. <u>Note that the input matrix does not change.</u>

   Example of execution on the given input table:

```
>>> get_diff_data(data_dict)
array([[-2.7,  1.5, -2.7, -2.7, -2.2],
       [-4.4, -0.2, -0.7, -1.5, -1.4],
       [-1. , -1.2,  0.6, -0.3, -1.6],
       [-2.5, -4.1, -3.1, -2.7, -2.8]])
```

4. Write a function *get_highest_loss_month* that returns the name of the month where the sum of monthly difference across all candidates is the largest (i.e., the **loss** of weight was maximum). **You may assume that such a month exists**. Note that the function receives

the dictionary containing the keys of *data, columns_names, row_names* (and not the output matrix from section 3). Use the function from Section 3.

Example of execution: in the example matrix, in September, the trainers lost 10.6 kg together, an amount greater than in the other months.

```
>>> get_highest_loss_month(data_dict)
'September'
```

5. Weight change is known to be proportional to the initial weight. Write a function *get_relative_diff_table* that returns the weight change table. For each participant and each month, the table shows the weight change relative to the previous month, i.e. the "monthly difference" divided by the weight of the previous month. Note that the function receives the dictionary containing the keys of *data, columns_names, row_names* (and not the output matrix from section 3). Use the function from Section 3.

   Example of execution:

```
>>> get_relative_diff_table(data_dict)
array([[-0.03214286,  0.01845018, -0.0326087 , -0.03370787, -0.02842377],
       [-0.05527638, -0.00265957, -0.00933333, -0.02018843, -0.01923077],
       [-0.01481481, -0.01804511,  0.00918836, -0.00455235, -0.02439024],
       [-0.02258356, -0.03789279, -0.02977906, -0.02673267, -0.02848423]])
```

   For example, the figure marked in the output above is the result of the calculation:

$$\frac{november - october}{october} = \frac{74.3 - 75}{75} = -0.0093333$$

Question 3 (image processing)

Grayscale images can be represented as a two-dimensional *ndarray*, where each organ is a number in the range 0-255 where 0 represents black and 255 represents white.

Use the *imageio* package to load the images.

1. One of the important measures in information theory is entropy. Entropy is a way to quantify the disorder contained in a word or a picture. Think of an image with just one color, for example black: this picture is very "neat" (note how few words it takes to describe it) and indeed its entropy value is 0. On the other hand, think of a wide-scale image of grayscale: it contains a lot of information and is less "neat". Its entropy is therefore larger.
   In this section, we want to calculate the entropy of a grayscale image and thus quantify the disorder in the image. The formula for calculating the entropy is:

$$S = \sum_{i=0}^{N} -P_i \cdot \log_2 P_i$$

where P_i is the probability to get any shade of grey between 0 and 255. In other words, the P_i are the values of the normalized histogram of the picture. *N* is the total number of shades of grey.
For example, let us compute the entropy of an image containing 4 pixels: 2 white pixels and 2 black pixels:

$$S = \sum_{i=0}^{N} -P_i \cdot \log_2 P_i = -\frac{1}{2} \cdot \log_2\left(\frac{1}{2}\right) - \frac{1}{2} \cdot \log_2\left(\frac{1}{2}\right) = 1$$

In this specific case, the probability of a black or a white pixel is one half.

Write a function called *def compute_entropy(img)* which receives the name of an image *img* (string) and returns the entropy of the greyscale image.

- You need to discard P_i values that are zero
- You may assume that the input is valid: an image of a given size with shades of grey between 0 and 255.
- You may assume that the image contains more than one shade of grey.
- Hint: use *np.bincount*

Example of execution:

```
>>>print(compute_entropy('cameraman.tif'))
>>>7.009716283345514
```

2. When we enlarge an image, we need to find a way to color the new pixels with the help of the old pixels. One of the basic methods for doing this is through the principle of "the nearest neighbor". The pixel in the enlarged image will receive the value of the closest pixel in the original image. The formula for a pixel in the enlarged image will be:

$$Bigpixelvalue[i,j] = smallpixelvalue[floor(i * \frac{ySmallSize}{yBigSize}), floor(j * \frac{xSmallSize}{xBigSize}))]$$

Where *xBigSize* is the size of the *x* axis (number of columns) of the enlarged image, and so on for the other values. The values *i* and *j* are the indexes of the pixels in the enlarged image.

Write a function called *nearest_enlarge(img, a)* which Receives the name of the image *img* (*string*) and the enlargement ratio *a*, greater than 1 (*int*). The function returns an enlarged image (two-dimensional array of type *ndarray*) where the number of pixels of each dimension of the image is multiplied by *a*.

Example of execution:

```
>>>I=nearest_enlarge('cameraman.tif',2)
>>>plt.figure()
>>>plt.imshow(I,cmap = plt.cm.gray)
```

```
>>>plt.show()
```