15-110 Fall 2019
Midterm Exam 02

Out: Sunday 27th October, 2019 at 15:00 AST
Due: Sunday 27th October, 2019 at 16:20 AST

Introduction

This midterm exam includes all the topics studied so far in the course.
The total number of points available from the questions is 115, where 15 points are *bonus* points (i.e., you only need 100 points to get the maximum grade).

Warning: if the code of a function doesn't execute because of syntax errors of *any type* (i.e., the function doesn't even reach the end) then you'll get 0 points, the function won't be evaluated at all during the grading process! This means that you should / must try out your code, function by function, before submitting it!

In the handout, the file `exam02.py` is provided. It contains the functions already defined but with an empty body (or a partially filled body). You have to complete the body of each function with the code required to answer to the questions.
You need to submit to Autolab the `exam02.py` file with your code.

Only the provided *reference cards* (possibly with your annotations) are admitted as a support during the exam.

The code must be written and tested using Spyder on the computers in the classroom.

1 Playlist

You are making a playlist of songs and sometimes you accidentally end up with the same song repeated. Although it is ok for a song to occur more than once in a playlist (because we just cannot get enough of it), we don't like it when it is repeated *immediately* after itself. You would like to have a button on your app to remove such repetitions. After taking 15-110 you realize that you *can* implement this functionality!

Problem 1.1: (18 points)

Implement the function `no_repeat(l)` that removes repeated elements from a list `l` when they occur consecutively, leaving only one copy of it. The function should return a new list, without modifying the input list. For simplification, you can assume the list contains only integers.

For instance, `no_repeat([1,1,2,1,3,3,3,2,2])` must return `[1, 2, 1, 3, 2]`.

Expected lines of code: 5

2 Matrix data

A list of r lists, where each one of the r lists has precisely c numeric elements, can be represented as a rectangular data *matrix* with r rows and c columns. For instance, below, the list of lists to the left represents a matrix with three rows and two columns, while the right list of lists represents a matrix with two rows and four columns.

$$\begin{bmatrix} 1 & 4 & 0.5 \\ -6.2 & 2.0 & 1.5 \\ 4 & 0.3 & 5 \end{bmatrix} \qquad \begin{bmatrix} 1 & 4 & 0.5 & 3 \\ -6.2 & 2.0 & 1.5 & 0 \end{bmatrix}$$

Problem 2.1: (22 points)

Implement the function `floor_matrix(m)` that takes as input a rectangular numeric matrix with r rows and c columns (i.e., a list of r lists of c numeric elements), and *modifies* the input matrix by by applying the function `floor(v)` of the `math` module to each one of the matrix elements. `floor(v)` returns the largest integer less than or equal than `v`. For instance, `floor(9.4)` returns 9.

The function `floor_matrix(m)` returns the maximum of the floored matrix values.

Remember to import and refer to the module `math`.

You can use the `help()` function to check further the behavior of the `floor()` function.

For instance, for `m = [ [1.8, 2, -5.3], [9.7, 0.5, 2.3], [1.2, 8.1, -2] ]`, calling `floor_matrix(m)` returns `9`, and the matrix `m` becomes equal to `[ [1, 2, -6], [9, 0, 2], [1, 8, -2] ]`.

Expected lines of code: 10

3 Get digits

Problem 3.1: (26 points)

Implement the function `get_digit(num, pos=1)` that takes two integers as input and returns the digit at position `pos` in `num`. The digits of `num` are indexed from right to left, starting at 0. The `pos` argument is optional and takes default value 1.

For example `get_digit(15110, 3)` should return the integer 5.

If `num` is not an integer, than the function must return `None`.

If `pos` refers to not a valid position for `num` (or fr the value input by the user), the function must return `None`.

If `num` is a *negative* integer, than the function must ask the user to input a (positive) integer. If the user inputs anything which is not a positive integer, the function should keep asking.

For instance, invoking `get_digit(-100, 2)` causes the function to ask to input a non-negative integer through the keyboard, as shown below, where the user issues two wrong inputs before entering a correct positive integer (234). The value returned by the function in this case is 2.

```
Input a non-negative integer: aa
Input a non-negative integer: -2
Input a non-negative integer: 234.
```

Expected lines of code: 11 (Hint: you should use while loops, break, input() function, isdigit() method, among others)

4 Dictionaries for student grades

Suppose we are keeping track of students' grades using a python string that looks like this:

```
s = '''rick   , 10, 10 , 10 , 9.7, 8.7
       morty  , 8  , 7.5, 10 , 9   , 7.6
       beth   , 7  , 9.6, 8.5, 10
       jerry  , 6  , 5.4, 3.8, 10
       summer, 10, 9.5, 8.5, 5   , 7.2, 8'''
```

Each student is on a separate line, where the first element is the name, followed by the scores, separated by spaces. Notice that not all students have the same number of assignments.

After learning about dictionaries you realize that there is a much better way to store this data than using a string!

Problem 4.1: (26 points)

Implement the function `process(s)` that takes a string in the format described above, and returns a dictionary where the keys are the students' names (without extra spaces), and the values are lists of floats, corresponding to the students' scores.

For example, if `s` is the string above, `process(s)` should return the dictionary:

```
{'beth': [7.0, 9.6, 8.5, 10.0],
 'jerry': [6.0, 5.4, 3.8, 10.0],
 'morty': [8.0, 7.5, 10.0, 9.0, 7.6],
 'rick': [10.0, 10.0, 10.0, 9.7, 8.7],
 'summer': [10.0, 9.5, 8.5, 5.0, 7.2, 8.0]}
```

The following string functions may be useful for this task: `s.split(sep)` and `s.strip()`, while for dictionaries you only need `{ }` and `[ ]` operators.

Expected lines of code: 9

Problem 4.2: (23 points)

Implement the function `compute_avg(d)` that takes a dictionary constructed by the function `process(s)`, and returns another dictionary where each student is associated with its average on all assignments. The first dictionary must be unchanged.

For example, if `d` is the dictionary:

```
{'beth': [7.0, 9.6, 8.5, 10.0],
 'jerry': [6.0, 5.4, 3.8, 10.0],
 'morty': [8.0, 7.5, 10.0, 9.0, 7.6],
 'rick': [10.0, 10.0, 10.0, 9.7, 8.7],
 'summer': [10.0, 9.5, 8.5, 5.0, 7.2, 8.0]}
```

then `compute_avg(d)` should return:

```
{'beth': 8.775,
 'jerry': 6.3,
 'morty': 8.42,
 'rick': 9.68,
 'summer': 8.03}
```

Note that in order to get the above nicely written floats with at most three decimal digits, you have to use the function `round(v, digits)`, that takes as input a float `v` and and integer, `digits`, that specifies the maximum number of decimal digits used to store the float `v`. For instance, the average grade of `summer` is given by `v = (10.0 + 9.5 + 8.5 + 5.0 + 7.2 + 8.0)/6` = 8.0333333.... Rounding `v` as `v = round(v, 3)`, `v` becomes equal to 8.033 (which is the value that will be written in the dictionary if `v` is used).

The function also prints out a string that reports about maximum, minimum, and median values of the average student grades. For instance, for the given example, the function prints out the following *multi-line* string (Note: you have to write the string with a single `print()` statement, not with three distinct `print()` statements!)

```
'Max␣avg␣grade:␣9.68'
'Min␣avg:␣6.3'
'Median␣avg:␣8.42'
```

Expected lines of code: 12