15-110 Fall 2019
Final Exam: Programming Questions

Out: Sunday 8th December, 2019 at 13:30 AST
Due: Sunday 8th December, 2019 at 16:30 AST

Introduction

This final exam includes all the topics studied during the course.

The total number of points available from the all questions (written + programming) is 140, where 40 points are *bonus* points (i.e., you only need 100 points to get the maximum grade).

The written questions provide 40 points, while the programming ones provide 100 points.

You have one hour for the written questions and two hours for the programming questions.

Warning: if the code of a function doesn't execute because of errors of *any type* (i.e., the function doesn't even reach the end producing a meaningful output) 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 `examfinal.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 `examfinal.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 Make a chocolate package

Problem 1.1: (15 points)

We want make a package of `total` kilos of chocolate. We have `small` bars that are small (1 kilo each) and `big` bars that are big (5 kilos each).

Implement the function `chocolate_package(small, big, total)` that takes as inputs the number of small bars that are available, `small`, the number of big bars that are available, `big`, and the total number of kilos of chocolate that must be in the package, `total`. All the input arguments are integers. The function must return the integer number of small bars to use, under the constraint that big bars must be used before small bars, if feasible. If isn't possible to create the package, the function must return -1.
For instance:

- `chocolate_package(4, 1, 4)` returns 4,
- `chocolate_package(3, 1, 9)` returns -1,
- `chocolate_package(6, 1, 11)` returns 6.

☛ Expected number of lines in the solution code: $\sim$ 12

2 Longest increasing subsequence

Problem 2.1: (20 points)

Implement the function `longest_increasing_subseq(x)` that takes as input a sequence of numbers, `x`, and finds the longest subsequence `x[i:j]` of `x`, such that consecutive numbers are monotonically increasing, i.e., `x[k] <= x[k+1]` for all `k` in `range(i,j)`. If there is more than one subsequence with the same length, the first is returned. For instance:

- For `x = [12, 45, 32, 65, 78, 23, 35, 45, 57]`, the longest subsequence is `[23, 35, 45, 57]`, that corresponds to `x[5:8]`.
- For `x = [12, 45, 32, 65, 78, 79, 23, 35, 45, 57]`, there are two longest subsequences, with the first being `[32, 65, 78, 79]`, that corresponds to `x[2:5]`.
- For `x = [12, 11, 12, 11, 12, 11, 12]`, there are three longest subsequences, with the first being `[11, 12]`, that corresponds to `x[1:2]`.
- For `x = [11, 11, 11, 11, 12]`, there is one longest subsequence, that corresponds to the entire list `x`.

The function must return a dictionary with two elements, where the keys are the indexes `i` and `j` of the start and the end of the subsequence, and the values are the entries `x[i]` and `x[j]` in the sequence `x`.

For instance, in the case of the first example above, the function returns `{5: 23, 8: 57}`, while in the case of the second example the returned dictionary is `{2: 32, 5: 79}`.

☛ Expected number of lines in the solution code: $\sim$ 15

3 Read and display data

Each day, temperature data are being collected by sensors placed in different locations and are stored in a csv file. The file stores temperature values taken over different days of the year in six different cities: Doha, Rome, New York, London, Moscow, Paris. Each file record reports the day and the temperature values for the different cities, with the field values being separated by commas.

For instance, the first nine records of the file might look like:

```
Day,      Rome,       NY,    Moscow,     Doha,    London,     Paris
  4,     29.62,     27.34,     28.18,     30.08,     29.50,     25.34
  5,     28.38,     19.15,     28.38,     33.02,     18.03,     28.60
  9,     29.89,     29.09,     18.77,     32.34,      9.10,     24.86
 10,     22.52,      9.68,     26.18,     33.88,     20.47,     23.88
 11,     24.48,     17.77,     15.21,     33.79,     27.02,     22.22
 12,     29.79,     23.90,     13.27,     36.09,     20.34,     21.98
 13,     25.45,     22.50,     14.87,     31.41,     29.93,     21.36
 15,     25.36,     17.27,     10.35,     34.22,     25.85,     24.80
```

The file contains a text header which specifies which field in a record corresponds to which city. For instance, in this example case, Doha's temperatures are stored in the fifth field of each record.

A file `temperatures.csv` with a full range of data is provided in the handout for testing your code.

Problem 3.1: (25 points)

Implement the function `read_temperatures(filename)`, that takes as input a string `filename` with the name of a file with the above structure. The goal of the function is to read the file and store the data only for the cities of Doha, Rome, and Paris in a dictionary data structure.

- You must assume that you don't know how the fields are defined in the file (i.e., you don't know in which fields data for Doha, Rome, and Paris is stored). This information must be acquired by reading the header of the file.
- As a first step, you have to open the file for reading it. If, for any reason, the file passed as argument to the function isn't accessible, the function prints out the message `'Error: file temperatures.dat is not accessible'` and returns `False`. Note that in this example case the file name passed to the function was `'temperatures.dat'` (watch out: you must print out the name of the file that was passed as input, not hard code temperatures.dat in your program!)
- As a second step, you have to read the header of the file in order to learn which fields in the records correspond, respectively, to the data of Doha, Rome, and Paris,

  At this aim, it might be convenient to define a dictionary structure `map_cities_to_fields`, with three string keys, `'doha'`, `'rome'`, `'paris'`, and the field position as the value of a key. For instance, for the file example above, after reading the header, the dictionary would look like: `{'doha': 4, 'rome': 1, 'paris': 6,`
- Now that you know where the fields your are interested in are, you can start by reading the rest of the file, record by record, selecting and storing only the data related to Doha, Rome, and Paris.

  At this aim, you must use a dictionary data structure `data_by_city` with keys `'doha'`, `'rome'`, `'paris'`, `'days'`. The data associated to each key is a list, containing all the temperature values for the city, or the identifier of the days, in the case of the `'days'` key.

For instance, if the temperature file consists of the above eight data records, the resulting dictionary is:

```
{ 'doha': [30.08, 33.02, 32.34, 33.88, 33.79, 36.09, 31.41, 34.22],
  'rome':  [29.62, 28.38, 29.89, 22.52, 24.48, 29.79, 25.45, 25.36],
  'paris': [25.34, 28.6, 24.86, 23.88, 22.22, 21.98, 21.36, 24.8],
  'days': [4, 5, 9, 10, 11, 12, 13, 15] }
```

- In order to avoid to write a uselessly long and potentially bugged code, you should not write ad hoc lines of code for reading and processing the data of each city (i.e., Doha, Rome, Paris), but you should rather exploit the dictionary data structure `map_cities_to_fields` suggested above to write a compact piece of code that can work for any of the three cities.
- Note that the file might contain comment lines, that are identified by the presence of the character `'#'` as first character of the record. In this case, the record must be discarded, since it doesn't contain useful data to store.
- The function returns the dictionary data structure with the selected temperature data.

☛ Expected number of lines in the solution code: ~ 30

Once data are conveniently stored in a dictionary data structure, it is easy to further process and display the data. The following three functions precisely aim to achieve these goals.

Problem 3.2: (15 points)

Implement the function `get_statistics(temperature_dict)` that takes as input a dictionary with temperature data as described above (which would be the result of the previous function).

When called, the function `get_statistics()` prints out some statistics about the stored data. More specifically, it prints out the following data with the following multiline format (the specific numbers refer to the example data above):

```
doha - min: 30.08, max: 36.09, median: 33.79, mean: 33.10
rome - min: 22.52, max: 29.89, median: 28.38, mean: 26.94
paris - min: 21.36, max: 28.60, median: 24.80, mean: 24.13
```

Note that `min` and `max` refer to respectively the minimum and the maximum value of the recorded temperatures for the city, while `median` and `mean` are the median and the mean value of the temperatures.

All values must be printed with two decimal digits.

The function returns a tuple of floats, `(all_min, all_max, all_mean)`, where the min, max, and mean values are computed out of the combined data of the three cities.

For instance, in the case of the example data above, the function returns the tuple:
`(21.36, 36.09, 28.06)`.

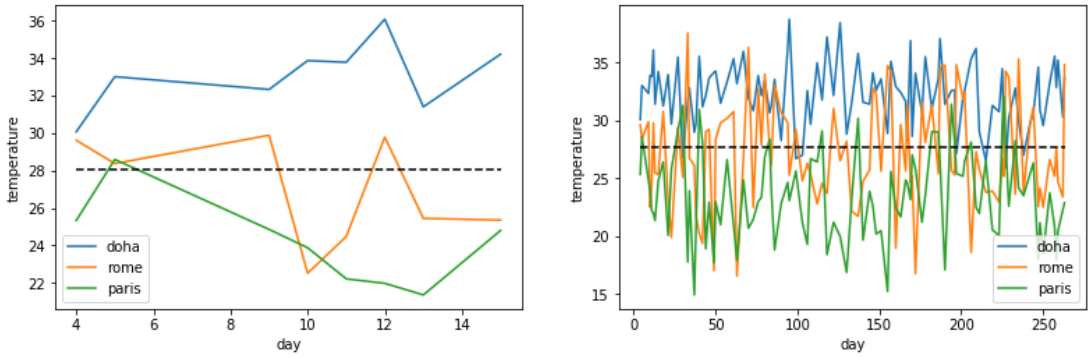
☛ Expected number of lines in the solution code: ~ 18

Problem 3.3: (15 points)

Implement the function `display_trend(temperature_dict)` that takes as input a dictionary with temperature data as described above. The function generates a plot with all the temperature data for the selected three cities. The plot must look like the one shown to the left in the figure below for the case of the example data above. The plot to the right is the one for the full data file `temperatures.csv` included in the handout.

The black dashed line represents the average temperature value computed out of all temperatures for the three cities (Hints: you can use the `get_statistics()` function to obtain the average value for free; remember that to draw a line parallel to an axis you only need the coordinates of the two end points of

the line, where in this case the list `temperature_dict['days']` provides the x values that you need).



☛ Expected number of lines in the solution code: ~ 18

Problem 3.4: (10 points)

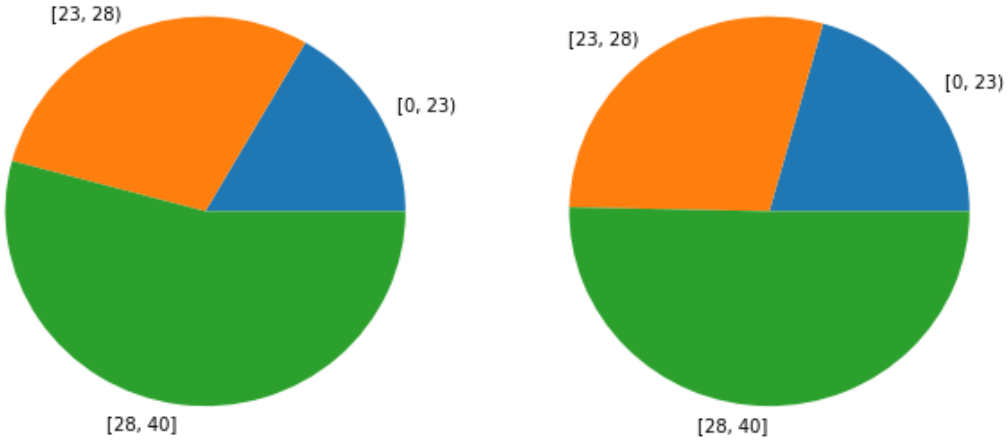
Implement the function `display_aggregate(temperature_dict)` that takes as input a dictionary with temperature data as described above. The function generates a pie chart with the aggregate temperature data for the selected three cities (i.e., considering all data from the three cities as a single set of data).

The pie chart shows the relative percentage of temperature values in the following three intervals:

- $[0, 23)$
- $[23, 28)$
- $[28, 40)$

The pie chart must look like the one shown to the left in the figure below for the case of the example data above. The pie chart to the right is the one for the full data file `temperatures.csv` included in the

handout.



☛ Expected number of lines in the solution code: ~ 18