Problem Set 11 - Classes

Hello student,

the goal of this problem-set is to start using classes.
Classes are a crucial element of modern programming, they are used to models complex programs allowing developers to create meaningful abstractions.

Assignment 11.1 - Fractions (3 points)

In this exercise, you are asked to develop a class representing a fraction (numerator and denominator).

The methods this class will expose are the following:

- **sum** - sum of two fractions, returning a new Fraction object;
- **product** - multiply two fracitons, returning a new Fraction object;
- **opposite** - calculate the opposite of a fraction, returning a new Fraction object;
- **inverse** - calculate the inverse of a fraction, returning a new Fraction object;
- **prettify** - returns a pretty representation of the fraction in string format.

As a reminder:

Sum: $\frac{a}{b} + \frac{c}{d} = \frac{ad+bc}{bd}$ Product: $\frac{a}{b} * \frac{c}{d} = \frac{ac}{bd}$

Opposite: $\frac{a}{b} = -\frac{a}{b}$ Inverse: $\frac{a}{b} = \frac{b}{a}$

Furthermore, upon creation each function needs to be reduced, this is done by finding the GCD (great common divisor) using the Euclidean Algorithm.

The GCD can be found using a recursive function (yey!) defined as follows:

- Given two numbers a and b;
- If b is 0, return a;
- Otherwise, $r = a\%b$ (reminder of division);
- If r is 0, return b;
- Otherwise, $a = b$ and $b = r$, then re-calculate the GCD.

Write then an output showing the functionalities of your software, for example starting with two fractions ($\frac{-1}{-3}$ and $\frac{8}{6}$) a possible result could be:

```
f1 = 1.0/3.0
f2 = 4.0/3.0
opposite of f1 = -1.0/3.0
inverse of f1 = 3.0
f1 + f2 = 5.0/3.0
f1 * f 2 = 4.0/9.0
```

Assignment 11.2 - BMI with classes (3 points)

Last week you were asked to re-iterate the original assignment **assignment 8.2** using files (in **assignment 10.2**).

This week we will add a third iteration using classes. The scope of this exercise is to understand how to abstract the logic using classes.

The output of the software should still be the same, only this time you use a `main` file and 3 classes.

The required classes and methods are:

Person - represents a person, with all the data as fields (instance variables). It will expose the following methods:

- `__init__` (constructor), which will receive all the parameters (name, height, weight, gender) and optionally the bmi;
- `calculate_bmi`, which will return the bmi;
- `as_dict`, which will represent the object as a dictionary.

People - a class containing a list of people, which will allow to perform operations on multiple persons, the following methods are required:

- `__init__` (constructor), which will receive a list of person as parameter;
- `calculate_people_bmi`, returns a new instance of People which contains new instances of Person with the BMI;
- `calculate_highest_bmi`, returns the Person instance with highest BMI;
- `calculate_lowest_bmi`, returns the Person instance with lowest BMI;
- `calculate_average_bmi`, returns the average BMI;
- `calculate_male_average_bmi`, returns the average BMI for males;
- `calculate_female_average_bmi`, returns the average BMI for females;
- `as_dict`, which will represent the object as a dictionary (a list of dictionaries).

FileInterface - a class allowing the user to perform operations on files, it must contain the following methods:

- `load_people`, provided a file-name, it returns an instance of People with all the lines loaded as Person instances;
- `save_dict_to_csv`, given a list of Person (as dictionaries) and a file-name, saves the list as csv.

As a reminder, an example of the returning files is the following:

- **bmi.csv** - containing all people with their BMI
- **highest_bmi.csv** - containing the person with the highest BMI
- **lowest_bmi.csv** - containing the person with the lowest BMI
- **average_bmi.csv** - containing the overall average, the males average and the females average

An example of the expected content of the files is reported here:

```
# bmi.csv
name,height,weight,gender,bmi
Patricia Miller,1.66,49.0,female,17.78
William Adams,1.68,60.0,male,21.26
George Clark,1.86,94.0,male,27.17
John Hall,1.54,96.0,male,40.48
...

# highest_bmi.csv
name,height,weight,gender,bmi
Richard Torres,1.5,109.0,male,48.44

# lowest_bmi.csv
name,height,weight,gender,bmi
Amanda White,1.94,45.0,female,11.96

# average_bmi.csv
overall,males,females
26.48,26.51,26.45
```

Assignment 11.3 - Setting migration pt. 2 (optional)

During **assignment 9.3** you were asked to develop a program to migrate settings from one older version of the settings-manager to the new.
For this assignment you are asked to expand a little bit the software using the new knowledge you acquired: classes.
What you need to do is dividing the software into different classes. This has the advantage of being clearer (thanks to abstraction) and allow code re-use.

The classes and functions you must create are the following:

Setting - represent a setting (bind, key, action) and has the following method:

- `pretty`, returns a pretty representation of the binding as a string;

SettingManager - used to manage a set of settings, it also maintains the history of the change made, it has the following methods:

- `__init__` (constructor), which receives optionally a list of Setting, it stores them internally;
- `update_keybinding`, given a new Setting, stores it instead of the previous bind, saves the change made in local history;
- `get_action`, returns the action associated with a key;
- `history_pretty`, presents history in a pretty format (as a list of string);
- `flush_history`, clears history.

SettingConverter - used to read settings in the old format, it has the following methods:

- `__init__` (constructor), receives the name of the file to work on (old settings);
- `parse_settings`, returns a SettingManager populated with its Setting instances;

BinarySettingsIo - manages settings in the new format, it has the following methods:

- `__init__` (constructor), receives the name of the file with the binary settings (to write or to read);
- `save_settings`, saves the history (from a SettingManager instance) to a "history.cfg" file (old, new, timestamp). It then checks if a binary configuration file is already present, in which case it renames it by appending ".old" at the end, then dumps the data (SettingsManager) using pickle;
- `load_settings`, creates a SettingsManager instance from a binary file using pickle;

Some examples and hints are available on the next page.

An example of the main function could be the following:

```
from setting_converter import SettingConverter
from binary_settings_io import BinarySettingsIo

if __name__ == "__main__":
    settings_file_name = "settings.cfg"
    binary_setting_io = BinarySettingsIo(settings_file_name)

    # load settings from old format to new
    converter = SettingConverter(settings_file_name)
    setting_manager = converter.parse_settings()

    # save new settings
    binary_setting_io.save_settings(setting_manager)

    # reload new settings
    new_settings = binary_setting_io.load_settings()

    # invert settings of "e" and "f"
    e = new_settings.get_action("e")
    f = new_settings.get_action("f")

    e.key = "f"
    f.key = "e"

    new_settings.update_keybinding(f)
    new_settings.update_keybinding(e)

    binary_setting_io.save_settings(new_settings)
```

An extract of the "history.cfg" file is the following (please note this are only 2 lines in the file, they are wrapped only for space issues):

```
[bind: bind_US_standard key: f action: +ping], [bind: bind_US_standard key: e
   action: +use;], 2020-11-30 18:30:55.020498
[bind: bind_US_standard key: e action: +use;], [bind: bind_US_standard key: f
   action: +ping], 2020-11-30 18:30:55.020518
```

Hint: to rename files use pathlib.Path.rename (https://docs.python.org/3/library/pathlib.html#pathlib.Path.rename)
Hint2: to get the current datetime, cast to string datetime.datetime.now (https://docs.python.org/3/library/datetime.html?highlight=datetime#datetime.datetime.now)