Python Programming

Homework 4

Instructor: Dr. Ionut Cardei

The following problems are from Chapters 9 (Dictionaries and Sets), 11 (Intro to Classes), and 12 (More on Classes).

For full credit:

- make sure you follow exactly all requirements listed in this document,
- the program must have all required features and run as expected.

Write your answers into a new Word document called **h4.doc**.
Write your full name at the top of the file and add a heading before each problem solution.

For each problem:

- write a **visible subtitle** with the problem number
- insert the code from p1/2/3.py
- insert any additional items, such as screenshots

Convert the doc file to PDF.
For this homework you need to upload on Canvas the PDF file and the Python .py source files (p1.py, p2.py,…).

Don't forget to read the **Important Notes** at the end of this file.

Problem 1. Top Movies and Actors

This problem is about analyzing data from IMDB lists with top rated and top grossing movies.
There are these files linked from the Homework 4 Canvas page:

- imdb-top-rated.csv, listing the ranking of the top rated 250 movies. It has this format: *Rank,Title,Year,IMDB Rating*
- imdb-top-grossing.csv, listing the ranking of the highest grossing 250 movies. It has this format: *Rank,Title,Year,USA Box Office*
- imdb-top-casts.csv, listing the director and cast for the movies in the above files. It has this format: *Title, Director, Actor 1, Actor 2, Actor 3, Actor 4, Actor 5*. The actors are listed in billing order. This file does not have a heading.

These files are from Duke U. and seem to date from 2014.

Write a program in file **p1.py** that does the following:
a) Displays a ranking (descending) of the movie directors with the most movies in the top rated list. Print only the top 5 directors, with a proper title above.

b) Displays a ranking (descending) of the directors with the most movies in the top grossing list. Print only the top 5 directors, with a proper title above.

c) Displays a ranking (descending) of the actors with the most movie credits from the top rated list. Print only the top 5 actors, with a proper title above.

d) Displays a ranking (descending) with the actors who brought in the most box office money, based on the top grossing movie list. For a movie with gross ticket sales amount *s*, the 5 actors on the cast list will split amount *s* in the following way:

Actor #	1 (first billed)	2	3	4	5
$$ per actor	16*s/31	8*s/31	4*s/31	2*s/31	s/31

Print only the top 5 actor pairs, with a proper title above.

EXTRA CREDIT 5 points:
e) Displays in order the top 10 “grossing actor pairs” that played together in the same movie. The total amount (used for sorting) for a pair of actors is the sum of the gross revenue allocated to each actor with the scheme in the table above, but computed only for movies where the two actors played together. We can expect Harrison Ford and Mark Hamill to be near the top, since they were together in the original Star Wars movies.

Take a screenshot of the program’s output (a-d + e) and insert it in the h4.doc file right after the code from file p1.py.

Design and Implementation Requirements
To get credit for this problem, follow these requirements:
a) Apply the top-down design methodology. Seek commonality among the tasks listed above. Break the problems into subproblems divide&conquer-style, then write functions dealing with the subproblems.

b) Compute and use 3 dictionaries with the key in the form of a tuple (movie_name, movie_year) for storing movie cast information, ratings info, and gross info, respectively. We need to include the year as part of the key since it’s possible in principle to get two different movies with the same title, but it is less likely to be from the same year. Use a dictionary for storing actor information with the key being the actor name and value being the list of (movie_name, movie_year) tuples for movies in which they played, and any other data needed, such as gross allocated (per the table above). Store multiple values for one entry in a tuple or list.

c) Write docstrings for functions and comment your code following the guidelines from the textbook. Follow the Python coding style rules.

Problem 2. Polynomials

Design and implement (in file p2.py) a class called *Poly* for representing and operating with polynomials.

A polynomial in variable X of degree $n \geq 0$ has this general expression:

$P(X) = a_0 + a_1 X + a_2 X^2 + \ldots + a_{n-1} X^{n-1} + a_n X^n$

with coefficients $a_k \in \mathbb{R}$ and $a_n \neq 0$.

Find out more about polynomials at
http://www.mathplanet.com/education/algebra-1/factoring-and-polynomials/monomials-and-polynomials
or at
https://www.math.auckland.ac.nz/class255/08s1/01s2/polynomials.pdf

a) The *Poly* class must support the following operations, illustrated with examples below:

- Initialization, with the constructor taking an iterable (like a list [$a_{0,} a_{1,} \ldots, a_n$]) that generates the coefficients, starting with a_0. The coefficients are *float*s or *int*s, but the constructor must convert them to type *float*. The degree of the polynomial is given by the length of the sequence of the sequence.
- Conversion to string (__str__). Skip terms $a_k X^k$ with coefficient a_k=0. Use the default number of decimals for *float*s.
- Representation, for printing at the terminal (__repr__).
- Indexing. This operation takes parameter *k* and returns the coefficient a_k if 0<=k<=n or throws *ValueError* otherwise. If *p* is a Poly object *p*[*k*] returns a_k. (__getitem__)
- Addition with another *Poly* object (__add__).
- Multiplication with another *Poly* object and with a *float* or an *int*. (__mul__ and __rmul__)
- Testing for equality (__eq__, __ne__). Two polynomials are equal if their coefficients are respectively equal. Equal polynomials must be of the same degree.
- Evaluation of the polynomial for a given value x for variable X. The method is called *eval* :
    - if x is an *int* or *float* then *p.eval*(x) returns the value of expression $\sum_{k=0}^{n} a_k x^k$.
    - if x is a sequence of elements x_0, x_1,… (an iterable, such as a tuple or a list), then *p.eval*(x) returns a list with the matching elements [*self.eval*(x_0), *self.eval*(x_1), ….]. Use a list comprehension for this evaluation.

b) Write in file **p2.py** a function called *test_poly* that tests **all operations and methods** from part a). Use the function **testif**() from Homework 3 or something similar.

Here are examples how *Poly* objects could be used from the Python shell:

```
p1 = Poly([1, -2, 1])    # poly of grade 2: p1(X)=1-2X+X^2
p2 = Poly((0, 1))        # create poly of grade 1 with a tuple: p2(X)=X, (a_0==0)

print(p1)                # print calls __str__ and prints 1.0-2.0X+X^2

p1                       # python calls __repr__ and displays 1.0-2.0X+X^2

p1 == p2                 # returns False
p1 == Poly((1, -2, 1))   # return True
p1 != p2                 # returns True

p3 = p1 + p2             # sum, __add__
print(p3)                # prints 1.0-X+X^2.0   (use default number of decimals)

p1[1]                    # indexing the coefficients: returns -2 (a1 for p1)

p4 = p2 * p1             # product with another Poly: p4 becomes X-2X^2+X^3
p5 = p1 * 2              # product with int or float: p5 becomes 2-4X+2X^2
p6 = 3 * p1              # product with int or float: p6 becomes 3-6X+3X^2 (__rmul__)

print( p1.eval(2) )      # evaluate p1 at point 2: prints 1.0
print( p1.eval([0,-1,1]) # evaluate p1 for a list of points: prints [1,4,0]
```

Extra credit: 2 points
c) add a method to class *Poly* called *graphit*(*xseq*) that takes a sequence of *float*s in parameter *xseq* (such as a list), evaluates with method eval() the polynomial in the *xseq* points, and then plots the function nicely using the pylab or matplotlib *plot*() function. Display the coordinate axes and proper labels. Use sufficient points in *xseq* to make the chart smooth. Include a **screenshot of the plot** in file h4.doc.

Problem 3. HR Matters

a) Design and implement in a file **p3.py** a class hierarchy for employees in a startup company. The base class is Employee. This has subclasses Manager, Engineer. Class Manager has a subclass called CEO.

Each employee has these:

- Data attributes: name (string), base salary (float), phone number (string). Make these private attributes (__ prefix).
- Constructor taking and saving as attributes name, phone number, and base salary. (Ensure proper call chain for superclass constructors.)
- Methods: accessors for the name and the phone number and a method called ***salary_total***() that returns the total salary, including any extra benefits that subclasses might have. A method (called a *mutator*) that updates the base salary.
- The __str__ method to generate a string representing the object. E.g. "Manager("Sophia Loren","561-2977777", 100000).

- The __repr__ method to generate the official string representation of the object.

An Engineer object does not have anything in addition to what an Employee has.

A Manager has in addition to Employee a bonus (float). Its total salary is the base salary + bonus.

A CEO has in addition (to a Manager) stock options (float). Its total salary is the base salary + bonus + stock options. (in the real world stock options are never added to the salary, though)

The ***salary_total*** method must be overridden in subclasses to compute the total salary as described above. It should NOT access the superclass base salary attribute, but it should use the *salary_total*() method provided by the base class or superclass.

b) Write a function *print_staff*() that takes a sequence (e.g. a list) of employee objects (incl. subclass instances) and prints their name, phone#, and **total salary**, with one object per line.

Write a *main*() method in file p3.py that demonstrates the classes described above.

Among other code, create in *main*() several instances of each class in the employee hierarchy and add them to a list, then call *print_staff()* on that list. (if everything works correctly, this is an illustration of polymorphism)

Submission Instructions

Convert the **h4.doc** file to PDF format (file **h4.pdf**) and upload the following files on Canvas by clicking on the Homework 4 link:

1. h4.pdf
2. p1.py
3. p2.py
4. p3.py

Grading (100 points max + extra credit):

Problem 1: 35% + 5 extra credit points

- algorithm and code correctness: 25%
- programming style: 5%
- screenshot: 5%

Problem 2: 35% + 2 extra credit points

- algorithm and code correctness: 25%
- programming style: 5%
- screenshot: 5%

Problem 3: 30%

- code correctness: 25%
- programming style: 5%
- screenshot: 5%

IMPORTANT NOTES:

- A submission that does not follow the instructions 100% (i.e. perfectly) will not get full credit.
- Upload the PDF, **and .py source files** on Canvas.
- Only submissions uploaded before the deadline will be graded.
- You have unlimited attempts to upload this assignment, but only the last one uploaded before the deadline will be graded.