Michael Quach

Problem 1

```
#Michael Quach
#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.
#
rated = open("imdb-top-rated.csv","r",encoding='utf8')         #Rank,Title,Year,IMDB Rating
gross = open("imdb-top-grossing.csv","r",encoding='utf8')   #Rank,Title,Year,USA Box Office
casts = open("imdb-top-casts.csv","r",encoding='utf8')         #Title,Director,Actor1,Actor2...Actor5

ratedd = {}
grossd = {}
castsd = {}

for line in rated:
        thisline = line.strip().split(',')
        ratedd = {(thisline[0],thisline[1]): (thisline[2],thisline[3])}
        print(ratedd[(thisline[0],thisline[1])])

for line in gross:
        thisline = line.strip().split(',')
        grossd = {(thisline[0],thisline[1]): (thisline[2],thisline[3])}
        print(grossd[(thisline[0],thisline[1])])

for line in casts:
        thisline = line.strip().split(',')
        castsd = {(thisline[0],thisline[1]): (thisline[2],thisline[3])}
        print(castsd[(thisline[0],thisline[1])])

#######Wait what? How the hell do I sort a bunch of tuples by one of its values? Why am I not
using a list or matrix for this????
#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

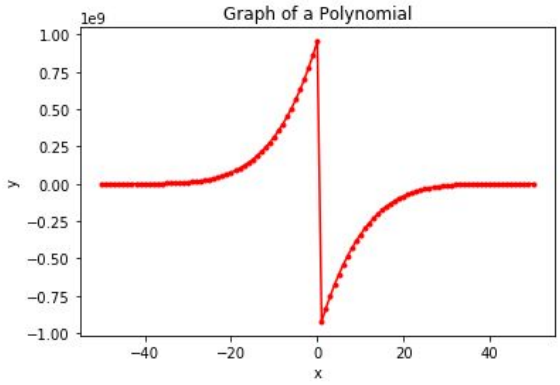
```
#Michael Quach
#Problem 2. Polynomials
import pylab
#Design and implement (in file p2.py) a class called Poly for representing
#and operating with polynomials.
class Poly:
        coeffs=[]
#       a) The Poly class must support the following operations, illustrated with examples below:
#-- Initialization, with the constructor taking an iterable (like a list [ ])
#that generates the coefficients, starting with a0. The coefficients are floats
#or ints, 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.
        def __init__(self,coeffs):
        index=0
        while(index<len(coeffs)):
        self.coeffs.append(float(coeffs[index]))
        index+=1
#-- Conversion to string (__str__). Skip terms akXk with coefficient ak=0.
#Use the default number of decimals for floats.
        def __str__(self):
        nomial = ''
        index=0
        while(index<len(self.coeffs)):
        nomial =  str(self.coeffs[index]) + 'x^' + str(index) + ' + ' + nomial
        index+=1
        nomial = nomial + '0'
        return nomial
#-- Representation, for printing at the terminal (__repr__).
        def __repr__(self):
        return self.__str__()
#-- Indexing. This operation takes parameter k and returns the coefficient
#ak if 0<=k<=n or throws ValueError otherwise.
#If p is a Poly object p[k] returns ak. (__getitem__)
        def __getitem__(self, k):
        if 0<=k<=len(self.coeffs):
        return self.coeffs[k]
        else:
        print("ValueError:",k,"is not within the bounds of this polynomial.")
#-- Addition with another Poly object (__add__).
        def __add__(self, other):
        result = []
        index=0
        while(index<len(self.coeffs)):
```

```
        result.append(self.coeffs[index]+other.coeffs[index])
        index+=1
        return result
#-- Multiplication with another Poly object and with a float or an int.
#(__mul__ and __rmul__)
        def __mul__(self, other):
        if(type(self)!=type(other)):
        result=[]
        index=0
        while(index<len(self.coeffs)):
                result.append(self.coeffs[index] * other)
                index+=1
        return result
        else:
        result=[]
        selfi=0
        while(selfi<len(self.coeffs)):
                otheri=0
                while(otheri<len(self.coeffs)):
                try:
                result[selfi+otheri] += self.coeffs[selfi] * other.coeffs[otheri]
                except IndexError:
                start = len(result)
                finish = selfi+otheri
                while (start<finish):
                        result.append(0)
                        start+=1
                result.append(self.coeffs[selfi] * other.coeffs[otheri])
                otheri+=1
                selfi+=1
        return result

        def __rmul__(self,other):
        return other*self
#-- Testing for equality (__eq__, __ne__). Two polynomials are equal if their
#coefficients are respectively equal. Equal polynomials must be of the same degree.
        def __eq__(self, other):
        if(len(self.coeffs)==len(other.coeffs)):
        index=0
        while(index<len(self.coeffs)):
                if(self.coeffs[index] != other.coeffs[index]):
                return False
                index+=1
```

```
        return True
        else:
        return False

        def __ne__(self, other):
        return not self.__eq__(other)
#-- 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
######Sigma(k=0 to n, a_k*x^k)######
#◦ if x is a sequence of elements x0, x1,… (an iterable, such as a tuple or a list),
#then p.eval(x) returns a list with the matching elements
#[self.eval(x0), self.eval(x1), ….  ].  Use a list comprehension  for this evaluation.
        def eval(self, other):
                if (type(other)==int or type(other)==float):
                        result=0.0
                        index=0
                        while(index<len(self.coeffs)):
                                result += self.coeffs[index]*(other**index)
                                index+=1
                        return result
                else:
                #       result=[]
                #       for index in other:
                #               result.append(self.eval(other[index]))
                #       return result
                        return [self.eval(other[index-1]) for index in other]

        def graphit(self, xseq):
                yseq=self.eval(xseq)
                pylab.plot(xseq, yseq, 'r.-')
                pylab.title('Graph of a Polynomial')
                pylab.xlabel('x')
                pylab.ylabel('y')
                pylab.show()
#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.
#
def test_poly(nomial):
        print('nomial:',nomial)
        print('nomial.__str__:',nomial.__str__)
        print('nomial.__repr__:',nomial.__repr__)
        print('nomial[3]:',nomial[3])
        print('nomial+nomial:',nomial+nomial)
```

```
        print('nomial*nomial:',nomial*nomial)
        print('nomial*11:',nomial*11)
        print('nomial==nomial:',nomial==nomial)
        print('nomial!=nomial:',nomial!=nomial)
        print('nomial.eval(5):',nomial.eval(5))
        print('nomial.eval([1,2,3,4]):',nomial.eval([1,2,3,4]))
        nomial.graphit([i for i in range(-50,51)])
#Extra credit: 2 points c) add a method to class Poly called graphit(xseq)
#that takes a sequence of floats 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.

polymer = Poly([0.5,1,1.5,2,2.5,3])
test_poly(polymer)
```

```
nomial: 3.0x^5 + 2.5x^4 + 2.0x^3 + 1.5x^2 + 1.0x^1 + 0.5x^0 + 0
nomial.__str__: <bound method Poly.__str__ of 3.0x^5 + 2.5x^4 + 2.0x^3 + 1.5x^2 + 1.0x^1 + 0.5x^0
+ 0>
nomial.__repr__: <bound method Poly.__repr__ of 3.0x^5 + 2.5x^4 + 2.0x^3 + 1.5x^2 + 1.0x^1 +
0.5x^0 + 0>
nomial[3]: 2.0
nomial+nomial: [1.0, 2.0, 3.0, 4.0, 5.0, 6.0]
nomial*nomial: [0.25, 1.0, 2.5, 5.0, 8.75, 14.0, 17.5, 19.0, 18.25, 15.0, 9.0]
nomial*11: [5.5, 11.0, 16.5, 22.0, 27.5, 33.0]
nomial==nomial: True
nomial!=nomial: False
nomial.eval(5): 11230.5
nomial.eval([1,2,3,4]): [10.5, 160.5, 1002.5, 3868.5]
```



Problem 3

```
#Michael Quach
#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.
```

```
class Employee:
#Each employee has these:
#-- Data attributes:
#       name (string),
        __name=''
#       base salary (float),
        __salary=0.0
#       phone number (string).
#       Make these private attributes (__ prefix).
        __phone=''
#-- Constructor taking and saving as attributes name, phone number,
#and base salary. (Ensure proper call chain for superclass constructors. )
        def __init__(self, name, salary, phone):
                self.__name=name
                self.__salary=salary
                self.__phone=phone
#-- 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.
        def getname(self):
                return self.__name
        def getphone(self):
                return self.__phone
        def salary_total(self):
                return self.__salary
#A method (called a mutator) that updates the base salary.
        def setsalary(self, salary):
                self.__salary=salary
#-- The __str__ method to generate a string representing the object.
#E.g. “Manager(“Sophia Loren”,”561-2977777”, 100000).
        def __str__(self):
                return str('Employee('"'+self.__name+'"','"'+self.__phone+'"','+str(self.__salary)+')')
#-- The __repr__ method to generate the official string representation of the object.
        def __repr__(self):
                return self.__str__()

#
#An Engineer object does not have anything in addition to what an Employee has.
class Engineer(Employee):
        def __init__(self, name, salary, phone):
                self.__name=name
                self.__salary=salary
                self.__phone=phone
```

```
	def __str__(self):
		return str('Engineer("'+self.__name+'","'+self.__phone+'",'+str(self.__salary)+')')
	def __repr__(self):
		return self.__str__()
#A Manager has in addition to Employee a bonus (float).
#	Its total salary is the base salary + bonus.
class Manager(Employee):
	__bonus = 0.0
	def __init__(self, name, salary, phone):
		self.__name=name
		self.__salary=salary
		self.__phone=phone
	def salary_total(self):
		return self.salary_total()+self.__bonus
	def __str__(self):
		return str('Manager("'+self.__name+'","'+self.__phone+'",'+str(self.__salary)+')')
	def __repr__(self):
		return self.__str__()
#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)
class CEO(Manager):
	__stock = 0.0
	def __init__(self, name, salary, phone):
		self.__name=name
		self.__salary=salary
		self.__phone=phone
	def salary_total(self):
		return self.salary_total()+self.__stock
	def __str__(self):
		return str('CEO("'+self.__name+'","'+self.__phone+'",'+str(self.__salary)+')')
	def __repr__(self):
		return self.__str__()
#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.
#
def print_staff(employed):
	item=0
	while(item<len(employed)):
		print(str(employed[item]))
	item+=1
```

```
#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)

Jake = Employee('Jake',10.00,'123456789')
Jake2 = Employee('Jakey',10.01,'012345678')
Josh = Employee('Josh',11,'1293485760')
Bob = Engineer('Bob', 15, '9191919191')
Sallary = Manager('Sallary',100,'9549549549')
Bill_Gates = CEO('Bill Gates',1000001,'1101000110')

emp = [Jake,Jake2,Josh,Bob,Sallary,Bill_Gates]
print_staff(emp)
```