CSE101 – Introduction to Computers
Python Programming Assignment # 5
(25 points, Submission due date: Friday 31 May 2019)

Instructions

For each of the following problems, create an error free efficient Python program. Each program should be submitted in a separate Python file respectively that follows a particular naming convention. (E.g. The Python program for Question 1 should be in .py file with name Assign4Answer1.py. The Python program for question 2 should be in .py file with name Assign4Answer2.py. Include one or two input cases in your program. The program should execute properly in PyCharm).

Problems

Problem 1: File encryption using Caeser cipher dictionary (8 points)

A. Write a function named `encrypt` that will be passed two arguments, a string and a dictionary that represents a Caeser cipher that specifies how letter substitutions should be performed. In this cipher, each letter is replaced by the letter three places later in the alphabet, i.e. "a" becomes "d", "b" turns into "e", and so on. At the end of the alphabet the cipher "wraps around" so "x" is replaced by "a". The return value of `encrypt` function should be a new string where every letter from the input string has been translated according the cipher.

```
>>> cipher = {’a’:’d’, ’b’:’e’, ’c’:’f’, … ’x’:’a’, ’y’:’b’, ’z’:’c’}
```

This call to `encrypt` shows an example of how the cipher is applied to an input string:

```
>>> encrypt("abba", cipher)

’deed’
```

Define a complete dictionary of Caeser cipher using the statement shown above and test your function on several other strings that contain only the letters "a", "b", and "c". (2 points)

B. Create a new function named `encryptCase` which is a modification of your encrypt function created in part A so that (1) if the input string contains a character that is not in the cipher the character is just copied to the output string and (2) case is preserved, i.e. if the input has an upper case letter the corresponding letter in the output is also upper case. (3 points)

```
>>> encryptCase(’Et tu, Brute?’, cipher)

’Hw wx, Euxwh?’
```

C. Write a function named `crypto_quote` that will read the text in a file, encrypt it with your `encryptCase` function, and print the result on the terminal window. The two arguments to your function should be the path of a text file and the cipher to use. Here is an example that encrypts the quote by Cicero using the cipher: (3 points)

```
>>> crypto_quote("./data/quote1.txt", cipher)
```

```
“Wrxog brx olnh ph wr jlyh brx d irupxod iru vxffhvv? Iw’v txlwh vlpsoh,
uhdoob: Drxeoh brxu udwh ri idloxuh. Yrx duh wklqnlqj ri idloxuh dv wkh
hqhpb ri vxffhvv. Bxw lw lvq’w dw doo. Yrx fdq eh glvfrxudjhg eb idloxuh
ru brx fdq ohduq iurp lw, vr jr dkhdg dqg pdnh plvwdnhv. Mdnh doo brx
fdq. Bhfdxvh uhphpehu wkdw’v zkhuh brx zloo ilqg vxffhvv.”
   -Tkrpdv J. Wdwvrq
```

Problem 2: CSV File Processing (9 points)

A comma-separated values (CSV) file is a delimited text file that uses a comma to separate values. A CSV file stores tabular data (numbers and text) in plain text. Each line of the file is a data record. Each record consists of one or more fields, separated by commas. The use of the comma as a field separator is the source of the name for this file format. Refer to the part on "Parsing CSV Files With Python’s Built-in CSV Library" from a tutorial on Reading and Writing CSV files available at the link: https://realpython.com/python-csv/#writing-csv-file-from-a-dictionary-with-csv

A. Modify the code given in section ‘Reading CSV Files Into a Dictionary With csv’ to write a function `readcsv()` which reads csv file at the path ‘./data/worldpopulation.csv’ and prints following output for each country: (4 points)

```
GI Gibraltar has population of 34.733 in 2018, and will have
population of 35.897 in 2030.
TC Turks and Caicos Islands has population of 35.963 in 2018, and
will have population of 41.528 in 2030.
```

B. Refer to section on ‘Writing CSV File From a Dictionary With csv’. Create a function `writecsv()` which extend the `readcsv()` function to perform the following:
   a. Calculate ratio of 2030 population and 2018 population for each country. E.g. for South Korea, this ratio is 1.0300.
   b. Create a file `writecsv.py` which contains a program to write this ratio along with other parameter values in file `‘./data/worldpopulationchange.csv’`. The first few lines in this file should look like the following: (5 points)

Flag	Country	Population2018	Population2030	ratio
CN	China	1415045.928	1441181.813	1.018469991
IN	India	1354051.854	1512985.207	1.11737612
US	United States	326766.748	354711.67	1.085519479
ID	Indonesia	266794.98	295595.234	1.10794901

Tip: Prior to reading data in the worldpopulation.csv file, create output csv file and its headers using the following:

```
output_file = open('./data/worldpopulation_growth.csv', mode='w')
fieldnames = ['Flag', 'Country', 'Population2018', 'Population2030',
'ratio']
output_writer = csv.DictWriter(output_file, fieldnames=fieldnames)
output_writer.writeheader()
```

In a dictionary `row` create a new key ‘ratio’ and assign it a value using the following logic:

```
row['ratio'] = float(row["Population2030"])/
float(row["Population2018"])
```

Problem 3: Car dealership (8 points)

Download cardealership.py file into your Assignment5 folder. In it you will see the following classes that represent cars for sale at various car dealerships: `class Car`, `class CarAttributes`, `class Dealership`.

You will be asked to write two **methods** inside the Dealership class and one **function**.

For the examples below we will be using the following objects.

```
car1 = Car(1, 'Ford', 23000, CarAttributes('Red', 'Rain', 'Level-1'))
car2 = Car(2, 'BMW', 46000, CarAttributes('Blue', 'Regular',
'Regular'))
car3 = Car(3, 'Ferrari', 150000, CarAttributes('Violet', 'Regular',
'Level-2'))
car4 = Car(4, 'Toyota', 26000, CarAttributes('Black', 'Snow',
'Regular'))
car5 = Car(5, 'BMW', 50000, CarAttributes('Red', 'Sport', 'Level-3'))
car6 = Car(6, 'Lotus', 50000, CarAttributes('Grey', 'Sport',
'Regular'))
car7 = Car(7, 'Audi', 40000, CarAttributes('Blue', 'Regular', 'Level-
2'))
car8 = Car(8, 'Audi', 45000, CarAttributes('Blue', 'Rain', 'Regular'))
car9 = Car(9, 'Ford', 30000, CarAttributes('Violet', 'Sport', 'Level-
1'))

dealership1 = Dealership([car1, car2, car3], 'KMac')
dealership2 = Dealership([car4, car5, car6, car7], 'JRM')
dealership3 = Dealership([car8, car9], 'ALee')
```

Note 1: Above you see that a `CarAttributes` object is used as a value to be set into an instance variable in a Car object. This is an example of an object having another object as its component. We would say that a `CarAttribute` object is being *composed* into a `Car` object in this case. This is an example of the concept called *object composition*. It is a common idea used to handle complex objects. A *complex object* is an object consisting of other objects.

Note 2: A special method called `reset_cars` is given to reset the updated values to original values of the object after certain operations have been performed. Do not call this function from inside your own methods or functions!

Task 1: Add a Car to a Dealership (4 points)

In `cardealership.py` file complete the method `add_cars` for the Dealership class. The method takes one argument, cars, which is a *list of lists* of car details. Each list within the cars list represents the details (properties) for a single car. You may assume the entire list is always valid. A details list for a particular car will always be presented in this order: `[id, brand, price, color, tires, trim-level]`. The `id, brand`, and `price` will be stored inside a `Car` object and the other three properties will be stored inside a `CarAttributes` object inside the `Car` object.

Your method should create new `Car` objects and add them to the `car_list` given to you in the `Dealership` class.

Examples:

Consider the following lists of lists of car details:

```
p1List = [
           [11, 'Mercedes', 40000, 'Grey', 'Snow', 'Regular'],
           [12, 'Ford', 20000, 'Red', 'Rain', 'Level-1'],
         ]
p2List = [ ]
p3List = [
           [13, 'Mercedes', 40000, 'Grey', 'Snow', 'Regular'],
           [14, 'Mercedes', 40000, 'Blue', 'Snow', 'Regular'],
           [15, 'Mercedes', 40000, 'Orange', 'Snow', 'Regular'],
         ]
```

Function Call 1 --------------------:

```
  dealership1.add_cars(p1List)
```

Updated dealership1.car_list:

```
KMac:
    Car: [ <1> Ford - 23000 - Attributes: [Red - Rain - Level-1] ]
    Car: [ <2> BMW - 46000 - Attributes: [Blue - Regular - Regular] ]
    Car: [ <3> Ferrari - 150000 - Attributes: [Violet - Regular -
Level-2] ]
    Car: [ <11> Mercedes - 40000 - Attributes: [Grey - Snow -
Regular ] ]
    Car: [ <12> Ford - 20000 - Attributes: [Red - Rain - Level-1] ]
```

Function Call 2 --------------------:

```
  dealership2.add_cars(p2List)
```

Updated dealership2.car_list:

```
JRM:
    Car: [ <4> Toyota - 26000 - Attributes: [Black - Snow - Regular] ]
    Car: [ <5> BMW - 50000 - Attributes: [Red - Sport - Level-3] ]
    Car: [ <6> Lotus - 50000 - Attributes: [Grey - Sport - Regular] ]
    Car: [ <7> Audi - 40000 - Attributes: [Blue - Regular - Level-2] ]
```

Function Call 3 --------------------:

```
  dealership3.add_cars(p3List)
```

Updated dealership3.car_list:

```
ALee:
    Car: [ <8> Audi - 45000 - Attributes: [Blue - Rain - Regular] ]
    CSE 101 - Fall 2017 Lab #11 Page 3
    Car: [ <9> Ford - 30000 - Attributes: [Violet - Sport - Level-1] ]
    Car: [ <13> Mercedes - 40000 - Attributes: [Grey - Snow -
Regular] ]
    Car: [ <14> Mercedes - 40000 - Attributes: [Blue - Snow -
Regular] ]
    Car: [ <15> Mercedes - 40000 - Attributes: [Orange - Snow -
Regular] ]
```

Note: To access the contents of the attributes property of a Car object you need to use the dot operator. For example, suppose car1 refers to a Car object. To change that car's paint color to red we would type this:car1.attributes.paint = 'Red'.

Task 2. Update a Car (4 points)

In `cardealership.py` file complete the method `update_car` in the Dealership class. The method takes the following parameters, in this order:

1. `id`: the ID # of the car to be updated.
2. `new_value`: a tuple containing the detail to be updated and the corresponding value. The tuple will look similar to this: ('brand', 'Dodge'). Any one of the five details can be modified, as identified by one of these strings: `'brand', 'price', 'paint', 'tires', or 'trim'`.

Your function should update the property of the car that matches the id in the given dealership and return 'Updated'. If the id doesn't match and car offered for sale by the dealership, return 'Car not found', capitalized exactly as written. **Note:** No two cars will ever have the same id.

Examples:

Function Call 1 --------------------:

```
dealership1.update_car(1, ('brand', 'Hyundai'))
```

Return Value: `"Updated"`

Updated Dealership:

```
KMac:
    Car: [ <1> Hyundai - 23000 - Attributes: [Red - Rain - Level-1] ]
    Car: [ <2> BMW - 46000 - Attributes: [Blue - Regular - Regular] ]
    Car: [ <3> Ferrari - 150000 - Attributes: [Violet - Regular -
Level-2] ]
```

Function Call 2 --------------------:

```
dealership2.update_car(100, ('paint', 'Red'))
```

Return Value: `"Car not found"`

Updated Dealership:

```
JRM:
    Car: [ <4> Toyota - 26000 - Attributes: [Black - Snow - Regular] ]
    Car: [ <5> BMW - 50000 - Attributes: [Red - Sport - Level-3] ]
    Car: [ <6> Lotus - 50000 - Attributes: [Grey - Sport - Regular] ]
    Car: [ <7> Audi - 40000 - Attributes: [Blue - Regular - Level-2] ]
```

Function Call 3 --------------------:

```
dealership3.update_car(8, ('trim', 'Level-1'))
```

Return Value: `"Updated"`

Updated Dealership:

```
ALee:
    Car: [ <8> Audi - 45000 - Attributes: [Blue - Rain - Level-1] ]
    Car: [ <9> Ford - 30000 - Attributes: [Violet - Sport - Level-1] ]
```