Lab 10 – CSE 101 (Spring 2021)

Objectives

The primary objective of this lab assignment is to get you practice with object-oriented programming in Python

1. Object-Oriented Programming Tutorial (1 point)

Download student.py and use_student.py into your project folder and do the following in the given order.

1. Add the following method to Student class in student

```
def __init__(self, name, id, major, gpa):
    self.name = name
    self.id = id
    self.major = major
    self.gpa = gpa
```

   Note that there is no underscore in front of an instance variable. This is a convention that some people use, but it is not that common.

2. Add the following two lines to main in use_student.py and run it.

```
s1 = student.Student('Amy', 1, 'CS', 3.21)
print('s1:', s1)
```

   Make sure you understand what the `__init__` method is doing in this context. That is, the constructor `Student(...)` call invokes the `__init__` method. Also note what the output looks like. It means that `s1` is an object that is found in a memory location and provides the location (which varies each type you run the program), for example `0x1053d95d0`. That number is a hexadecimal, or base 16, number (so it uses 0-9 and a-f to represent the values). In other words, that is the string representation of the object `s1`. This number may not be that meaningful, so the next step shows a better way to represent the Student when you print it.

3. Add the following method to Student class in student.py and then run the use_student.py program.

```
def __repr__(self):
    return '(' + self.name + ', ' + self.major + ')'
```

   Do you see a better string representation of `s1` printed now? The `print` function requires a string form to display. When a string representation of an object such as `s1` is needed, the Python system calls the special method `__repr__` automatically and use the return value of the method.

4. Add the following two lines to `main` in `use_student.py` and run it.

```
s2 = student.Student('Ken', 2, 'TSM', 3.42)
print('s2:', s2)
```

5. Add the following method to `Student` class in `student.py`.

```
def __eq__(self, other):
    return self.id == other.id
```

6. Add the following two lines to `main` in `use_student.py` and run it.

```
print('s1 == s1:', s1 == s1)
print('s1 == s2:', s1 == s2)
```

   The == operator automatically triggers a call to the `__eq__` method.

7. Add the following method to `Student` class in `student.py`.

```
def __lt__(self, other):
    return self.gpa < other.gpa
```

8. Add the following two lines to `main` in `use_student.py` and run it.

```
print('s1 < s1:', s1 < s1)
print('s1 < s2:', s1 < s2)
```

   The < operator automatically triggers a call to the `__lt__` method.

9. Similarly add code that uses the 'greater than' ('>') operator.
10. The methods that we have added to `Student` so far are called special methods. There are more special methods that you can explore if interested. However, now we will add some regular kind of methods. Add the following method to `Student` class in `student.py`.

```
    def change_major(self, new_major):
        self.major = new_major
```

11. And, add the following two lines to `main` in `use_student.py` and run it.

```
s1.change_major('TSM')
print('s1:', s1)
```

   and verify that `s1`'s major is now changed to `TSM`.

12. Let's add one more regular method. Add the following method to `Student` class in `student.py`.

```
    def change_gpa(self, new_gpa):
        self.gpa = new_gpa
```

13. And, add the following two lines to `main` in `use_student.py` and run it.

```
s1.change_gpa(s1.gpa + 0.3)
print('s1.gpa:', s1.gpa)
```

    and verify that s1's GPA is now changed to a new value. Note how an instance variable is accessed in the main function using a dot notation.

14. We can even create a list of `Student` objects and do something with it. Add the following in the `main` function in `use_student.py` and run it.

```
tsm_majors = [s1, s2]
gpa_sum = 0.0
for s in tsm_majors:
    gpa_sum = gpa_sum + s.gpa
print('Average GPA = ' + str(gpa_sum/len(tsm_majors)))
```

This was a quick tutorial on how to create a class, how to create some objects using the class, and how to use the class in actual code, for example in the main function in `use_student.py`. Now, that you are familiar with this process, let's use it to solve some real problems.

2. Create a Point Class (2 points)

Create a file named `point.py` and follow the instructions below.

1. Define a class named `Point` that will represent a point on a graph. The class will have two instance variable variables for the x and y coordinates. Create a constructor method (recall this is must be named `__init__`) that allows you to pass the x and y coordinates as arguments to the constructor, for example:

```
>>> p1 = Point(1,1)
>>> p2 = Point(4,5)
```

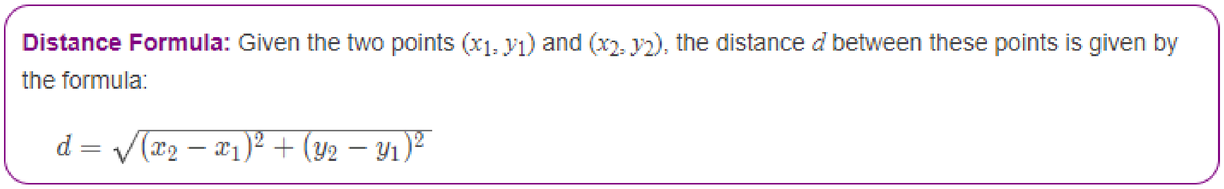
Next, you will write some additional methods in your class (the examples refer to the two points p1 and p2 shown above):

2. Write a `__repr__` method that displays the point in standard mathematical notation, for example:

```
>>> p1
(1,1)
```

3. Write a `distance` method that compute the distance between two points, for example:

```
>>> p1.distance(p2)
5.0
```


Distance Formula: Given the two points (x_1, y_1) and (x_2, y_2), the distance d between these points is given by the formula:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Read more: https://www.purplemath.com/modules/distform.htm

4. Write a `polar` method that returns a pair of values (tuple) corresponding to the polar coordinates of the point:

```
>>> p1.polar()

(1.4142135623730951, 0.7853981633974483)
```

The polar coordinates of a point (x, y) are a pair of numbers (r, q) where r = sqrt(x^2 + y^2) and q = $\tan^{-1}$ y/x

You can use Python's math, which has a function named atan that computes $\tan^{-1}$. The formula for polar coordinates is valid only if the x-coordinate of a point is greater than 0. The polar method should return None if x is negative or 0.

You can read more at: http://tutorial.math.lamar.edu/Classes/CalcII/PolarCoordinates.aspx

3. Car Dealership Program (2 points)

Download dealership.py. In this file 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.

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], 'Seoul Auto')
dealership2 = Dealership([car4, car5, car6, car7], 'Incheon Cars')
dealership3 = Dealership([car8, car9], 'Busan Vehicles')
```

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!

Part 1. Add a Car to a Dealership

In dealership.py, 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`:

```
Seoul Auto:
    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`:

```
Incheon Cars:
    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`:

```
Busan Vehicles:
    Car: [ <8> Audi - 45000 - Attributes: [Blue - Rain - Regular] ]
    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'`.

Part 2. Update a Car

In `dealership.py`, 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 any car offered for sale by the dealership, return `'Car not found'`. **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:

```
Seoul Auto:
    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:

```
Incheon Cars:
    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:

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

4. Submission

Submit your completed `student.py`, `use_student.py`, `point.py`, and `dealership.py` programs on Blackboard.