Object Oriented Programming in Python

Exercise 0

Implement a "SquareManager" that is able to evaluate the perimeter, the area and the diagonal of a square given the lenght of it's side.
The result should be somehting like the following

```
from Exercise_0 import *
if __name__=="__main__":
    sm=SquareManager(3)
    print(f"The area of the square with side {sm.side} = {sm.area()}")
    print(f"The perimeter of the square with side {sm.side} = {sm.perimeter()}")
    print(f"The diagonal of the square with side {sm.side} = {sm.diagonal():.3f}")
```

Exercise 1

Implement a "Point" class that is able to represent 2-D points. The class must contains the methods to obtain the results shown below

```python
from Exercise_1 import *
if __name__=="__main__":
    # create 2 points and calculate distance among them
    a=Point(7,1)
    b=Point(1,1)
    print(a.distance(b))

    # move a point according to a vector (x,y)
    a.move(2,2)
    print(a)
```

Tips and tricks:

- The formula for the distance betwees two points is

$$d(P_1, P_2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

- To use the square root function you need to import math and use math.sqrt like below

```python
import math
print(math.sqrt(4))
```

Exercise 2

Implement a "Line" class that is able to represent 2-D lines.
Using the class "Points" of the previous exercise and this one try to obtain the following results

```python
from Exercise_2 import *
if __name__=="__main__":
    # 1 Simple creation
    l1=Line(m=3,q=2)
    print(l1)
    #2 Create line from 2 points
    a=Point(0,1)
    b=Point(2,2)
    l2=Line()
    l.line_from_points(a,b)
    print(l2)
    #3 Function for distance from point and intersection with another line
    l=Line(m=1,1=0)
    a=Point(1,5)
    print(l.distance(a))
    m=Line(-1,0)
    i=l.intersection(m)
    print(i)
```

Tips and tricks

- The formula to obtain the equation from 2 points is
  $$y = x(y_2 - y_1)/(x_2 - x_1) - x_1(y_2 - y_1)/(x_2 - x_1) + y_1$$
- The formula for the distance between a line and a point is
  $$d(P, r) = \frac{|ax_P + by_P + c|}{\sqrt{a^2 + b^2}}$$
- the formula to find the intersection is
  $$P\left(\frac{q_2 - q_1}{m_1 - m_2}, m_1\frac{q_2 - q_1}{m_1 - m_2} + q_1\right)$$

Interact with your script

In all the exercises we made before we were able to run the main just once but in most cases that's not really useful, we would like to create a menu for our main n order to be able to execute the same code in different ways according to some commands and then quit when we've finished.
An easy method to do this is shown below for the Exercise 1.
This we'll be useful in the next exercises!

```
if __name__=="__main__":
    #We want to run the script untile the user input a specific command to quit
    #(in this case "q")
    while True:
        #Putting \ in a string allows us to continue writing in the next line
        user_input=input("The available command are:\n\
    d:Distance between 2 points\n\
    m:Move a point according to a vector\n\
    q:Quit\n")
        if user_input=="d":
            #Ask the user two points and calculate the distance
            pass
        elif user_input=="m":
            #Ask the user a point and a vector to apply
            pass
        elif user_input=="q":
            #exit from the loop
            break
        else:
            print("Command not recognized")
    print("Goodbye!")
```

Exercise 3

In this exercise we wanto to create a "Deck" class and a "Card" class to represent a poker deck of cards. The requirements are

1. The Deck should have a *deal()* method that pick a card (or multiple card from the deck) and remove them from the deck
2. The Deck should have a *shuffle()* method
3. The Card class should have a suit (Hearts, Diamonds, Clubs, Spades) and a value (A,2,3,4,5,6,7,8,9,10,J,Q,K)

Tips and tricks

- Pay attention to consider the case of trying to draw more cards than the actual number of cards in the deck.
- In the Python package *random* there is a function thay may help you to shuffle the deck, look here

Exercise 4

Do this exercise after the theory lesson on the dataformat

Create the classes "Contact" that must be able to store the contact presents in the file *"contacts.json"*

```
{
    "name": "Cassio",
    "surname":"Zen",
    "email": "cassiozen@gmail.com"
}
```

Exercise 5

Do this exercise after the theory lesson on the dataformat

Using the class "Contact" fromt he previous exercise and "AddressBook".

The "AddressBook" class must be able to read the content of the file and perform CRUD (Create,Read,Update,Delete). The Update is the most difficult so i suggest to begin with the other. Below you can find an example

```python
from Exercise_5 import *
book=AddressBook() #create an object of the class
book.show() #show all the contacts
book.find_by_name('Dan') #find a contact
book.remove_contact('Dan') #remove a contact according to his name
book.add_contact('Peter','Parker','notspiderman@marvel.com') #add a contact
```

Once you've done this you can now create a client to use the functions you implemented in a "user-frienldy way"
The result should be something like the following

```
Welcome to the application to manage your contacts
Press 's' tho show the list of contacts
Press 'n' to add a contact
Press 'f' to find a contact
Press 'd' to delete a contact
Press 'q' to quit
```

Exercise 6

Do this exercise after the theory lesson on the dataformat

Create the function to evaluate:

1. The average **ratings** among the players
2. The average **height** and **weigth** (in meters and kilograms while they're store in inches and pounds)
3. The average age

The file playerNBA.json contains the list of all the NBA player of this season with their stats and their bio. Each one looks like this:

```
{
  "pos": "G",
  "name": "Stephen Curry",
  "hgt": 75,
  "tid": 9,
  "born": {
    "year": 1988,
    "loc": "Akron, OH"
  },
...
}
```

Inheritance

Inheritance is used to create new classes by using existing classes. New ones can both be created by extending and by restricting the existing classes. Let's see a simple example below

```python
class Animal(object):
    """docstring for Animal"""
    def __init__(self, name):
        self.name=name
    #This method is used to define how an object of this class will be represented by the function print()
    def __repr__(self):
        return f"{self.name} is an animal"
    def jump(self):
        print(f"{self.name} has jumped")

class Quadruped(Animal):
    """docstring for Quadruped"""
    def __init__(self,name):
        Animal.__init__(self,name)
        self.n_of_legs=4
    #This method is used to define how an object of this class will be represented by the function print()
    def __repr__(self):
        return "{} is a quadruped".format(self.name)


class Dog(Quadruped):
    """docstring for Dog"""
    def __init__(self, name,family):
        Quadruped.__init__(self,name)
        self.family=family
        self.verse="Woof!"
    #This method is used to define how an object of this class will be represented by the function print()
    def __repr__(self):
        return "{} is a dog of the family {}".format(self.name,self.family)
    def talk(self):
        print(f"{self.name} says: {self.verse}")
```

We can test it woth the following script:

```
if __name__ == '__main__':
    pongo=Animal('Pongo')
    print(pongo) # Pongo is an animal
    pongo.jump() # Pongo has jumped

    pongo=Quadruped('Pongo')
    print(pongo)  # Pongo is a quadruped
    pongo.jump() # Pongo has jumped

    pongo=Dog('Pongo','Dalmata')
    print(pongo) # Pongo is a dog of the family Dalmata
    print(pongo.n_of_legs) # 4
    pongo.jump()  # Pongo has jumped
    pongo.talk() # Pongo says: Woof!
```

As you can see according to the way in we instantiate the object we can acces to different methods and we can always use the method of the parent-class from the child class

Polymorphism

Polymorphism allows us to use method of parent classes and modify them according to our need. An example of this was alredy shown in the code above as we use the method "*__repr__*" to define how our object will be printed.
Another example can be done for the Exercise 1: imagine we want to define a Point2D and a Point3D and we want to calculate their distance from the origin, we could do as follows

```python
import math
class Point2D(object):
    """Class for a Point object: the available methods are move(x,y) and distance(another_point)"""
    def __init__(self, x,y,):
        self.x = x
        self.y = y
    def __repr__(self):
        return "{},{}".format(self.x,self.y)
    def distanceFromOrigin(self):
        print (math.sqrt((self.x)**2+(self.y)**2))
class Point3D(Point2D):
    def __init__(self,x,y,z):
        Point2D.__init__(self,x,y)
        self.z=z
    def __repr__(self):
        return "{},{},{}".format(self.x,self.y,self.z)
    def distanceFromOrigin(self):
        print (math.sqrt((self.x)**2+(self.y)**2+(self.z)**2))
        #To maximize the potentiality of polymorphism we could do
        #distance=math.sqrt(Point2D(self.x,self.y).distanceFromOrigin()**2+self.z**2)
if __name__=="__main__":
    a=Point2D(1,1)
    a.distanceFromOrigin() # 1.4142135623730951
    b=Point3D(1,1,1)
    b.distanceFromOrigin() # 1.7320508075688772
```

Exercise 7

As exercise on polymorpishm and inheritance you can create the parent-class "Circle"and the child-class "Cylinder". The Circle class must have the methods to calculate area and perimeter while the Cylinder class must have the method to calculate area and volume

Encapsulation

When programmin using classes we sometimes want to avoid the final user to execute some method or access some attributes to avoid possible mistakes and problems. We can doing this using encapsulation. Let's make and example considering the class "*Student*" defined below

```python
class Student(object):
    def __init__(self,name,surname,birthday):
        self.name=name
        self.surname=surname
        self.birthday=birthday
if __name__=="__main__":
    s=Student("Mickey","Mouse","16/01/1928")    #Create a student
    print(s.name+" "+s.surname) #Print his full name
    s.surname="Duck"#Change the student's surname and print it again
    print(s.name+" "+s.surname)
```

As you see we can access to the attributes of the class without any restriction. If we want instead to restrict the access to the attribute *name* and *surname* to avoid the possibility to modify these after the object creation, the attribute birthday instead well have the same behaviour. To achieve this we would do as follows

```python
class Student(object):
    def __init__(self,name,surname,birthday):
        self.__name=name
        self.__surname=surname
        self.birthday=birthday
    def getFullName(self):
        self.__name+" "+self.__surname
```

If in this case we try to access the attributes __*name* or __*surname* we would get an error. The only way to obtain the complete name would be to use the method *getFullName()*. So to use encapsulation in Python and hide some attributes to the final user we must simpply put "__" before the name its name (the same can be done to hide methods)

Exercise 9

As exercise to exploit encapsulation let's create a class "*Car*", the class must have the attributtes for name, speed and gear with the following properties

1. the *name* can't be changed after the creation of the object, but it can be retrieved
2. the *speed* can be set to any value less than 250 (if greater it will be changed to 250) and can be retrieved
3. the *gear* can go from 1 to 6 and can be changed only up and down by one (it's set to 1 when the object is created) and can be retrieved