MCT-243 : COMPUTER PROGRAMMING-II
using Python 3
Object Oriented Programming

Prepared By:
Mr. Muhammad Ahsan Naeem

YouTube Playlist
https://youtube.com/playlist?list=PLWF9TXck7O_zuU2_BVUTrmGMCXYSYzjku

Lab 33: Object Oriented Programming

This Lab Session will start with the tasks related to OOP concepts covered in previous Lab Session.

Tasks:

[1] Upgrade the student class so that after registering the subject(s) the list of registered subjects should get sorted. Create an instance of class, add a few subjects and display them to verify if they are sorted.

[2] For the same student class, create a list of few students with different registered subject. Sort the list based on number of subjects registered

[3] A **Point** on a cartesian plane has two co-ordinates as x-coordinate and y-coordinate. Do the followings:

a. Define a class named as `Point`.
b. A Point class instance should have two data attributes. Define those attributes as `x` and `y` with default value of `0` for both.
c. Define an instance method named as `reset()`. It will not have any input argument (except `self`) and it should reset the point to origin i.e. `0` for both `x` and `y` coordinates.
d. Define an instance method named as `move()`. It will have two input arguments as `x` and `y` coordinates and will set the `xy` coordinates of the point to these inputs.
e. Update the `reset()` method created at in (c) so that it uses `move()` method to reset the coordinates.
f. Define two instance methods named as `xTranslate()` and `yTranslate()`. These methods will take one input argument and will move the point instance from its current position to a new position which is input units towards `x` direction in case of `xTraslate()` and towards `y` direction in case of `yTranslate()`.
g. Define an instance method named as `showPoint()` that will display the points as `(x,y)` on the screen.
h. Define an instance method named as `distOrigin()` that will calculate the distant of the point from origin.

Use all above methods in main program to test their correct working. You can test these methods in main program by calling these on `Point` objects. An example test code is given here:

```
from Point import Point
p1=Point(2,3)
p2=Point()
p3=Point(5,5)
print(p2.showPoint()) #Should print (0,0)
print(p3.showPoint()) #Should print (5,5)
p3.reset()
print(p3.showPoint()) #Should print (0,0)
p2.move(10,10)
print(p2.showPoint()) #Should print (10,10)
p2.xMove(-5)
print(p2.showPoint()) #Should print (5,10)
p2.yMove(5)
print(p2.showPoint()) #Should print (5,15)
```

Class Variables (more on it):

Here is the **MechaStudent** class we created last week except that it contains one more instance method named as **welcome** and is self-explanatory.

```
class MechaStudent:
    'This class defines a Mechatronics Student and associated methods
 on it.'
    department='Mechatronics'
    offSubjects=['Mech','LA','ES','CP2','MOM','Isl/Pak']
    def __init__(self,name,reg,sec):
        self.name=name
        self.reg=reg
        self.sec=sec
        self.email=reg+'@mct.uet.edu.pk'
        self.courses=[]
    def welcome(self):
        return self.name+' Welcome!'
    def regSubj(self,*subj):
        for i in subj:
            if i in MechaStudent.offSubjects:
                self.courses.append(i)
            else:
                raise ValueError(f'{i} is not offered.')
        self.courses.sort()
```

Another additional part is the starting doc string which is used for brief description of the class as we did in case of user-defined functions.

In **MechaStudent** class we created two class variables; one is shown here:

```
offSubjects=['Mech','LA','ES','CP2','MOM','Isl/Pak']
```

A class variable is accessible via class or the instance. We used it in one instance method **regSubj()** as:

```
    def registerSubj(self,*subj):
        for i in subj:
            if i in MechaStudent.offSubjects:
                self.subjects.append(i)
```

We used it via class. We could use it via instance as:

```
    def registerSubj(self,*subj):
        for i in subj:
            if i in self.offSubjects:
                self.subjects.append(i)
```

Are the two methods of accessing class variable same? The two ways are shown again here:

```
MechaStudent.offSubjects
```

```
self.offSubjects
```

Generally, both will give the same result but as described earlier that a class variable can be accessed via the instance and there is also a possibility that we change the value for one instance while the value of class variable is unchanged. That can result into different outputs for the above two line. See the code below for better understanding:

```
std1=MechaStudent('MUHAMMAD USMAN','2018-MC-01','A')
std2=MechaStudent('DANISH','2018-MC-51','B')
std1.offSubjects=['EC','EDC']
```

For above case **`MechaStudent.offSubjects`** will refer to original list defined as Class variable but **`self.offSubjects`** will refer to **`['EC','EDC']`** for **`std1`** and it will refer to original Class variable for **`std2`**.

__dict__ attribute:

We studied earlier that when a class is created though it's empty, Python creates and sets a few things at back end. One of such things is **`__dict__`** data attribute which is set for class and any instance of the class. As the name suggest that it is magical (magically created!) and is a dictionary. This attribute holds the information of class (as class attribute) and the instance (as instance attribute). This is really helpful for the debugging purposes. For lengthy classes, it will be difficult to go through the code and identify different class attributes or to see different instance attributes of the instance.
With the above class defined we can use it on class as:

```
from MechaStudent import MechaStudent
print(MechaStudent.__dict__)
```

You will see all key-values inside **`__dic__`** attribute. To view them in a better format, we can use it as:

```
from MechaStudent import MechaStudent
for item in MechaStudent.__dict__.items():
    print(item)
```

Here is the output:

```
('__module__', 'MechaStudent')
('__doc__', 'This class defines a Mechatronics Student and associated
method on it')
('department', 'Mechatronics')
('offSubjects', ['Mech', 'LA', 'ES', 'CP2', 'MOM', 'Isl/Pak'])
('__init__', <function MechaStudent.__init__ at 0x000001EF2D685F78>)
('welcome', <function MechaStudent.welcome at 0x000001EF2D692C18>)
('regSubj', <function MechaStudent.regSubj at 0x000001EF2D68A3A8>)
('__dict__', <attribute '__dict__' of 'MechaStudent' objects>)
('__weakref__', <attribute '__weakref__' of 'MechaStudent' objects>)
```

You can see that it prints the detail of the class including all its data attributes and method attributes. A few things are there that you might not understand (e.g. **`__weakref__`**) but you can just ignore that at this moment.
Now let's check this attribute for the instance of the class as shown here:

```
from MechaStudent import MechaStudent
std1=MechaStudent('Usman','2018-MC-01','A')
std2=MechaStudent('Danish','2018-MC-51','B')
std1.gpa=3.2
for item in std1.__dict__.items():
    print(item)
```

A new attribute **gpa** is added to the **std1** instance. Here is the output:

```
('name', 'Usman')
('reg', '2018-MC-01')
('sec', 'A')
('email', '2018-MC-01@mct.uet.edu.pk')
('courses', [])
('gpa', 3.2)
```

The output is simple to understand but one important thing to notice is that there is no data attribute as **'department'** for the instance. Although, we saw that we can access the class attribute via the instance. A simple case can be:

```
print(std1.department) #It will print Mechatronics
```

But this attribute not available in **__dic__**.

Basically, the way things are working is that the attribute **department** is not an instance attribute but a class attribute. But when we try to access this attribute as **std1.department** where **std1** is an instance, the Python interpreter searches for this attribute within the instance namespace (present inside **__dict__**) first. If that is found there, interpreter will get the value from there. However, if not present in instance namespace, Python interpreter will move to search it from class namespace and in above case it finds it there with the value **Mechatronics**.
Now just to make things more clear to yourself, add **department** attribute to one of the instances and set a different value to it as shown here:

```
from MechaStudent import MechaStudent
std1=MechaStudent('Usman','2018-MC-01','A')
std2=MechaStudent('Danish','2018-MC-51','B')
std1.department='Civil'
print(std1.__dict__)
print(std2.__dict__)
```

You will see that **department** is added as attribute of **std1** with value **Civil** and is not there for **std2**. While trying to access department for **std1**, it will be picked from its namespace with value **Civil** and in case of **std2** it will be picked from class variable with value **Mechatronics**.

Method as Data Attribute (or property):

We have seen the **__init__()** method of **MechaStudent** class where the **email** attribute is initialized without passing in the **email** as input argument because **reg** attribute value is picked and set as **email** after appending appropriate domain. The code is given here again:

```
def __init__(self,name,reg,sec):
        self.name=name
        self.reg=reg
        self.sec=sec
        self.email=reg+'@mct.uet.edu.pk'
        self.courses=[]
```

On the similar approach with **Point** class, we want to set distance from origin as data attribute and that should be generated from x-y components rather than passing the value by ourselves. The **__init__()** function for the class is given here:

```
from math import sqrt
class Point:
    def __init__(self,x=0,y=0):
        self.x=x
        self.y=y
        self.distOrigin=sqrt(self.x**2+self.y**2)
```

An example use of the class is shown here:

```
from Point import Point
p1=Point(2,3)
print(p1.distOrigin)
```

It will have the following output:

```
3.605551275463989
```

So far, this approach is working well. But let's explore this with a little variety.
Consider again the **MechaStudent** class where an instance is created, and a couple of data attributes are printed as shown here:

```
from MechaStudent import MechaStudent
std1=MechaStudent('Usman','2018-MC-01','A')
print(std1.reg)
print(std1.email)
```

which will have following; very obvious, output:

```
2018-MC-01
2018-MC-01@mct.uet.edu.pk
```

But if we change the reg attribute of the student to some new registration number, as we have cases of re-admission where student gets new registration number, will this affect the **email** attribute? The attribute **email** is generated from attribute **reg**, so by updating **reg**, will **email** get updated?
Let's explore this with the following code:

```
from MechaStudent import MechaStudent
std1=MechaStudent('Usman','2018-MC-01','A')
print(std1.reg)
print(std1.email)
std1.reg='2019-R/2018-MC-01'
```

```
#After change in registration
print(std1.reg)
print(std1.email)
```

The above code has following output:

```
2018-MC-01
2018-MC-01@mct.uet.edu.pk
2019-R/2018-MC-01
2018-MC-01@mct.uet.edu.pk
```

You can see that the attribute **reg** is updated but the attribute **email** is still having the old value.

Why is that so?

Because the method **__init__()** is called only once at the time of instantiation of the object and at that time email attribute was set to **2018-MC-01@mct.uet.edu.pk**. But after the **reg** attribute is updated, the method **__init__()** is not called and hence **email** attribute will remain same as previous.

Is it good or bad?

To answer this, first decide whether we want to change the **email** of a student after his/her registration number is updated? Most probably we don't want that. So, for this case it seems a good feature but what about **distOrigin** attribute of **Point** instance? When a point is instantiated, it will get **distOrigin** value calculated and assigned to it. Later, if the point is reset or moved and we want to access **disOrigin**, it would still be the same old one. In this case this is absolutely not something we want. So, let's see the ways we can handle this.

For understanding purpose, let's consider the **MechaStudent** class and assume that we want to update **email** attribute once the **reg** attribute is updated although the new email generated is not a valid email format. To do that we can set **email** as method attribute rather than the data attribute (This is valid for all class example we studied in lecture where we said the attribute can be data or method). The code with this approach is given here:

```
    def __init__(self,name,reg,sec):
        self.name=name
        self.reg=reg
        self.sec=sec
        self.courses=[]
        self.email=self.reg+'@mct.uet.edu.pk'
    def setEmail(self):
        self.email=self.reg+'@mct.uet.edu.pk'
```

You can see that the email is set as attribute in **__init__()** method but there is also another method **setEmail()** that will simply re-execute the same statement. (We cannot have name of a method same as of an attribute). Now we will have to call this method to update **email** each time **reg** attribute is updated. Example is shown here:

```
from MechaStudent import MechaStudent
std1=MechaStudent('Usman','2018-MC-01','A')
print(std1.reg)
```

```
print(std1.email)
std1.reg='2019-R/2018-MC-01'
#After change in registration
std1.setEmail()
print(std1.reg)
print(std1.email)
```

Now you will see updated email.

Another very similar approach can be not to have **email** as attribute but as method that simply return the email instead of assigning it as attribute. The approach is shown here:

```
    def __init__(self,name,reg,sec):
        self.name=name
        self.reg=reg
        self.sec=sec
        self.courses=[]
    def email(self):
        return f'{self.reg}@mct.uet.edu.pk'
```

Now we have email as method attribute and we will have to use it as method. Example program is given here:

```
from MechaStudent import MechaStudent
std1=MechaStudent('Usman','2018-MC-01','A')
print(std1.reg)
print(std1.email())
std1.reg='2019-R/2018-MC-01'
#After change in registration
print(std1.reg)
print(std1.email())
```

The best solution is yet to come.

The best solution for such cases is to have a method as in above code but make it a data attribute (also known as property) via function decorator. This solution is literally very simple. All we need is to add **@property** decorator to the method **email** (of above code) and that method will become a data attribute. And whenever there is some change in the attribute(s) on which this decorated method depends (e.g. **reg** in above case), this method will get called automatically and the attribute will get updated. See the code given here very carefully:

```
    def __init__(self,name,reg,sec):
        self.name=name
        self.reg=reg
        self.sec=sec
        self.courses=[]
    @property
    def email(self):
```

```
        return f'{self.reg}@mct.uet.edu.pk'
```

Now **email** apparently is a method but because of property decorator it is actually a data attribute which will get updated automatically whenever **reg** attribute is updated. See the code here where **email** is treated as data attribute:

```
from MechaStudent import MechaStudent
std1=MechaStudent('Usman','2018-MC-01','A')
print(std1.reg)
print(std1.email)
std1.reg='2019-R/2018-MC-01'
#After change in registration
print(std1.reg)
print(std1.email)
```

Another very big advantage of this approach is a that we don't need to change anything in the main program which was using our class. It is still the same as the first version where **email** was added as data attribute in **__init__()** method. So, if we choose to add email as new method then we would have to change all occurrences of **email** to **email()** in the main program. But in this final approach, there is nothing to change in main program.

For now, remove the property decorator and make **email** as data attribute inside **__init__()** method as we don't want to update the **email** once the **reg** is updated. We will use this idea in next task.

Tasks:

In the **Point** class add **distOrigin** as data attribute but via property decorator because of all good reasons described above.