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 38: Object Oriented Programming

Multiple Classes

In this lab, we will see how to deal with multiple classes with the possibility of dependency of one class attribute on another class.

Let consider one class **Author** to represent an author. Here is the basic class code and a sample use of the class:

```
class Author:
    def __init__(self,name,age):
        self.name=name
        self.age=age
        self.books=[]
    def __repr__(self):
        return f'{self.name}-{self.age}'

## Main Program ##
auth1=Author('Ahsan Naeem',34)
auth2=Author('Rzi Abbas',30)
auth3=Author('Misbah-ur-Rehman',30)
print(f'{auth1}={auth1.books}')
print(f'{auth2}={auth2.books}')
print(f'{auth3}={auth3.books}')
```

You can see there is one property **books** and the initial value of the book property is an empty list. Later on, we want to insert books of that author in this list. So, let's create another class **Book** for the book object as shown here:

```
class Author:
    def __init__(self,name,age):
        self.name=name
        self.age=age
        self.books=[]
    def __repr__(self):
        return f'{self.name}-{self.age}'
class Book:
    def __init__(self,title,year,authors=None):
        self.title=title
        self.year=year
        if(authors is None):
            self.authors=[]
```

```
        else:
            self.authors=authors
    def __repr__(self):
        return f'{self.title}-{self.year}'


## Main Program ##
auth1=Author('Ahsan Naeem',34)
auth2=Author('Rzi Abbas',30)
auth3=Author('Misbah-ur-Rehman',30)
print(f'{auth1}={auth1.books}')
print(f'{auth2}={auth2.books}')
print(f'{auth3}={auth3.books}')
b1=Book('Python Programming',2020,[auth1,auth2])
b2=Book('OOP in Python',2020,[auth1,auth3])
print('_________________')
print(f'{auth1}={auth1.books}')
print(f'{auth2}={auth2.books}')
print(f'{auth3}={auth3.books}')
```

See the class **Book** has one property **authors** and the default value is **None**. Then inside the **__init__** method we are assigning the value as empty list, if the authors list is not provided and otherwise the the value of authors is a list if authors i.e., the **Author** class objects.
In the Main program you can see that after creating the three authors, two books are also created and a list of author objects is passed for both books. When the authors **books** property is printed afterwards, you will still see the empty list of books for the authors.

Why is that so?

Because we have not created any link between the **Author** and the **Book** class. Now let's create that link by creating the **getter** and **setter** for the **authors** attribute of the **Book** class as shown here:

```
class Author:
    def __init__(self,name,age):
        self.name=name
        self.age=age
        self.books=[]
    def __repr__(self):
        return f'{self.name}-{self.age}'
class Book:
    def __init__(self,title,year,authors=None):
        self.title=title
        self.year=year
        if(authors is None):
            self.authors=[]
        else:
```

```
            self.authors=authors
    @property
    def authors(self):
        return self._authors
    @authors.setter
    def authors(self,newauthors):
        for a in newauthors:
            if(not isinstance(a,Author)):
                raise TypeError
            else:
                a.books.append(self)
        self._authors=newauthors
    def __repr__(self):
        return f'{self.title}-{self.year}'

## Main Program ##
auth1=Author('Ahsan Naeem',34)
auth2=Author('Rzi Abbas',30)
auth3=Author('Misbah-ur-Rehman',30)
print(f'{auth1}={auth1.books}')
print(f'{auth2}={auth2.books}')
print(f'{auth3}={auth3.books}')
b1=Book('Python Programming',2020,[auth1,auth2])
b2=Book('OOP in Python',2020,[auth1,auth3])
print('________________')
print(f'{auth1}={auth1.books}')
print(f'{auth2}={auth2.books}')
print(f'{auth3}={auth3.books}')
```

The important part is the **setter** method and specially the following part:

```
for a in newauthors:
     if(not isinstance(a,Author)):
          raise TypeError
     else:
          a.books.append(self)
```

The **if** condition is checking if the author provided is actually not the **Author** class object then it should raise an exception and in the **else** clause, the book i.e. **self** is appended into the **books** property of that author. And now if you will run the same Main Program, you will be able to see that once a **Book** is created and authors are specified, that book will be added to the **books** property of the authors automatically. This is the output of the program:

```
Ahsan Naeem-34=[]
Rzi Abbas-30=[]
Misbah-ur-Rehman-30=[]

________________
Ahsan Naeem-34=[Python Programming-2020, OOP in Python-2020]
```

```
Rzi Abbas-30=[Python Programming-2020]
Misbah-ur-Rehman-30=[OOP in Python-2020]
```

Enumeration Class

An **Enumeration** is a special class that represents a group of constants. It comes very handy and let's see the detail of the Enumeration class.

Note→ There is a Python function `enumerate()` that must not be confused with the **Enumeration** class.

Consider this `Instructor` class which is not the **Enumeration** class but a simple class:

```
class Instructor:
    desigList=['Prof', 'Assoc Prof', 'AP', 'Lect','TF']
    def __init__(self,fName,lName,desig):
        self.fName=fName
        self.lName=lName
        self.desig=desig
        self.courses=[]
    def __repr__(self):
        return f'{self.fName} {self.lName} - {self.desig}'
```

The list of designations consists of designations as initials:
Prof → Professor
Assoc Prof → Associate Professor
AP → Assistant Professor
Lect → Lecturer
TF → Teaching Fellow

Let's create getter and setter for the `desig` property and use that in Main Program as:

```
class Instructor:
    desigList=['Prof', 'Assoc Prof', 'AP', 'Lecturer','TF']
    def __init__(self,fName,lName,desig):
        self.fName=fName
        self.lName=lName
        self.desig=desig
        self.courses=[]
    def __repr__(self):
        return f'{self.fName} {self.lName} - {self.desig}'

    @property
    def desig(self):
        return self._desig
    @desig.setter
```

```
    def desig(self,newdesig):
        if(newdesig in Instructor.desigList):
            self._desig=newdesig
        else:
            raise ValueError(f'{newdesig} is not a valid
Designation!')
## Main Program ##
i1=Instructor('Ali','Raza','Assoc Prof')
print(i1.desig)
```

The program is working fine but in the use of the class, we will have to remember all designation names used in `desigList`. And if we want to have the full names of all designations, then we will have to pass those lengthy names every time. It's here where we can use an Enumeration Class for the designation and one designation will be an object of the `Designation` class. This is very different class as compared to the normal class.
The objects of the Enumeration class are known as **members**. They have **names** and **values**. To make a class as **Enumeration Class**, it should be child of the class `enum` available in the `Enum` module. We have not studied the concept of **Child** class but at this stage we just need to see how a class is declared as a child class of another class and that is simply by specify the **Parent** class name inside the `()` while defining the child class.

The Enumeration Class for `Designation` is shown here:

```
from enum import Enum
class Desig(Enum):
    Prof='Professor'
    AssocProf='Associate Professor'
    AP='Assistant Professor'
    Lect='Lecturer'
    TF='Teaching Fellow'
```

Unlike the normal class we can create an object of **Enumeration** class using the **Dot Notation** as shown here:

```
d=Desig.Prof
print(type(d))
```

The output will be:

```
<enum 'Desig'>
```

We can access the **name** and the **value** of the `Desig` object using `.name` and `.value` as shown here:

```
d=Desig.Prof
print(d.name)    # Will print Prof
print(d.value)   # Will print Professor
```

Other than many useful features of the **Enumeration** class, one advantage is that we can use the short names while coding and can convert those to full names e.g. we can use **`Prof`** while coding and the value assigned will be **`Professor`**. Moreover, we get the intellisense help while coding.

We can access all members of an Enumeration Class using **`__members__`** attribute and we can also access one member of the Enumeration Class by passing the name of the member. See the use here:

```
print(Desig.__members__)
print(Desig['TF'])
print(Desig['TF'].name)
print(Desig['TF'].value)
```

Now let's use this Enumeration Class with the **`Instructor`** class as shown here:

```
from enum import Enum
class Desig(Enum):
    Prof='Professor'
    AssocProf='Associate Professor'
    AP='Assistant Professor'
    Lect='Lecturer'
    TF='Teaching Fellow'
class Instructor:
    def __init__(self,fName,lName,desig:Desig):
        self.fName=fName
        self.lName=lName
        self.desig=desig
        self.courses=[]
    def __repr__(self):
        return f'{self.fName} {self.lName} - {self.desig}'

    @property
    def desig(self):
        return self._desig.value
    @desig.setter
    def desig(self,newdesig):
        self._desig=newdesig

## Main Program ##
i1=Instructor('Ali','Raza',Desig.AssocProf)
print(i1.desig)
```

Note the following points:

- The **`desigList`** inside the **`Instructor`** class has been removed since we will set the designation using the Enumeration Class, **`Designation`**.

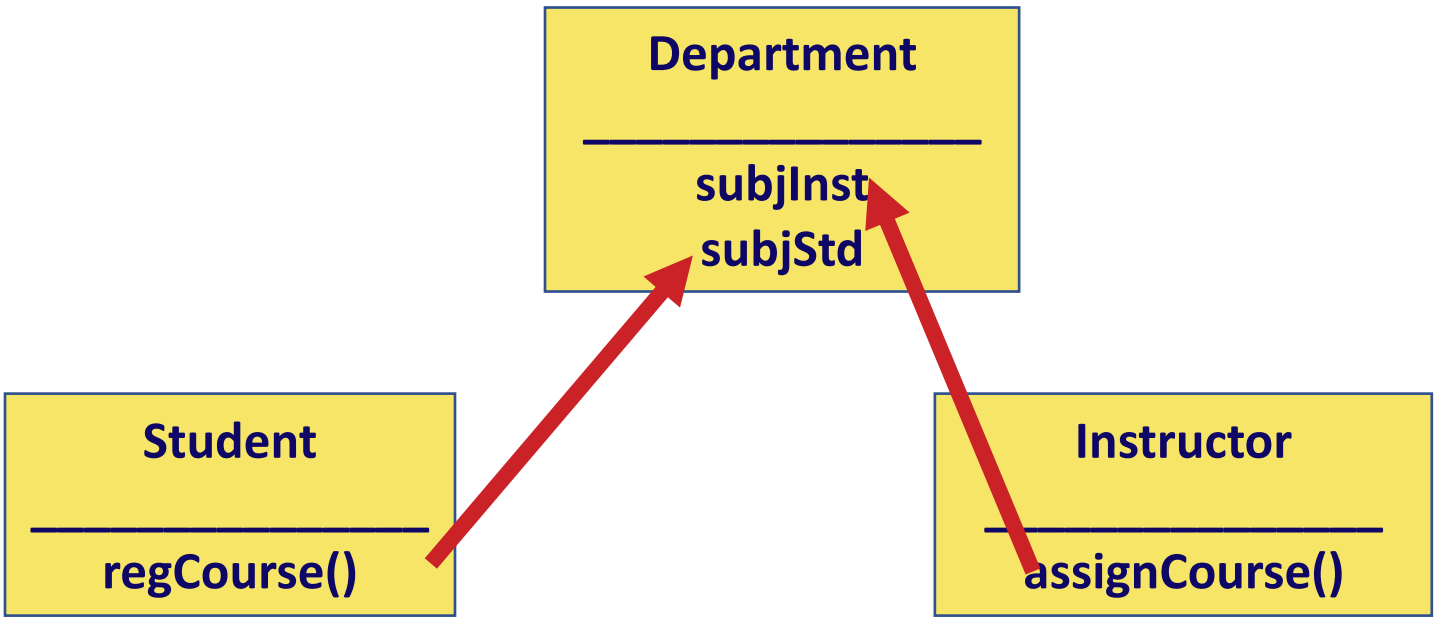
- The input argument `desig` in the `__init__` method now indicates that it will be `Desig` class object as: `desig:Desig`. This is just the convention for better readability of the code.
- Inside the `setter` method, the designation is passed directly (assuming that it will be a valid Designation Enumeration class member) and inside the getter method the `value` of that member is returned.
- Check the Main Program that the designation is passed as `Desig` Class member, but when the property is accessed, it will return the `value` of the member.
- If you want to include the provision that `desig` could be passed as string and still getting the same results, please refer to the video provided in the references.

Multiple Classes

Now once again, let's come back to the multiple classes scenario. We already had the `MechaStudent` class. Now we will create a class for the `Instructor` and one for the `Department`. The students can register courses and the instructor are assigned the courses. Both are linked together through `Department` class. The `Department` class will simply have two class level attributes `subjInst` and `subjStd`. Both will be the dictionaries. `subjInst` will have keys as Subject names and value as list of instructors teaching that course. `subjStd` is the dictionary with keys as subject names and value as list of students registered in that course.

In `MechaStudent` class, a student can register a course using `regCourse` method. When a student will register the course, the information will be added into the `subjStd` dictionary of the `Department` class. Similarly an instructor will be assigned a course using `assignCourse` method and the information will be updated in the `subjInst` dictionary of the `Department` class.

The relation is shown here:



The final codes are shown below. The concept of **Enumeration Class** is used for the Subjects as well to map Subject abbreviations to full name of the subject:

```
from functools import total_ordering
from enum import Enum
class Subj(Enum):
    Mech='Mechanism'
    LA='Linear Algebra'
    ES='Embedded Systems'
    CP2='Computer Programming-II'
    MOM='Mechanics of Material'
    Proj='Design Project'
class Desig(Enum):
    Prof='Professor'
    AssocProf='Associate Professor'
    AP='Assistant Professor'
    Lect='Lecturer'
    TF='Teaching Fellow'
class Instructor:
    allInsts=[]
    def __init__(self,fName,lName,desig:Desig):
        self.fName=fName
        self.lName=lName
        self.desig=desig
        self._courses=[]
        Instructor.allInsts.append(self)
    def __repr__(self):
        return f'{self.fName} {self.lName} - {self.desig}'
    @property
    def desig(self):
        return self._desig.value
    @desig.setter
    def desig(self,newdesig):
        if(isinstance(newdesig,Desig)):
            newdesig=newdesig.name
        if(newdesig in Desig.__members__):
            self._desig=Desig[newdesig]
        else:
            raise ValueError(f'{newdesig} is not a valid
Designation!')
    def assignCourse(self,*sub):
        for s in sub:
            if isinstance(s,Subj):
                s=s.name
            if s not in Subj.__members__:
                raise ValueError(f'{s} is not offered!')
            if Subj[s].value not in self._courses:
```

```
                self._courses.append(Subj[s].value)
                if Department.subjInst.get(s) is None:
                    Department.subjInst[s]=[self]
                else:
                    Department.subjInst[s].append(self)
@total_ordering
class MechaStudent:
    'This class defines a Student for Mechatronics Department'
    _department='Mechatronics'
    _allStudents=[]
    def __init__(self,fName,lName,reg):
        self.fName=fName
        self.lName=lName
        self.reg=reg
        self.email=f'{self.reg.lower()}@uet.edu.pk'
        self._courses=[Subj.Proj.value]
        self._groupMember=None
        self.fullName=f'{self.fName} {self.lName}'
        MechaStudent._allStudents.append(self)

    ## Getters and Setters
    @property
    def fullName(self):
        return f'{self.fName} {self.lName}'
    @fullName.setter
    def fullName(self,newname):
        f,l=newname.split(' ')
        self.fName=f
        self.lName=l
        self._fullName=newname
    @property
    def fName(self):
        return self._fName
    @fName.setter
    def fName(self,newname):
        if(MechaStudent.validName(newname)):
            self._fName=newname
        else:
            raise ValueError('Name should contain alphabet only and
at least 2 of those!')
    @property
    def lName(self):
        return self._lName
    @lName.setter
    def lName(self,newname):
        if(MechaStudent.validName(newname)):
            self._lName=newname
```

```
        else:
            raise ValueError('Name should contain alphabet only and
at least 2 of those!')
    @property
    def reg(self):
        return self._reg
    @reg.setter
    def reg(self,newreg):
        if(isinstance(newreg,str) and str(newreg).startswith('MCT-
UET-')):
            self._reg=newreg
        else:
            raise ValueError('Reg must start as MCT-UET-')

    ## Static Methods ##
    @staticmethod
    def validName(name):
        if(isinstance(name,str) and len(name)>=2 and name.isalpha()):
            return True
        else:
            return False
    ## Instance Methods ##
    def regCourse(self,*sub):
        for s in sub:
            if isinstance(s,Subj):
                s=s.name
            if s not in Subj.__members__:
                raise ValueError(f'{s} is not offered!')
            if Subj[s].value not in self._courses:
                self._courses.append(Subj[s].value)
                if Department.subjStd.get(s) is None:
                    Department.subjStd[s]=[self]
                else:
                    Department.subjStd[s].append(self)
    ## Class Methods ##
    @classmethod
    def notRegSub(cls):
        a=set()
        m=[e.value for e in Subj]
        for std in cls._allStudents:
            s=set(std._courses)
            a.update(s)
        return list(set(m).difference(a))
    @classmethod
    def withoutGroupMembers(cls):
        return list(filter(lambda s: s._groupMember is
None,cls._allStudents))
```

```
    ## Magic Methods ##
    def __repr__(self):
        return f'{self.lName}-{self.reg[-2:]}'
    def __add__(self,other):
        if(self._groupMember is not None):
            raise ValueError(f'{self} already has {self._groupMember}
as group member')
        elif(other._groupMember is not None):
            raise ValueError(f'{other} already has
{other._groupMember} as group member')
        else:
            self._groupMember=other
            other._groupMember=self
    def __sub__(self,other):
        if(self._groupMember is None and other._groupMember is None):
            return
        elif(self._groupMember is not other):
            raise ValueError(f'{self} is not group member of
{other}.')
        else:
            self._groupMember=None
            other._groupMember=None
    def __lt__(self,other):
        return len(self)<len(other)
    def __eq__(self,other):
        return len(self)==len(other)
    def __len__(self):
        return len(self._courses)
    def __iter__(self):
        yield self.fName
        yield self.lName
        yield self.reg
        for c in self._courses:
            yield c
class Department:
    subjInst={}
    subjStd={'Proj':MechaStudent._allStudents}
```

See this method in **Instructor** class that assigns a course to the **Instructor** and how it updates information in **Department** class:

```
    def assignCourse(self,*sub):
        for s in sub:
            if isinstance(s,Subj):
                s=s.name
```

```
            if s not in Subj.__members__:
                raise ValueError(f'{s} is not offered!')
            if Subj[s].value not in self._courses:
                self._courses.append(Subj[s].value)
                if Department.subjInst.get(s) is None:
                    Department.subjInst[s]=[self]
                else:
                    Department.subjInst[s].append(self)
```

Similarly, observe the same in **regCourse** method in **MechaStudent** class.

```
    def regCourse(self,*sub):
        for s in sub:
            if isinstance(s,Subj):
                s=s.name
            if s not in Subj.__members__:
                raise ValueError(f'{s} is not offered!')
            if Subj[s].value not in self._courses:
                self._courses.append(Subj[s].value)
                if Department.subjStd.get(s) is None:
                    Department.subjStd[s]=[self]
                else:
                    Department.subjStd[s].append(self)
```

Let's add another method in the **Department** class to make the students group randomly. The code is shown here:

```
@staticmethod
    def createGroups():
        shuffle(MechaStudent._allStudents)
        l=len(MechaStudent._allStudents)
        for i in range(0,l,2):
            if(i!=l-1):
                MechaStudent._allStudents[i]+MechaStudent._allStudent
s[i+1]
```

A sample use of the classes in Main Program is here:

```
from Student import MechaStudent,Subj,Instructor,Desig,Department
std1=MechaStudent('Anwar','Ali','MCT-UET-01')
std2=MechaStudent('Akbar','Khan','MCT-UET-02')
std3=MechaStudent('Hamza','Akhtar','MCT-UET-03')
std4=MechaStudent('Faisal','Iqbal','MCT-UET-04')

std1.regCourse('CP2','Mech',Subj.MOM)
std2.regCourse('ES','Mech','MOM')
```

```
std3.regCourse('ES','LA','MOM','Mech','CP2')
std4.regCourse('LA','Mech','MOM')

i1=Instructor('Ali','Raza','AP')
i2=Instructor('Ahsan','Naeem',Desig.AP)
i3=Instructor('Rzi','Abbas','Lect')
i4=Instructor('Misbah','Rehman','Lect')
i5=Instructor('Shujat','Ali','TF')
i6=Instructor('Mohsin','Rizwan','AP')

i1.assignCourse('Mech')
i2.assignCourse('CP2')
i3.assignCourse(Subj.MOM)
i4.assignCourse('ES','Proj')
i5.assignCourse('LA')
i6.assignCourse('Proj')

for ins in Instructor.allInsts:
    print(f'{ins}\t{ins._courses}')

for item in Department.subjInst.items():
    print(item)

for item in Department.subjStd.items():
    print(item)
Department.createGroups()
for std in MechaStudent._allStudents:
    print(std,std._groupMember)
```

The detailed use of the classes with many **`MechaStudent`** class objects can be found in the video provided in References.

References:

[1] Multiple Classes Basics: https://youtu.be/9QIh13BZ0Eo
[2] Enumeration Class: https://youtu.be/8gsZGHgAMC8
[3] Multiple Classes (Another Example): https://youtu.be/rcbNOzVigF8
[4] Demo of Multiple Classes on bigger data: https://youtu.be/ylKeMnXus0o
[5] A Challenge Problem for OOP: https://youtu.be/TtwziQ8dEcA
[6] Challenge -II : https://youtu.be/nbtF21_uRJg
[7] Solution of Challenge-I and II: https://youtu.be/MQFOwoAF6Bw