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 37: Object Oriented Programming

Polymorphism and Magic Methods

Polymorphism (**Poly**=Many; **Morph**=Shape) means that multiple methods (or operators) have same name but multiple forms. And those different forms behave differently.

In Computer Programming there are three types of Polymorphism:

i. **Method Overloading:**
In this form, more than one methods have exactly the same name but a different signature e.g. the number of input arguments. The signature we use while calling the method will determine which one to call. Unlike many other programming languages like C, C++, JAVA etc., Python doesn't support Method Overloading.

ii. **Operator Overloading:**
In this form the operators like +,-,*,/ behave differently in different cases. For example, **`a+b`** has different behavior when **`a`** and **`b`** are integers and when they are list or strings. In case of integers the + operator will mathematically add the two numbers and in case of list or string it will concatenate the two. We can define the behavior of these operators for our class objects e.g., what to do when + operator is used between two objects of our class. We do it using the magic methods (also known as dunder methods ans special methods). We will study these in this lab session.

iii. **Method Overriding:**
In context to class inheritance (not studied yet), a subclass can have same method as of the base class but having totally different behavior. We will study these in future lab sessions.

Magic Methods:

We can write magic/special methods in our class to add basic functionalities of the class objects. Different magic methods are described here. We will be using the **`MechaStudent`** class as example to demonstrate their use.

a) String Representation→ There should be a nice string representation of the class objects so that we can directly print the class object by passing it within **`print`** statement. The magic method for this purpose are **`__str__()`** and **`__repr__()`** and we have seen their use in one of the previous lab sessions.

b) Comparison→ There should be easy ways to compare two class objects. It is generally a common requirement to see if two objects are same or not or whether one object is greater than or less the other and so on. If we try to implement these relations on two **`MechaStudent`** objects as:

```
from Student import MechaStudent
std1=MechaStudent('Anwar','Ali','MCT-UET-01')
std2=MechaStudent('Akbar','Khan','MCT-UET-02')
std3=MechaStudent('Asad','Shabir','MCT-UET-03')
std4=MechaStudent('Faisal','Iqbal','MCT-UET-04')

std1.registerSubject('CP-II','Mech','MOM')
std2.registerSubject('ES','Mech','MOM')
```

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

print(std1>std2)
```

You will see following error:
TypeError: '>' not supported between instances of 'Student' and 'Student'

The error is self-explanatory. To set such comparisons between two class instances, firstly, we must decide the rule of comparison. For example, in **`MechaStudent`** class, the objects can be compared based on the registered courses. So, lets first define following rules:

i. Two students will be equal if both have equal number of registered subjects.
ii. A student, with more registered subjects than the other, will be considered as greater than the other.

If we consider only the first point of equality, we can implement it in Main program as:

```
print(len(std1._courses)==len(std2._courses))
```

And this will print **`True`**. But we can implement this definition of equality within the class and use that in main program in a much simpler way as:

```
print(std1==std2)
```

The magic method to define the rule of equality within the class is **`__eq__(self,other)`**. This method must have just these two input arguments i.e. **`self`** and **`other`**. For above case it should be defined in the class as:

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

And now in main program we can compare two objects directly as shown here:

```
from Student import MechaStudent
std1=MechaStudent('Anwar','Ali','MCT-UET-01')
std2=MechaStudent('Akbar','Khan','MCT-UET-02')
std3=MechaStudent('Asad','Shabir','MCT-UET-03')
std4=MechaStudent('Faisal','Iqbal','MCT-UET-04')

std1.registerSubject('CP-II','Mech','MOM')
std2.registerSubject('ES','Mech','MOM')
std3.registerSubject('ES','LA','MOM','Mech')
std4.registerSubject('LA','Mech','MOM')

print(std1==std2)   # Will print True
print(std1==std3)   # Will print False
```

Likewise we can define the rule of one object being greater than the other within the class using the magic method **`__gt__(self,other)`** as shown here:

```
def __gt__(self,other):
        return len(self._courses)>len(other._courses)
```

Once this method is defined in class, we can compare the class objects using `<` or `>` operator as shown here for the same students and their registered subjects as in last Main program:

```
print(std1==std2)    # Will print True
print(std1>std3)     # Will print False
print(std1<std3)     # Will print False
```

There is also another magic method `__lt__(self,other)` to define the less than rule and then can use `<` and `>`. We should define only one of `__gt__(self,other)` and `__lt__(self,other)` as the other relation will be derived automatically. However, if for some rare reason you want to define `>` and `<` operators which are not the compliment of each other, then you can define both.

Another good thing is that now we can apply `sort()` method on a list of class objects as rule of comparison has been defined. So, we can have now:

```
from Student import MechaStudent
std1=MechaStudent('Anwar','Ali','MCT-UET-01')
std2=MechaStudent('Akbar','Khan','MCT-UET-02')
std3=MechaStudent('Asad','Shabir','MCT-UET-03')
std4=MechaStudent('Faisal','Iqbal','MCT-UET-04')

std1.registerSubject('CP-II','Mech','MOM')
std2.registerSubject('ES','Mech','MOM')
std3.registerSubject('ES','LA','MOM','Mech')
std4.registerSubject('LA','Mech','MOM')

MechaStudent._allStudents.sort()
print(MechaStudent._allStudents)
```

Moreover, if we want to sort the list based on some other rule e.g. the names of the students, we can still do it by specifying the `key` within `sort()` method.

The two comparison methods; one for equality and other for greater-than have been define but if we try to compare the objects as:

```
print(std1>=std3)
```

It will generate following error:

```
TypeError: '>=' not supported between instances of 'MechaStudent' and 'MechaStudent'
```

We have more magic methods for the comparison; `__ge__()` for greater than or equal to and `__le__()` for less than or equal to.

You might be very rightly thinking that it is a lot of code we need to write for comparison and why not the definition of `>=` is taken when it has definitions for both equality `==` and greater than `>` ?

Python does provide a class decorator named as `total_ordering` available `functools` module that can derive other comparison relations from the defined relations. For that, the documentation of `functools` says:

> *Given a class defining one or more rich comparison ordering methods, this class decorator supplies the rest. This simplifies the effort involved in specifying all of the possible rich comparison operations. The class must define one of __lt__(), __le__(), __gt__(), or __ge__(). In addition, the class should supply an __eq__() method.*

Means that we must provide `__eq__()` definition and any one out of other four and the rest will be derived by the decorator by itself.

We already have the two magic methods `__eq__()` and `__gt__()` defined in the class `MechaStudent`. So, if we decorate it as:

```
from functools import total_ordering
@total_ordering
class MechaStudent:
```

Now we can directly use `>=` or `<=` as shown here:

```
print(std1>=std3) #Will print False instead of error
```

Be Careful → With `__eq__()` defined; `setGroupMember()` will generate different error when a student already has a `GroupMember`. The reason is that when we have `__eq__()` defined, this will be called whenever we use `==` on two students. Note that `self.groupMember` is a `MechaStudent` class object. When we have `self.groupMember==None` , `__eq__()` is called on `self` with `None` assigned to `other`. Inside `__eq__()` method, `other.courses` when executed, will generate error this time. The solution is to use `is` operator rather than `==` and that too results into `True` or `False` but will not call the `__eq__()` method as `__eq__()` is linked to `==` only and not to `is` operator.

c) Length of Object → We can find the length of different iterables using the built-in `len()` function. We can also set the `len()` function for our class and later will be able to use `len()` on the class objects. The magic method to do this is `__len__()`. Considering the Student class once again, as we have developed all comparisons based on number of registered subjects, it will be appropriate to set the length of a student as number of subjects registered by him/her. This can be done by defining the `__len__()` in the class as:

```
def __len__(self):
     return len(self._courses)
```

Now we can use it on any object of the class as:

```
print(len(std1)) # Will print 3
```

The good thing is that previously we were using `len()` on courses attribute in different comparison magic method for example the `__eq__()` was defined as:

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

But once the `len()` has been defined via `__len__()` and can be applied on the instance giving the number of courses registered, we can have `__eq__()` as:

```
def __eq__(self,other):
        return len(self)==len(other)
```

Finally, just to see that basically the magic function `__len__()` is defined inside the class for all the iterables on which we can use `len()`, see the following code that prints the length of a tuple in two ways:

```
a=(5,-3,4,-2,1)
print(len(a))
print(a.__len__())
```

But of course, we always use the first method i.e. `len()` and that in return calls the magic method `__len__()` defined inside the class.

d) Absolute value of the Object→We can have `__abs__()` special method to use `abs()` on the object. You can explore that by yourself.

e) Making the Object Iterable→Another important special method is `__iter__()` that makes the class object iterable just like other iterables like list, set etc. Usually we return a generator expression from this method. For `MechaStudent` class we can make it iterable with returning the courses registered by the student for understanding. The `__iter__()` method will be defined as:

```
def __iter__(self):
        return (x for x in self._courses)
```

Its use in Main program is shown here:

```
from Mechatronics import Student
std1=Student('Anwar','Ali','MCT-UET-01')
std1.registerSubj('CP-II','Mech','MOM')

for i in std1:
    print(i)
```

The output will be:

```
Proj
CP-II
Mech
MOM
```

For cases where the values we want to return from the iterator are present inside an object which already is iterable e.g.in above case the courses are inside a list which is iterable, then we can use the `iter()` method of that iterable as shown here:

```
def __iter__(self):
        return iter(self._courses)
```

For the other cases where we want to return values from various sources, we can code `__iter__` as generator function with `yield` statement. For example, if we want to return `fName`, `lName`, `reg`, `courses` and `groupMemebr` of a student when it is iterated, we can code `__iter__()` as:

```
def __iter__(self):
        yield self.fName
        yield self.lName
        yield self.reg
        yield self._courses
        yield self._groupMember
```

f) Mathematical and Logical Operations→ We can also define different mathematical and logical operators on the class objects e.g. +, -, &, or etc. If we try to add two students like std1+std2, it will generate error as:

```
TypeError: unsupported operand type(s) for +: 'MechaStudent' and 'MechaStudent'
```

The error is self-explanatory that the + is not supported between `MechaStudent` and `MechaStudent`. To define + operator between two **`MechaStudent`** objects; first we need to decide what we want to get when two `MechaStudent` objects are added. Although it's not very logical but let's decide that when two **`MechaStudent`** objects are added, we wish to get a set of all subjects registered by those two.

To define + operator, the special method is **`__add__()`** and we can define it in **`MechaStudent`** class for the decided rule as:

```
def __add__(self,other):
        return set(self._courses).union(set(other._courses))
```

And now we can use + between two objects as:

```
from Mechatronics import Student
std1=Student('Anwar','Ali','MCT-UET-01')
std2=Student('Akbar','Khan','MCT-UET-02')
std3=Student('Asad','Shabir','MCT-UET-03')
std4=Student('Faisal','Iqbal','MCT-UET-04')

std1.registerSubj('CP-II','Mech','MOM')
std2.registerSubj('ES','Mech','MOM')
std3.registerSubj('ES','LA','MOM','Mech')
std4.registerSubj('LA','Mech','MOM')

print(std1+std2)  # Will print {'Mech', 'MOM', 'ES', 'Proj', 'CP-II'}
```

Likewise, there are magic methods for other mathematical and logical operations. The list is given in the table below:

Binary Operators			
Operator	Method	MechaStudent Class Example Use	Point Class Example Use
+	__add__(self, other)	std1+std2	p1+p2
-	__sub__(self, other)	std1-std2	p1-p2
*	__mul__(self, other)	std1*std2	p1*p2
//	__floordiv__(self, other)	std1//std2	p1//p2
/	__truediv__(self, other)	std1/std2	p1/p2
%	__mod__(self, other)	std1%std2	p1%p2
**	__pow__(self, other[, modulo])	std1**std2 pow(std1,std2)	p1**p2 pow(p1,p2)
<<	__lshift__(self, other)	std1<<std2	p1<<p2
>>	__rshift__(self, other)	std1>>std2	p1>>p2
&	__and__(self, other)	std1&std2	p1&p2
^	__xor__(self, other)	std1^std2	p1^p2
|	__or__(self, other)	std1|std2	p1|p2
Extended Assignments or In-placed Operators			
Operator	Method	MechaStudent Class Example Use	Point Class Example Use
+=	__iadd__(self, other)	std1+=std2	p1+=p2
-=	__isub__(self, other)	std1-=std2	p1-=p2
*=	__imul__(self, other)	std1*=std2	p1*=p2
/=	__idiv__(self, other)	std1/=std2	p1/=p2
//=	__ifloordiv__(self, other)	std1//=std2	p1//=p2
%=	__imod__(self, other)	std1%=std2	p1%=p2
**=	__ipow__(self, other[, modulo])	std1**=std2	p1**=p2
<<=	__ilshift__(self, other)	std1<<=std2	p1<<=p2
>>=	__irshift__(self, other)	std1>>=std2	p1>>=p2
&=	__iand__(self, other)	std1&=std2	p1&=p2
^=	__ixor__(self, other)	std1^=std2	p1^=p2
|=	__ior__(self, other)	std1|=std2	p1|=p2

Comparison Operators			
Operator	Method	MechaStudent Class Example Use	Point Class Example Use
<	__lt__(self, other)	std1<std2	p1<p2
<=	__le__(self, other)	std1<=std2	p1<=p2
==	__eq__(self, other)	std1==std2	p1==p2
!=	__ne__(self, other)	std1!=std2	p1!=p2
>=	__ge__(self, other)	std1>=std2	p1>=p2
>	__gt__(self, other)	std1>std2	p1>p2
Unary Operators			
Operator	Method	MechaStudent Class Example Use	Point Class Example Use
-	__neg__	-std1	-p1
+	__pos__	+std1	+p1
abs	__abs__	abs(std1)	abs(p1)
length	__len__(self)	len(std1)	len(p1)
Iterable	__iter__(self)	for i in std1:	for i in p1:

The **+** operator definition we implemented for **Mechatudent** class is good for understanding and one important point to remember is that unlike comparison methods where usually we should always have Boolean type in **return**, in case of **__add__()** we can have anything returned. We have two methods **setGroupMember()** and **dropGroupMember()** to set or remove **groupMember** of a student. Compared to the previous definition of **__add__()** method, it will be better if we replace **setGroupMember** with **__add__()** and replace **dropGroupMember()** with **__sub__()** so that if we wish to make two students the **groupMember** of each other we could simple do it as **std1+std2**. The complete code of **Mechatudent** class is given here with these two changes is given below:

```python
from functools import total_ordering
@total_ordering
class MechaStudent:
    'This class defines a Student for Mechatronics Department'
    _department='Mechatronics'
    _offSubjects=['Mech','LA','ES','CP-II','MOM','Proj']
    _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=['Proj']
        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 registerSubject(self,*sub):
        for s in sub:
            if s not in MechaStudent._offSubjects:
                raise ValueError(f'{s} is not offered!')
            if s in MechaStudent._offSubjects and s not in
self._courses:
                self._courses.append(s)
    ## Class Methods ##
    @classmethod
    def notRegSub(cls):
        a=set()
        for std in cls._allStudents:
            s=set(std._courses)
            a.update(s)
        return list(set(cls._offSubjects).difference(a))
    @classmethod
    def withoutGroupMembers(cls):
        return list(filter(lambda s:
s._groupMember==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!=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
```

One program using the above class is given here:

```
from Student import MechaStudent
std1=MechaStudent('Anwar','Ali','MCT-UET-01')
std2=MechaStudent('Akbar','Khan','MCT-UET-02')
std3=MechaStudent('Asad','Shabir','MCT-UET-03')
std4=MechaStudent('Faisal','Iqbal','MCT-UET-04')

std1.registerSubject('CP-II','Mech','MOM')
std2.registerSubject('ES','Mech','MOM')
std3.registerSubject('ES','LA','MOM','Mech')
std4.registerSubject('LA','Mech','MOM')

std1+std2                # Will print Akbar Khan-MCT-UET-02
std3+std4
print(std1._groupMember)
# std1+std3              # Will generate error
# std1-std3              # Will generate error
MechaStudent._allStudents.sort()
print(MechaStudent._allStudents)
# Will Print [Anwar Ali-MCT-UET-01, Akbar Khan-MCT-UET-02,
# Faisal Iqbal-MCT-UET-04, Asad Shabir-MCT-UET-03]
```

Tasks:

[1] In main Add following features in the class **Point**.

a) Instead of **mag** for the magnitude of the point, set magnitude as absolute value of point using **__abs__()** method.
b) Define **negative operator** on a point such that negative of a point will return a new point with co-ordinates as negative of the original point.
c) Define **+** and – operators between two point objects. There must be support for in-place operation for both operators.
d) Define division operator **/** between a point and a number.
e) Define all **rich comparisons** between two points based on the magnitude of the point.
f) Make Point object **iterable** that generates x and y components.

[2] In Main Program:

a) Create a list of **10** random **points**. The **x-y co-ordinates** of the points must be integer and within range **-5 to 5.**
b) From above list filter out and generate another list of points whose magnitude is between 2 and 3 (both inclusive).

References:

[1] https://youtu.be/LQp5Bf8yKzE
[2] https://youtu.be/bb7MLq2F5ic

NotImplemented and Reflection Function:

Watch the video described below:
https://youtu.be/vqzrLnZ9Y5M

Tasks:

[3] In Task-1 give above, include this:

a) Define ***** operator that could multiply a Point with a number in any order and should also be able to multiply two point objects resulting into **dot product** of two points.
b) Multiply each point of above list with **5** and take negative of those points.
c) Generate another list whose values will be the dot products of each point of above list with Point **(5,10)**