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

Prepared By:
Mr. Muhammad Ahsan Naeem

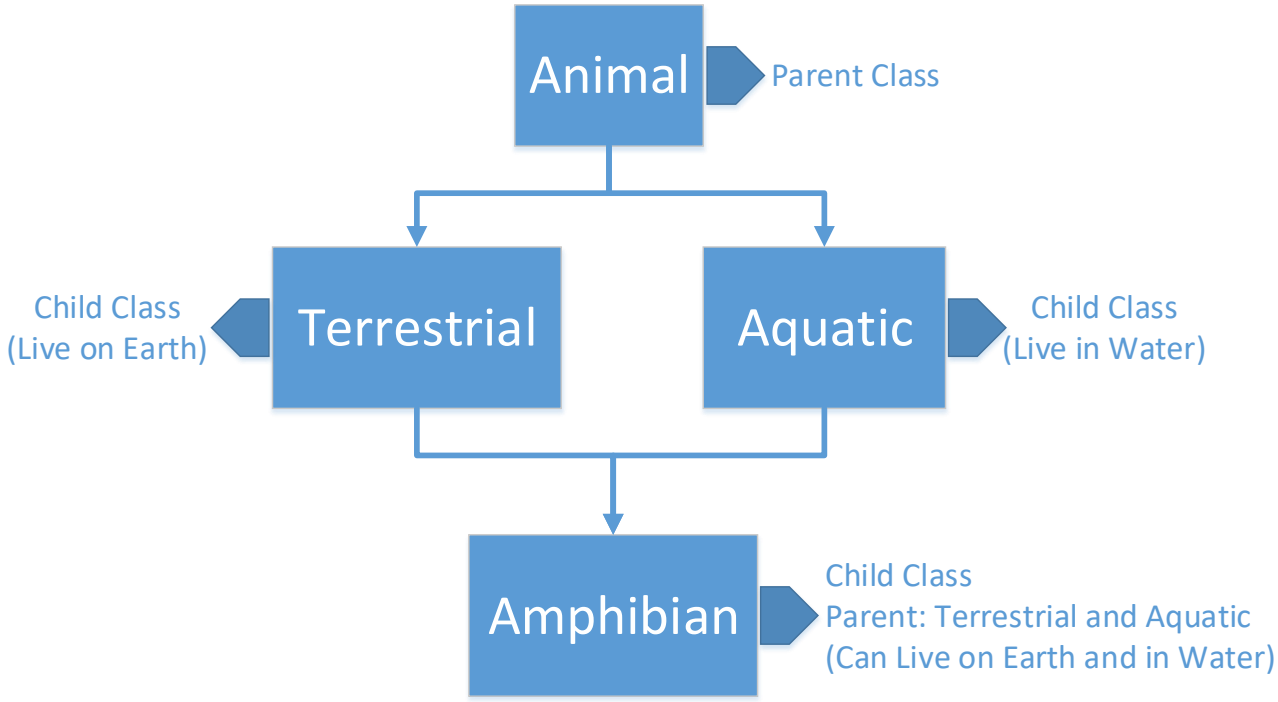
YouTube Playlist
https://youtube.com/playlist?list=PLWF9TXck7O_zuU2_BVUTrmGMCXYSYzjku

Lab 40: Object Oriented Programming

Multiple Class Inheritance

Multiple Inheritance:

Python also allows multiple inheritance which means that a class can be inherited from more than one parent class. To understand the concept, we will consider the class **`Animal`** having two subclasses; **`Terrestrial`** (the animals which live on earth) and **`Aquatic`** (the animals which live in water). Then there is another type of animals which can live on earth as well as in water e.g. turtle and are known as the **`Amphibian`**. Hence, we can have a subclass named as **`Amphibian`** that is inherited from the **`Terrestrial`** and **`Aquatic`** both. See the graphical representation as under:



Now see the code below and observe that the object of **`Amphibian`** class inherits all methods from the three classes:

```
class Animal:
    def breath(self):
        print("Yes I'm breathing!")
class Terrestrial(Animal):
    def walkOnEarth(self):
        print("I am walking now...")
class Aquatic(Animal):
    def stayInWater(self):
        print("I live in water...")
class Amphibian(Terrestrial,Aquatic):
    pass
```

```
## Main Program ##
turtle=Amphibian()
turtle.breath()
turtle.walkOnEarth()
turtle.stayInWater()
```

The output is:

```
Yes I'm breathing!
I am walking now...
I live in water...
```

Method Over-riding:

What if the two (or more) parent classes have the same method and we try to run that on the child class? See this code where the two parent classes have same method names:

```
class Animal:
    def breath(self):
        print("Yes I'm breathing!")
class Terrestrial(Animal):
    def walkOnEarth(self):
        print("I am walking now...")
    def stayInWater(self):
        print("I cannot stay in water...")
class Aquatic(Animal):
    def walkOnEarth(self):
        print("I can't live on earth...")
    def stayInWater(self):
        print("I live in water...")
class Amphibian(Terrestrial,Aquatic):
    pass

## Main Program ##
turtle=Amphibian()
turtle.breath()
turtle.walkOnEarth()
turtle.stayInWater()
```

Here is the output:

```
Yes I'm breathing!
I am walking now...
I cannot stay in water...
```

You can see that the two methods `walkOnEarth()` and `stayInWater()` have been inherited from class `Terrestrial`. But why not from class `Aquatic`?

To get the answer, let's see the **Method Resolution Order** for `Amphibian` class as:

```
print(help(Amphibian))
```

And you will see:

```
 |  Method resolution order:
 |      Amphibian
 |      Terrestrial
 |      Aquatic
 |      Animal
 |      builtins.object
```

You can clearly see from the order that the `Terrestrial` class comes before the `Aquatic` class. The two methods were not found in `Amphibian` class but were found in `Terrestrial` class and hence got executed. It should not be difficult to understand why `Terrestrial` class is before the `Aquatic` class. Because we specified it before the `Aquatic` while defining the subclass as:

```
class Amphibian(Terrestrial,Aquatic):
```

If we change the order here as:

```
class Amphibian(Aquatic,Terrestrial):
```

Now, `Aquatic` class will come before `Terrestrial` in MRO and the methods defined in `Aquatic` class will get executed if they lie in both classes and not present in `Amphibian` class.

But this is not something we want here i.e. changing the order and getting the methods from first parent class. In above case, for `Amphibian` class we want `walkOnEarth()` method be inherited from `Terrestrial` class and `stayInWater()` from the `Aquatic`.

One way can be to define these methods with repeated logic inside the subclass, but this is never a good idea as the logic can be lengthy and may need future updates. Alternatively, what we can define these methods inside the subclass but rather than repeating the same logic we will inherit it by specifying the parent class we wish to get that.

The complete code with this approach is shown below. Focus on the methods inside `Amphibian` class.

```
class Animal:
    def breath(self):
        print("Yes I'm breathing!")
class Terrestrial(Animal):
```

```
    def walkOnEarth(self):
        print("I am walking now...")
    def stayInWater(self):
        print("I cannot stay in water...")
class Aquatic(Animal):
    def walkOnEarth(self):
        print("I can't live on earth...")
    def stayInWater(self):
        print("I live in water...")
class Amphibian(Terrestrial,Aquatic):
    def walkOnEarth(self):
        Terrestrial.walkOnEarth(self)
    def stayInWater(self):
        Aquatic.stayInWater(self)

## Main Program ##
turtle=Amphibian()
turtle.breath()
turtle.walkOnEarth()
turtle.stayInWater()
```

The out is shown here:

```
Yes I'm breathing!
I am walking now...
I live in water...
```

It is important to note that we cannot use **`super()`** to refer to specific parent in case of multiple inheritance. If we do that, **`super()`** will refer to the first parent as was the case earlier. You can check this with the following class code:

```
class Amphibian(Terrestrial,Aquatic):
    def walkOnEarth(self):
        super().walkOnEarth()
    def stayInWater(self):
        super().stayInWater()
```

A similar approach is needed if we want to have different yet somewhat similar **`__init__`** methods in parent and child classes. Suppose we define **`__init__`** methods in **`Animal`**, **`Terrestrial`** and **`Aquatic`** class as:

```
class Animal:
    def __init__(self,weight,age):
        self.weight=weight
```

```
        self.age=age
class Terrestrial(Animal):
    def __init__(self,weight,age,legs):
        Animal.__init__(self,weight,age)
        self.legs=legs
class Aquatic(Animal):
    def __init__(self,weight,age,maxDepth):
        Animal.__init__(self,weight,age)
        self.maxDepth=maxDepth
```

The other methods have been removed for the purpose of clarity. You can see that the **Animal** class needs weight and age of the animal. **Terrestrial** class uses **__init__()** of **Animal** class to specify the same two parameters and has another one as legs. Likewise, the **Aquatic** class uses **__init__()** of **Animal** to set the two parameters and has another parameter as the maximum depth of water in feet till where the aquatic animal can stay named as **maxDepth**.

Now for the **__init__()** of **Amphibian** class, first decide about the parameters and refer to the parent class if needed. If we want to set both **legs** and **maxDepth** for **Amphibian** class, other than the **weight** and the **age**, we can refer to **__init__()** of one of the parent and set the leftover manually. The third way can be to refer to **__init__()** of both **Terrestrial** and **Aquatic** class. All three ways are shown here:

```
class Animal:
    def __init__(self,weight,age):
        self.weight=weight
        self.age=age
class Terrestrial(Animal):
    def __init__(self,weight,age,legs):
        Animal.__init__(self,weight,age)
        self.legs=legs
class Aquatic(Animal):
    def __init__(self,weight,age,maxDepth):
        Animal.__init__(self,weight,age)
        self.maxDepth=maxDepth
class Amphibian(Terrestrial,Aquatic):
    def __init__(self,weight,age,legs,maxDepth):
        #First method
        Terrestrial.__init__(self,weight,age,legs)
        self.maxDepth=maxDepth
        #Second method
        # Aquatic.__init__(self,weight,age,maxDepth)
        # self.legs=legs
        #Third Method
        # Terrestrial.__init__(self,weight,age,legs)
```

```
        # Aquatic.__init__(self,weight,age,maxDepth)

## Main Program ##

turtle=Amphibian(weight=7,age=3,legs=4,maxDepth=1000)
print(turtle.maxDepth)
```

Department Staff Example:

Continuing with the example of **Employees** of a department; **Instructor**, **Lab Director**, **AdminStaff**, **AdminOfficer**, let's now consider to add **Head of the Department (HoD)**. HoD is always a faculty member and also has the Admin powers. So, HoD will be child of two classes; i.e. **LabDirector** and **AdminOfficer**. He can reassign the Lab Director and all Admin Staff are his team members. Based on this idea here is the code:

```
class Employee:
    raise_amount=1.1 #10% annual raise in income
    total_emp=0
    def __init__(self,fName,lName,pay):
        self.fName=fName
        self.lName=lName
        self.pay=pay
        self.email=f"{self.fName.lower()}.{self.lName.lower()}@uet.ed
u.pk"
        Employee.total_emp+=1
    def raise_pay(self):
        self.pay*=self.raise_amount
    def __repr__(self):
        return f'{self.fName} {self.lName}'
class Instructor(Employee):
    def __init__(self,fName,lName,pay,desig):
        Employee.__init__(self,fName,lName,pay)
        # super().__init__(fName,lName,pay)
        self.desig=desig
        self.courses=[]
    def assignCourse(self,*subj):
        self.courses=list(set(self.courses+list(subj)))
class LabDirector(Instructor):
    def __init__(self,fName,lName,pay,desig,lab,*labEmp):
        Instructor.__init__(self,fName,lName,pay,desig)
        self.lab=lab
        self.labEmp=list(labEmp)
    def assignCourse(self,*subj):
        super().assignCourse(*subj)
        for ins in self.labEmp:
```

```
            ins.assignCourse(*subj)
    def addLabEmployee(self,ins):
        self.labEmp.append(ins)
    def dropLabEmployee(self,ins):
        self.labEmp.remove(ins)

class AdminStaff(Employee):
    raise_amount=1.15
    allAdminStaff=[]
    def __init__(self,fName,lName,pay,team=None):
        Employee.__init__(self,fName,lName,pay)
        self.team=team
        self.tasks=[]
        AdminStaff.allAdminStaff.append(self)
    def assignTask(self,*task):
        self.tasks=list(set(self.tasks+list(task)))
class AdminOfficer(AdminStaff):
    def __init__(self,fName,lName,pay,team):
        AdminStaff.__init__(self,fName,lName,pay,team)
    @property
    def teamMembers(self):
        return list(filter(lambda s:
s.team==self.team,AdminStaff.allAdminStaff))
class HOD(LabDirector,AdminOfficer):
    def __init__(self,fName,lName,pay,desig,lab,labEmp=None):
        LabDirector.__init__(self,fName,lName,pay,desig,lab,labEmp)
        AdminOfficer.__init__(self,fName,lName,pay,team='Admin')
    @property
    def teamMembers(self):
        return AdminStaff.allAdminStaff
    def changeLab(self,LD,newLab):
        LD.lab=newLab
```

Example use is shown here:

```
head=HOD('Ali','Raza',120000,'AP','Robotics')

i1=Instructor('Ahsan','Naeem',100000,'AP')
i2=Instructor('Rzi','Abbas',95000,'Lect')
i3=Instructor('Misbah','Rehman',90000,'Lect')
i4=Instructor('Shujat','Ali',85000,'TF')

labDir1=LabDirector('Mohsin','Rizwan',120000,'AP','Embedded
Systems',i1,i2)
print(labDir1.labEmp)
labDir1.assignCourse('Robotics')
print(labDir1.courses)
```

```
print(i1.courses)
print(i2.courses)

as1=AdminStaff('Imran','Hanif',90000,'Exam')
as2=AdminStaff('Saqib','Majeed',105000,'Dues')
as3=AdminStaff('Inam','Haider',110000,'Exam')

off1=AdminOfficer('Naveed','Aslam',120000,'Exam')
print(off1.teamMembers)

print(head.teamMembers)
print(labDir1.lab)
head.changeLab(labDir1,'Control Systems')
print(labDir1.lab)
```

How to decide whether to have a new class as subclass or a separate class?

We have studied:

- multi classes without inheritance
- subclass or class inheritance
- multiple inheritance

The beginner programmers often find it hard to decide whether to use a new class or a child class. Let's take the example of a **Deck** of playing cards and suppose we want to create two classes named as `Card` and `Deck`. So, the `Card` class should be the subclass of `Deck` class or a separate class? To decide that ask a simple question to yourself; **Is Card a Deck?** Of course not! Hence `Card` class will not be the subclass of `Deck` class. Consider the case of `Instructor` class that was child of the `Employee` class and ask the same question i.e. **Is Instructor an Employee?** The answer is **Yes** and hence the `Instructor` is rightly the child of `Employee` class.

Just to conclude the `Deck` and `Card` example, they will be different classes and `Deck` class `__init__()` method should create 52 objects of `Card` class. One possible way of coding the two classes with basic functionality like shuffling the deck is shown here:

```
import random
class Card:
    def __init__(self,suit,value):
        self.suit=suit
        self.value=value
    def __repr__(self):
        return f'{self.value}-{self.suit}'
    def __str__(self):
        return f'{self.value}-{self.suit}'
class Deck:
    def __init__(self):
```

```
        self.cards=[]
        for s in ['Spade','Club','Diamond','Heart']:
            for v in ['A','2','3','4','5','6','7',
            '8','9','10','J','Q','K']:
                self.cards.append(Card(s,v))
    def shuffle(self):
        random.shuffle(self.cards)
    def __repr__(self):
        return self.cards.__str__()

## Main Program ##
myDeck=Deck()
print(myDeck)
myDeck.shuffle()
print(myDeck)
```

In above scenario, we call that **`Card`** class is composed inside the **`Deck`** card and the phenomenon is known as **Class Composition**. Class Composition relates the two classes with **"has-a"** relation. **`Deck`** has **`Card`**.