Object-Oriented Programming (OOPS-2)

Inheritance

- Inheritance is a powerful feature in Object-Oriented Programming.
- Inheritance can be defined as the process where one class acquires the properties (methods and fields) of another. With the use of inheritance, the information is made manageable in a hierarchical order.
- The class which inherits the properties of the other is known as **subclass** *(derived class or child class)* and the class whose properties are inherited is known as **superclass** *(base class, parent class)*.

Let us take a real-life example to understand inheritance. Let's assume that **Human** is a class that has properties such as **height**, **weight**, **age**, etc and functionalities (or methods) such as **eating()**, **sleeping()**, **dreaming()**, **working()**, etc.

Now we want to create **Male** and **Female** classes. Both males and females are humans and they share some common properties (like **height**, **weight**, **age**, etc) and behaviors (or functionalities like **eating()**, **sleeping()**, etc), so they can inherit these properties and functionalities from the **Human** class. Both males and females also have some characteristics specific to them (like men have short hair and females have long hair). Such properties can be added to the **Male** and **Female** classes separately.

This approach makes us write less code as both the classes inherited several properties and functions from the superclass, thus we didn't have to re-write them. Also, this makes it easier to read the code.

Python Inheritance Syntax

```
class SuperClass:
    #Body of base class
class SubClass(BaseClass):
    #Body of derived class
```

The name of the superclass is passed as a parameter in the subclass while declaration.

Example of Inheritance in Python

To demonstrate the use of inheritance, let us take an example.

A polygon is a closed figure with 3 or more sides. Say, we have a class called Polygon defined as follows.

```
class Polygon:
    def __init__(self, no_of_sides): #Constructor
        self.n = no_of_sides
        self.sides = [0 for i in range(no_of_sides)]

    def inputSides(self): #Take user input for side lengths
       self.sides=[int(input("Enter side: "))for i in range(self.n)]

    def displaySides(self): #Print the sides of the polygon
        for i in range(self.n):
            print("Side",i+1,"is",self.sides[i])
```

This class has **data attributes** to store the number of sides **n** and magnitude of each side as a list called **sides**.

The `inputSides()` method takes in the magnitude of each side and `dispSides()` displays these side lengths.

Now, a triangle is a polygon with 3 sides. So, we can create a class called **Triangle** which inherits from **Polygon**. In other words, we can say that every triangle is a polygon. This makes all the attributes of the **Polygon** class available to the **Triangle** class.

Constructor in Subclass

The constructor of the subclass must call the constructor of the superclass by accessing the **__init__** method of the superclass as:

```
<SuperClassName>.__init__(self,<Parameter1>,<Parameter2>,...)
```

Note: The parameters being passed in this call must be the same as the parameters being passed in the superclass' **__init__** function, otherwise it will throw an error.

The **Triangle** class can be defined as follows.

```
class Triangle(Polygon):
    def __init__(self):
        Polygon.__init__(self,3) #Calling constructor of superclass

    def findArea(self):
        a, b, c = self.sides
        # calculate the semi-perimeter
        s = (a + b + c) / 2
        area = (s*(s-a)*(s-b)*(s-c)) ** 0.5
        print('The area of the triangle is %0.2f' %area)
```

However, the class **Triangle** has a new method `findArea()` to find and print the area of the triangle. This method is only specific to the **Triangle** class and not **Polygon** class. Here is a sample run:

```
>>> t = Triangle() #Instantiating a Triangle object
>>> t.inputSides()
Enter side 1 : 3
Enter side 2 : 5
Enter side 3 : 4
>>> t.dispSides()
Side 1 is 3
Side 2 is 5
Side 3 is 4
>>> t.findArea()
The area of the triangle is 6.00
```

We can see that even though we did not define methods like `inputSides()` or `displaySides()` for class Triangle separately, we were able to use them. If an attribute is not found in the subclass itself, the search continues to the superclass.

Access Modifiers

Various object-oriented languages like C++, Java, Python control access modifications which are used to restrict access to the variables and methods of the class. Most programming languages have three forms of access modifiers, which are **Public**, **Private,** and **Protected** in a class.

Public Modifier

The members of a class that are declared **public** are easily accessible from any part of the program. All data members and member functions of a class are **public** by default.

Consider the given example:

```
class Student:
    name = None # public member by default
    public age = None # public member

     # constructor
    def __init__(self, name, age):
        self.name = name
        self.age = age

obj = Student("Boy", 15)
print(obj.age)  #calling a public member of the class
print(obj.name) #calling a private member of the class
```

We will get the output as:

```
10
Boy
```

We will be able to access both **name** and **age** of the object from outside the class as they are **public**. However, this is not a good practice due to *security concerns.*

Private Modifier

The members of a class that are declared **private** are accessible within the class only. A private access modifier is the most secure access modifier. Data members of a class are declared private by adding a double underscore **'__'** symbol before the data member of that class. Consider the given example:

```
class Student:
    __name = None # private member
    age = None # public member

    def __init__(self, name, age):   # constructor
        self.__name = name
        self.age = age
obj = Student("Boy", 15)
print(obj.age)  #calling a public member of the class
print(obj.name) #calling a private member of the class
```

We will get the output as:

```
10
AttributeError: 'Student' object has no attribute 'name'
```

We will get an **AttributeError** when we try to access the **name** attribute. This is because **name** is a **private** attribute and hence it cannot be accessed from outside the class.

Note: *We can even have* ***public*** *and* ***private*** *methods.*

Private and Public modifiers with Inheritance

- The subclass will be able to access any **public** method or instance attribute of the superclass.
- The subclass will not be able to access any **private** method or instance attribute of the superclass.

Protected Modifier

The members of a class that are declared protected are only accessible to a class derived from it. Data members of a class are declared **protected** by adding a single underscore **'_'** symbol before the data member of that class.

The given example will help you get a better understanding:

```
# superclass
class Student:
     _name = None # protected data member
     # constructor
     def __init__(self, name):
          self._name = name
```

This is the parent class **Student** with a **protected** instance attribute **_name**. Now consider a subclass of this class:

```
class Display(Student):
        # constructor
        def __init__(self, name):
                Student.__init__(self, name)
        def displayDetails(self):
                # accessing protected data members of the superclass
                print("Name: ", self._name)

obj = Display("Boy") # creating objects of the derived class
obj.displayDetails() # calling public member functions of the class
obj.name  # trying to access protected attribute
```

This class **Display** inherits the **Student** class. The method `displayDetails()` accesses the **protected** attribute **_name**. Further, we try to access it again outside this class.

Output:

```
Name: Boy
AttributeError: 'Display' object has no attribute 'name'
```

You can observe that we were able to access the **protected** attribute **_name** from inside the `displayDetails()` method in the subclass. However, we were not able to access it outside the subclass and we got an **AttributeError**. This justifies the definition of the **protected** modifier.

Polymorphism

The literal meaning of polymorphism is the condition of occurrence in different forms. Polymorphism is a very important concept in programming. It refers to the use of a single type entity (method, operator, or object) to represent different types in different scenarios. Let's take a few examples:

Example 1: Polymorphism in addition(+) operator

We know that the **+** operator is used extensively in Python programs. But, it does not have a single usage. For integer data types, the **+** operator is used to perform arithmetic addition operation.

```
num1 = 1
num2 = 2
print(num1+num2)
```

Hence, the above program outputs **3**.

Similarly, for string data types, the **+** operator is used to perform concatenation.

```
str1 = "Python"
str2 = "Programming"
print(str1+" "+str2)
```

As a result, the above program outputs `"Python Programming"`.

Here, we can see that a single operator + has been used to carry out different operations for distinct data types. This is one of the most simple occurrences of **polymorphism** in Python.

Example 2: Functional Polymorphism in Python

There are some functions in Python which are compatible to run with multiple data types.

One such function is the `len()` function. It can run with many data types in Python. Let's look at some example use cases of the function.

```
print(len("abcdefgeh"))
print(len(["a", "b", "c"]))
print(len({"a": 1, "b": 2}))
```

Output

```
9
3
2
```

Here, we can see that many data types such as string, list, tuple, set, and dictionary can work with the `len()` function. However, we can see that it returns specific information(the length) about the specific data types.

Class Polymorphism in Python

Polymorphism is a very important concept in Object-Oriented Programming. We can use the concept of polymorphism while creating class methods as Python allows different classes to have methods with the same name.

We can then later generalize calling these methods by disregarding the object we are working with.

Let's look at an example:

Example 3: Polymorphism in Class Methods

```
class Male:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def info(self):
        print("Hi, I am Male")

class Female:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def info(self):
        print("Hi, I am Female")


M = Male("Sid", 20)
F = Female("Zee",21)

for human in (M, F): #Run loop over the set of objects
    human.info() #call the info function common to both
```

Output

```
Hi, I am Male
Hi, I am Female
```

Here, we have created two classes **Male** and **Female**. They share a similar structure and have the same method `info()`. However, notice that we have not created a common superclass or linked the classes together in any way. Even then, we can pack these two different objects into a tuple and iterate through them using a common **human** variable. It is possible due to **polymorphism**. We can call both the

`info()` methods by just using `human.info()` call, where **human** is first **M** (Instance of **Male**) and then **F** (Instance of **Female**).

Polymorphism and Inheritance

Like in other programming languages, the child classes in Python also inherit methods and attributes from the parent class. We can redefine certain methods and attributes specifically to fit the child class, which is known as **Method Overriding**.

Polymorphism allows us to access these overridden methods and attributes that have the same name as the parent class. Let's look at an example:

```
class Human:
    def __init__(self, name):
        self.name = name

    def info(self):
        print("Hi, I am Human")

class Male(Human):
    def __init__(self, name):
        super().__init__(name)

    def info(self):
        print("Hi, I am Male")

class Female(Human):
    def __init__(self, name):
        super().__init__(name)

    def info(self):
        print("Hi, I am Female")

M = Male("Sid")
F = Female("Zee")

for human in (M, F): #Run loop over the set of objects
```

```
    human.info() #call the info function common to both
```

Output

```
Hi, I am Male
Hi, I am Female
```

Due to polymorphism, the Python interpreter automatically recognizes that the `info()` method for object M (**Male** class) is **overridden**. So, it uses the one defined in the subclass **Male**. Same with the object F (**Female** Class).

Note: *Method Overloading*, a way to create multiple methods with the same name but different arguments, is not possible in Python.

Multiple Inheritance

- A class can be inherited from more than one superclass in Python, similar to C++. This is called **multiple inheritance**.
- In multiple inheritance, the features of all the superclasses are inherited into the subclass. The syntax for multiple inheritance is similar to single inheritance.

Example:

```
class SuperClass1:
    pass
class SuperClass2:
    pass
class SubClass(SuperClass1, SuperClass2):
    pass
```

Here, the `SubClass` class is derived from `SuperClass1` and `SuperClass2` classes and it has access to all the instance attributes and methods from both these superclasses.

Multilevel Inheritance

We can also inherit from a derived class. This is called **multilevel inheritance**. It can be of any depth in Python.

An example is given below.

```
class SuperClass:
    pass
class SubClass(SuperClass):
    pass
class SubSubClass(SubClass):
    pass
```

Here, `SubClass` is derived from `SuperClass`, and `SubSubClass` is derived from `SubClass.`

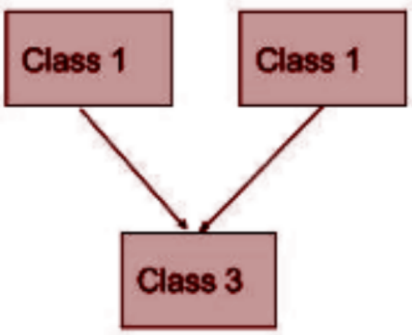

Fig: Mutiple Inheritance

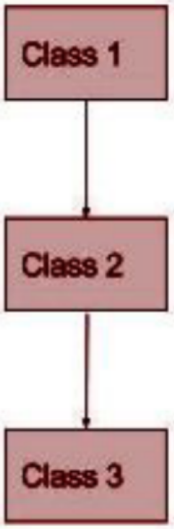

Fig: Multilevel Inheritance

Object Class

Every class in Python is directly or indirectly derived from a built-in class called the **Object** class. If a class does not inherit any other class then it is a direct child class of **Object** and if inherits some other class then it is indirectly derived. Therefore the **Object** class methods are available to all Python classes. There are three such methods provided by the **Object** class:

- **__new__() :** Used in instantiating a new object.
- **__init__():** Used for initialising a new object. We usually override this method while defining the constructors for any object. If we don't override it, the default constructor from the **Object** class gets called.
- **__str__() :** Returns a string representation of the object.

We can override all of these methods, but we usually override only **__init__()** and **__str__()**. Consider the given examples to understand it better:

```
class Circle(object):
    def __init__(self,radius):#Overriding the constructor
        self.radius = radius
c = Circle(4)
print(c) #Invokes the default __str__ method
```

Output:

```
<__main__.Circle object at 0x10794c850> #Some memory location
```

This is the output of the default **__str__** method. However, we can override it as follows:

```
class Circle(object):
    def __init__(self,radius):
        self.radius = radius
    def __str__(self): #Overriding the default __str__ method
        return "This is My circle"
c = Circle(4)
print(c)
```

Output:

```
This is My circle
```

Note: In Python 3.x, `class Circle(object)` and `class Circle` are the same.

Method Resolution Order (MRO)

Method Resolution Order(MRO) denotes the way a programming language resolves a method or an attribute. It defines the order in which the subclasses are searched when a method is called. First, the method or the attribute is searched within the subclass and then it is searched in its parent class and so on.

```
class A:
```

```
    def rk(self):
        print("In class A")
class B(A):
    def rk(self):
        print("In class B")
r = B()
r.rk()
```

Output:

```
In class B
```

In the above example, the method that is invoked is from class **B** but not from class **A**, and this is due to the Method Resolution Order(MRO). The order of priority that is followed in the above code is- **class B - > class A.**

In **multiple inheritances**, the methods are executed based on the **order specified while inheriting the classes** *(Order inside parenthesis)*. Let's look over another example to deeply understand the method resolution order:

```
class B:
    pass

class A:
    pass

class C(A, B):
    pass
```

The order in which the methods will be resolved will be **C, A, B**. This is because while inheriting the order is **A, B**.

Methods for Method Resolution Order(MRO) of a class:

To get the method resolution order of a class we can use the `mro()` method. By using this method we can display the order in which methods are resolved.

```
class B:
    pass

class A(B):
    pass

class C(A, B):
    pass

class D(C,A):
    pass

print(D.mro())
```

Output:

```
[<class '__main__.D'>, <class '__main__.C'>, <class '__main__.A'>, <class
'__main__.B'>, <class 'object'>]
```