Object Oriented Programming in Python

- In Python, object-oriented Programming (OOPs) is a programming paradigm that uses objects and classes in programming.
- Main concept of OOPs is to bind the data and the functions that work together as a single unit .
- **OOPs Concepts in Python includes:**
  - Class
  - Objects
  - Polymorphism
  - Encapsulation
  - Inheritance
  - Data Abstraction

Classes

- A class is a collection of objects.
- It is a logical entity that contains some attributes and methods.
- Classes are created by keyword class.
- Attributes are the variables that belong to a class.
- Attributes are always public and can be accessed using the dot (.) operator.
  - Eg.: Myclass.Myattribute

```
class Employee:
  # class attribute s
          name = "Dev"
          age = 40
emp1 = Employee()      # create emp1 object
emp1.name = "Raj“
emp1.age = 30
emp2 = Employee() # create another object emp2
#emp2.name = "Ali“
emp2.age = 45
# access attributes
print(f”{emp1.name}is{emp1.age}  years old”)
print(f”{emp2.name}is{emp2.age}  years old”)
print(emp1.__class__.name)   //accessing the class attribute values
```

```
Raj is 30 years old
Dev is 45 years old
Dev
```

Methods

- **The Python __init__ Method**
- The __ init __ is similar to constructors in C++ and Java.
- It is run as soon as an object of a class is instantiated
- It can take any number of arguments.
- The first argument of this method is special- **self**

- **The Python self**
- Class methods must have an extra first parameter (self) in the method definition. We do not give a value for this parameter when we call the method, Python provides it
- If we have a method that takes no arguments, then we still have to have one argument.
- This is similar to this pointer in C++ and this reference in Java.
- In __init__ , self refers to the object currently being created and for other methods, it refers to the instance whose method was called

```python
class Employee:

    # class attribute
    name = "Dev"
    age = 40
      # Instance attribute
    def __init__(self, name):
        self.name = name

# create emp1 object and Object instantiation
#emp1 = Employee() // error, since we are created the __init__() with
    an argument
emp1 = Employee("Raj")
emp1.name = "Kala" #"Raj" is overwritten as Kala

# create another object emp2
emp2 = Employee("Ali")

# access attributes
print(f"{emp1.name} is {emp1.age} years old")
print(f"{emp2.name} is {emp2.age} years old")
print(emp1.__class__.name) #accessing the cls attribute values
```

```
Kala is 40 years old
Ali is 40 years old
Dev
>
```

```
class Employee:

    # class attribute
    name = "Dev"
    age = 40
      # Instance attribute
    def __init__(self):
        self.name = "Ali"
        self.age=45
        self.id=1234
        self.dept="Accounts"

    def details(self):
        return self.name,self.age,self.id,self.dept

emp=Employee()
print("the employee details are:",emp.details())

```

```
the employee details are: ('Ali', 45, 1234, 'Accounts')
```

```
1  class Employee:
2
3      # class attribute
4      name = "Dev"
5      age = 40
6        # Instance attribute
7      def __init__(self):
8          self.name = "Ali"
9          self.age=45
10         self.id=1234
11         self.dept="Accounts"
12
13     def details(self):
14         return emp.__class__.name,self.age,self.id,self.dept
15
16 emp=Employee()
17 print("the employee details are:",emp.details())
18
19
```

the employee details are: (‘Dev', 45, 1234, 'Accounts')

Self is used within methods to call another methods from the class

```
1  class Employee:
2
3      # class attribute
4      name = "Dev"
5      age = 40
6        # Instance attribute
7      def __init__(self):
8          self.name = "Ali"
9          self.age=45
10         self.id=1234
11         self.dept="Accounts"
12
13     def details(self):
14        print("Details() returns",emp.__class__.name,self.age,self.id
              ,self.dept)
15     def job_details(self):
16         print("Job_details function calling the method-details()")
17         self.details()
18         print("Job_details() returns:",self.name,self.id,self.dept)
19
20 emp=Employee()
21 emp.job_details()
22 #print("the employee details are:",emp.details())
```

Job_details function calling the method-details()
Details() returns Dev 45 1234 Accounts
Job_details() returns: Ali 1234 Accounts

Display class attributes and methods

dir(name_of_class)

Or

dir(instance_of_class)

- returns a sorted list of attributes and methods belonging to an object.
- Returns the existing attributes and methods belonging to the class, including any special methods

Display class attributes and methods

Eg: Consider the class Employee

>>>(dir(Employee))

['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getstate__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'age', 'details', 'job_details', 'name']

ACCESSIBILITY

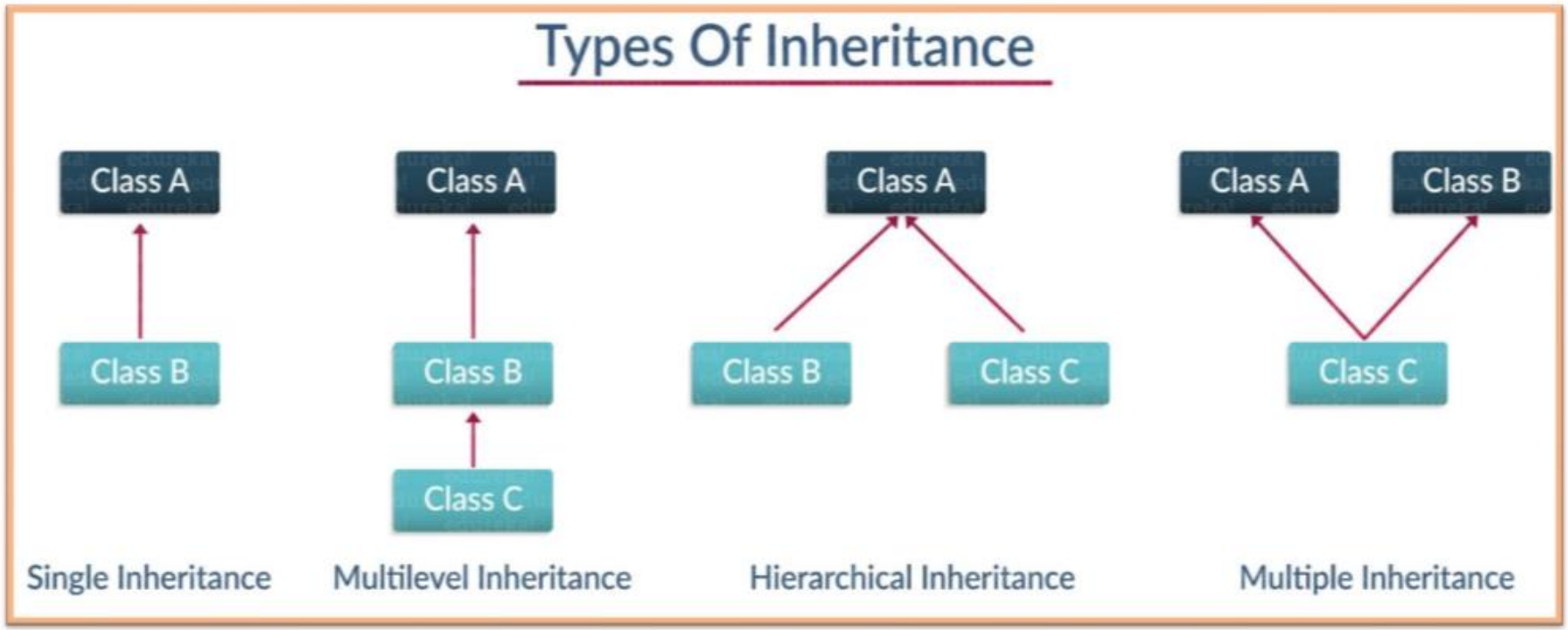
- In python, no keywords like public,private or protected.
- Default, all methods and attributes are public.
- Define private in python:
  - __Attribute
  - __Method_Name()

INHERITANCE

- Allows you to create a hierarchy of classes that share a set of properties and methods by deriving a class from another class.
- It is the capability of one class to derive or inherit the properties from another class.
- Offers reusability of code
- Transitive in nature.(Multilevel inheritance)

- Syntax:

```
Class BaseClass:
        {Body}
Class DerivedClass(BaseClass):
        {Body}
```



```
1  class Employee1():#This is a parent class
2      name="Dev"
3      age=43
4      def display(self):
5          print("Function in Super class")
6
7  class childemployee(Employee1):#This is a child class
8      def disp(self):
9          print("My name is ", self.name,"age is ",self.age)
10 emp=childemployee()
11 emp.display()
12 emp.disp()
13
```

Function in Super class
My name is Dev age is 43

MULTIPLE INHERITANCE

```
1  # Base class1
2  class Mother:
3      mothername = ""
4      def mother(self):
5          print(self.mothername)
6  # Base class2
7  class Father:
8      fathername = ""
9      def father(self):
10         print(self.fathername)
11 # Derived class
12 class Son(Mother, Father):
13     def parents(self):
14         print("Father :", self.fathername)
15         print("Mother :", self.mothername)
16
17 # Driver's code
18 s1 = Son()
19 s1.fathername = "RAM"
20 s1.mothername = "SITA"
21 s1.parents()
```

Father : RAM
Mother : SITA

```python
# Base class1
class Mother:
    mothername = "Vydehi"
    def mother(self):
        print(self.mothername)
# Base class2
class Father:
    fathername = ""
    def father(self):
        print(self.fathername)
# Derived class
class Son(Mother, Father):
    def parents(self):
        print("Father :", self.fathername)
        print("Mother :", self.mothername)

 # Driver's code
s1 = Son()
s1.fathername = "RAM"
#s1.mothername = "SITA"
s1.parents()
```

```
Father : RAM
Mother : Vydehi
```

```
1  class length:
2      l = 0
3      def length(self):
4          return self.l
5  class breadth:
6      b = 0
7      def breadth(self):
8          return self.b
9  class rect_area(length, breadth):
10     def r_area(self):
11         print("The area of rectangle with length "+str(self.l)+" units
               and breadth "+
12               str(self.b)+" units is "+str(self.l * self.b)+" sq.
                     units.")
13 obj = rect_area()
14 obj.l = int(input("Enter the required length for rectangle: "))
15 obj.b = int(input("Enter the required breadth for rectangle: "))
16 obj.r_area()
```

```
Enter the required length for rectangle: 2
Enter the required breadth for rectangle: 2
The area of rectangle with length 2 units and
breadth 2 units is 4 sq. units
```

- Advantages:
  - reusability of a code
  - higher performance and flexibility
- Disadvantages:
  - increased complexity
  - more chances of ambiguity
  - deeper coding knowledge

How to access the parent class constructors

```
# First Parent Class
class First:
    def __init__(self):
        self.greet = "I am First”
class  Second:
    def __init__(self):
self.name = "I am Second“
 # Child or Derived Class
class Child(First, Second):
    def __init__(self):
        First.__init__(self)      //we can get the attributes only if the parent class constructor is called.
        Second.__init__(self) // super().__init__()
     def  combine(self):
        print(self.greet, self.name)
        print("I am the combined function")
 obj = Child()
obj.combine()
```

OUTPUT:

I am First
I am Second
I am the combined function

What happens if methods have same name and functionalities?

- Method Overriding.
  - In a class hierarchy, if methods in sub class and super class have same name and headers, the method in sub class will only be executed
  - **Parent class method will be overridden by the sub class method.**
  - To overcome these, super() can be used.
- The "**Diamond Problem**" often occurs primarily in the case of multiple inheritance where a subclass inherits the conflicting methods from the multiple super classes that eventually create ambiguity in the code.

Example – Method Overriding

```
class Class1:
        def m(self):
                print("In Class1")
class Class2(Class1):
        def m(self):
                print("In Class2")
class Class3(Class1):
        def m(self):
                 print("In Class3")
class Class4(Class2,Class3): // if we write class Class4(Class3,Class2)
        pass                                    the output will be  “In Class3”
obj=Class4()
obj.m()
```

OUTPUT:

In Class2

Example – Method Overriding

```
class  Class1:
        def m(self):
                print("In Class1")
                print("1")
class  Class2(Class1):
        def m(self):
                print("In Class2")
                print("2")
                Class1.m(self)
                print("3")
class  Class3(Class1):
        def m(self):
                print("In Class3")
                print("4")
                Class1.m(self)
                print("5")
```

```
class  Class4(Class2, Class3):
        def m(self):
                print("In Class4")
                print("6")
                Class2.m(self)
                print("7")
                Class3.m(self)
                print("8")
obj = Class4()
obj.m()
```

OUTPUT:

In Class4
6
In Class2
2
In Class1
1
3
7
In Class3
4
In Class1
1
5
8

```
Class  Class1:
        def m(self):
                print("In Class1")
                print("1")
class  Class2(Class1):
        def m(self):
                print("In Class2")
                print("2")
                super().m()
                print("3")
class  Class3(Class1):
        def m(self):
                print("In Class3")
                print("4")
                super().m()
                print("5")
class  Class4(Class2, Class3):
        def m(self):
                print("In Class4")
                print("6")
                super().m()
                print("7")

obj = Class4()
obj.m()
```

OUTPUT:

```
In Class4
6
In Class2
2
In Class3
4
In Class1
1
5
3
7
```

Method resolution order:

In the case of multiple inheritance, a given attribute is first searched in the current class if it’s not found then it’s searched in the parent classes. The parent classes are searched in a left-right fashion and each class is searched once.

MULTI LEVEL INHERITANCE

```python
class Parent:
    def __init__(self,name):
        self.name = name
    def getName(self):
        return self.name
class Child(Parent):
    def __init__(self,name,age):
        Parent.__init__(self,name)
        self.age = age
    def getAge(self):
        return self.age
class Grandchild(Child):
    def __init__(self,name,age,location):
        Child.__init__(self,name,age)
        self.location=location
    def getLocation(self):
        return self.location
gc = Grandchild("Srinivas",24,"Hyderabad")
print(gc.getName(), gc.getAge(), gc.getLocation())
```

```
Srinivas 24 Hyderabad
```

MULTI LEVEL INHERITANCE

```
class Person:
        def __init__(self):
                print('Person- Hii')
        def age(self, a):
                print('Printing the age: ', a)
class Father(Person):
        def __init__(self):
                print('Father - Hii')
                super().__init__()      // also can used Person.__init__(self)
        def age(self, a):
                print('Printing the age(Father): ', a)
                super().age(a - 1)
class Mother(Father):
        def __init__(self):
                print('Mother - Hii')
                super().__init__()       // Father.__init__(self)
        def age(self, a):
                print('Printing the age(Mother): ', a)
                super().age(a + 5)
o = Mother()
o.age(30)
```

Output:

```
Mother - Hii
Father - Hii
Person - Hii
Printing the age(Mother):
30
Printing the age(Father):
35
Printing the age:  34
```

HIERARCHICAL INHERITANCE

```
# Base class
class Shape:
    color="Yellow"
    def __init__(self,color):
        self.color=color
    # Derived class1
class Rect(Shape):
    def __init__(self,length,breadth):
        self.length=length
        self.breadth=breadth
        print("This function is in Rect.")
        print(self.color)
    def calc_area(self):
        area=self.length*self.breadth
        print("Area Rect.")
        print(area)
```

```
# Derivied class2
class Tri(Shape):
    def __init__(self,base,height):
        self.base=base
        self.height=height
        print("This function is in Triangle.")
        print(self.color)
        Shape.__init__(self,"Red")   //modify
the value of color in Shape as Red
        print(self.color)
    def calc_area(self):
        area=self.base*self.height*0.5
        print("Area Triangle.")
        print(area)
 # Driver's code
object1 = Rect(1,2)
object2 = Tri(1,2)
object1.calc_area()
object2.calc_area()
```