OOPs

In Python, object-oriented Programming (OOPs) is a programming paradigm that uses objects and classes in programming. It aims to implement real-world entities like inheritance, polymorphisms, encapsulation, etc. in the programming.

Main Concepts of Object-Oriented Programming (OOPs)

1) **Class:** A class is a collection of objects. **Some points on Python class:**
   - 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.

2) **Objects:** The object is an entity that has a state and behaviour associated with it. It may be any real-world object like a mouse, keyboard, chair, table, pen, etc. Integers, strings, floating-point numbers, even arrays, and dictionaries, are all objects.

3) **Polymorphism:** Polymorphism simply means having many forms. For example, we need to determine if the given species of birds fly or not, using polymorphism we can do this using a single function.

4) **Encapsulation:** It describes the idea of wrapping data and the methods that work on data within one unit. A class is an example of encapsulation as it encapsulates all the data that is member functions, variables, etc.

5) **Inheritance:** Inheritance is the capability of one class to derive or inherit the properties from another class.

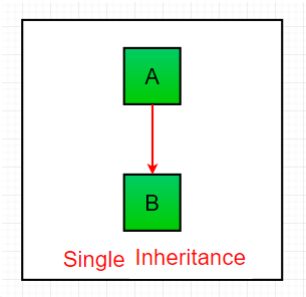
6Data Abstraction: Data Abstraction in Python can be achieved through creating abstract classes and inheriting them later.

OOPs

TYPES OF INHERITANCE

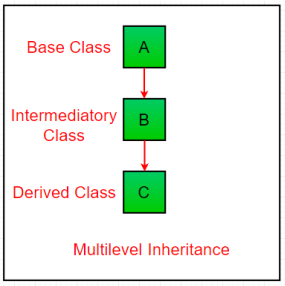
```
# 6.Inheritance
# One class inheritance the property of another class
# advantages: code reuseiblity
# While Inheritance: ==> data members
#                    ==> members function/method
#                    ==> constructor/Magical method
# Not Inheritated:   ==> Private members

# ////////////////////////////////////////////////////////Types of inheritance/////////////////////////////
# 1.Simple inheritance
# 2.Multi level inheritance
# 3.hierarchical inheritance
# 4.Mutliple inheritance (doesn't available in java)
# 5.Hybrid inheritance
```

1)Single Inheritance: Single inheritance enables a derived class to inherit properties from a singleparent class, thus enabling code reusability and the addition of new features to existing code.




```
# 1. Simple Inheritance
class A:
    def __init__(self,val):
        self.val = val
    def get_val(self):
        return self.val
class B(A):
    def __init__(self,num,val):
        super().__init__(val)
        self.num = num
    def get_num(self):
        return self.num

b = B(100,200)
print(b.get_val())
print(b.get_num())
```

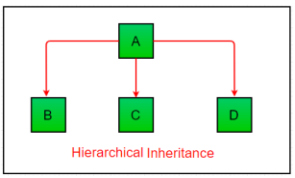
2)Multilevel Inheritance: In multilevel inheritance, features of the base class and the derived class are further inherited into the new derived class. This is similar to a relationship representing a child and grandfather.




```
# 2) Multilevel inheritance
class Product:
    def review(self):
        print("Product customer review")

class Phone1(Product):
    def __init__(self,brand,price,camera):
        self.brand = brand
        self.price = price
        self.camera = camera
    def buy(self):
        print("Buying a smartphone")
    def return_phone(self):
        print("Return an phone")

class SmartPhone1(Phone1):
    pass

s = SmartPhone1("apple",100000,13)
p = Phone1("samsung",1212,45)
s.review()
s.buy()
p.review()
```

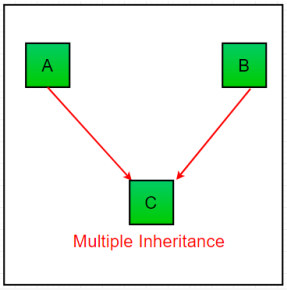
OOPs

3)Hierarchical Inheritance: When more than one derived classes are created from a single base this type of inheritance is called hierarchical inheritance. In this program, we have a parent (base) class and two child (derived) classes.




```
# 3) Hierarchical inheritance
class First:
    def __init__(self,name,brand,camera):
        self.name = name
        self.brand = brand
        self.camera = camera
class Second(First):
    def printf(self):
        print("This is second class")
class Third(First):
    pass
s = Second("iphone 12","apple",13)
print(s.name)
t = Third("Sdgfsdf","SDgfsd",31)
print(t.camera)
```

4)Multiple Inheritance: When a class can be derived from more than one base class this type of inheritance is called multiple inheritance. In multiple inheritance, all the features of the base classes are inherited into the derived class.




```
# 4) Multiple inheritance
class Phone2:
    def __init__(self,price,brand,camera):
        print("Inside phone constructor")
        self.__price = price
        self.brand = brand
        self.camera = camera
    def buy(self):
        print("Buying an smartphone")

class Product:
    def review(self):
        print("Customer review")
    def buy(self):                          # Phone2 constructor priority > Product constructor priority as ph
        print("Buying an product")          # This is known as MRO: Method Resoultion order (line 220)

class SmartPhone2(Phone2,Product):          # since Phone2 is written first therefore, Phone2 constructor get
    pass                                    # if phone2 constructor is not present then, Product constructor()

s = SmartPhone2(10000,"apple",13)
print(s.camera)
print(s.buy())   # Buying an smartphone
```

5)Hybrid Inheritance: Inheritance consisting of multiple types of inheritance is called hybrid inheritance.

```
# 5) Hybrid inheritance
class School:
  def func1(self):
  print("This function is in school.")

class Student1(School):
  def func2(self):
  print("This function is in student 1. ")

class Student2(School):
  def func3(self):
  print("This function is in student 2.")

class Student3(Student1, School):
  def func4(self):
  print("This function is in student 3.")

object = Student3()
object.func1()
object.func2()
```

OOPs

DIFFERENCE BETWEEN METHOD OVERLOADIND AND METHOD OVERRIDING (TYPES OF POLYMORPHISM)

1)METHOD OVERIDING

```
# 2. Method Overriding

class User:                    # parent class

    def login(self):
        print("login")
    def register(self):
        print("register")
    def greet(self):
        print("Hello miss")
class Student(User):            # child class
    def enroll(self):
        print("enroll")
    def review(self):
        print("review")
    def greet(self):            # Method Overriding: If same method/function with same name present in
        print("hello sir")      # child class as well as parent class then child class method will be executed.

obj = Student()
obj.login()
obj.greet()
```

```
# Method Overloading
# 1. Method Overriding: If both, parent class and child class have their same method/function name then child object will
#                       trigger child class method.
# 2. Method Overloading: Method overloading not fully work in python but can be used using brain.(line no 8)
#                        but line number 9 still called method overloading
# 3. Object Overloading
# we can't make multiplpe mehod with the same but we can overload arguments in python.



class Geometry:
    def area(self,a,b = 0):
        if(b == 0):
            print("circle:",3.14 * a * a)
        else:
            print("Rectangle:", a * b)

obj = Geometry()
print(obj.area(4))
print(obj.area(4,5))
```

Method overloading	Method overriding
It is possible only in same class.	It is possible only in derived classes.
Static methods can be overloaded	The method must be a non-virtual or static method for overriding.
Also known as static binding or early binding.	Also known as dynamic binding or late binding.
Used to implement compile time polymorphism	Used to implement run time polymorphism
It has same method name in same class with different signatures	Derived class has same method name with same signature as of base/parent class.
It helps to extend functionalities	It helps us to overwrite or change the existing functionalities.

OOPs

MAGICAL CONSTRUCTOR: it is the type of method for a class which trigger automatically on the creation of respective class object.

```
# 3. Constructor property


# Example A)
class Parent:
    def __init__(self,name):              # constructor 2
        self.name = name
    def get_name(self):
        return self.name

class Child(Parent):
    def __init__(self,value,name):         # constructor 1
        self.value = value
    def get_value(self):
        return self.value

c = Child(100,"hitendra")
print(c.get_value())
#print(c.get_name())       # constructor 1 is already present ==> constructor 2 doesn't trigired
# Example B)
class Parent:
    def __init__(self,num):                  # constructor 1
        self.__num = num       #100
    def get_num(self):
        return self.__num

class Child(Parent):
    def set_val(self,val):                    # no constructor
        self.val = val
    def get_val(self):
        return self.val        #20

c = Child(100)               # no child class constructor is present therfore constructor of
print(c.get_num())           # Parent class trigired now we can access Parent class members and methods
c.set_val(20)
print(c.get_val())
```

OOPs

SUPER KEYWORD IN PYTHON: The super() function is **used to give access to methods and properties of a parent or sibling class**. The super() function returns an object that represents the parent class.

```
# 5. .super().<method name>

# Example A)
class Phone:
    def __init__(self,price,brand,camera):
        print("Inside phone constructor")
        self.price = price
        self.brand = brand
        self.camera = camera

class SmartPhone(Phone):
    def __init__(self,price,brand,camera,os,ram):
        print("phele yaha")
        print("smartphone constructor")
        super().__init__(price,brand,camera)   # Because of this line we can use line 125 and it should b
        self.os = os
        self.ram = ram
        super().__init__(price,brand,camera)

s = SmartPhone(10000,"apple",21,"android",4)
print(s.os)
print(s.ram)
```

```
# Example B)
class A:
    def __init__(self,val):
        self.val = val
    def get_val(self):
        return self.val
class B(A):
    def __init__(self,num,val):
        super().__init__(val)
        self.num = num
    def get_num(self):
        return self.num

b = B(100,200)
print(b.get_val())
print(b.get_num())
```

```
# Example D)
class Phone:
    def __init__(self,price,brand,camera):
        print("Inside constructor")
        self.__price = price
        self.brand = brand
        self.camera = camera
    def buy(self):
        print("Buying a phone")

class Smartphone(Phone):
    def buy(self):
        print("Buying a SmartPhone")
        super().buy()     #invoke buy function/method of super class can't be used outside of the class

s = Smartphone(2000,"apple",13)
# s.super.buy()  will give error invalid outiside of the class
s.buy()
```

OOPs

Objects

```
# 7.Objects
# python objects are also mutable same as list,dictionaries and sets.
# when objects are passed inside an function changes reflect to the main object(cust).




# A) Objects as Mutables
class Customer:
    def __init__(self,name):
        self.name = name

def greet(customer):              # changes made with customer reflect to outer cust as they
    customer.name = "Bupesh"
    print(customer.name)
    print(id(customer))           # pointing to same memory address.(customer and cust)

cust = Customer("hitendra")
greet(cust)
print(id(cust))
print(cust.name)
```

```
# B) Objects as Lists
class Customer:
    def __init__(self,name,age):
        self.name = name
        self.age = age
    def intro(self):
        print("I am",self.name,"and I am",self.age,"years old")

c1 = Customer("Hitendra",18)
c2 = Customer("Ujesh",24)
c3 = Customer("Rakesh",30)

L = [c1,c2,c3]            # Mutliple objects can also be formed as an list

for i in L:
    i.intro()       # when i = c1 then c1.intro(<c1>) copy of c1 is created in self.
```

```
# C) Objects as variables
class Bank:
    def __init__(self,name,gender):
        self.name = name
        self.gender = gender

def greet(customer):                    # Now customer act same as object
    if(customer.gender == 'male'):
        print("Hello",customer.name,"sir")
    else:
        print("Hello",customer.name,"mam")

    cust = Bank("hitendra","male")
    return cust                         # we can also return an object from method/function

obj = Bank("ankita","female")
new_cust = greet(obj)                   # class objects can also passed in fucntions/method
print(new_cust.name)
print(new_cust.gender)
```

OOPs

CLASS INSTANCE AND COPIED VALUE