```
# class Book:

#     def __init__(self,author,name):
#        self.name = name
#        self.author = author
#        self.price = 0


#     def set_price(self,price): #comaparable to setter
#         self.price = price

#     def get_price(self): #comaparable to getter
#       return self.price

#     def details(self):
#            print("Book Name:", self.name,
#           "\nAuthor:", self.author,
#           "\nPrice:", self.price, "Taka")

#         #Design/Bluprint

#-------------------------------------------------



# b1 = Book("Opekkha","Humayun Ahmed") #b1=object/instance, variable_name=Class()
# b1.details()

# b1.set_price(255)
# b1.details()




# class Cat:
#     def __init__(self, color, action):
#         self.color = color
#         self.action = action

#     def view1(self):
#         print(self.color, "Cat is", self.action)

#     def compare(self, ct):
#         if self.action == ct.action:
#             print("Both cats are", ct.action) #or self.action

#         else:
#             print("They are different")



#-------------------------------------------------
```

```
# c1 =Cat("White","Jumping")
# c2 =Cat("Brown","Jumping")

# c1.view1()
# c2.view1()

# c1.compare(c2)   #Self receives c1, ct receives c2 (pass by reference)



# class Cat:
#     def __init__(self,color,action):
#         self.color = color
#         self.action = action

#     def view(self, num, clr):
#         num = num + 5
#         # clr1 = clr
#         clr[0] = "Green"
#         print("Method inside:", num)
#         print("Method inside:", clr)

# #-------------------------------------------
# colors = ["Black","Red","Yellow","Blue"]
# c1= Cat("White","Jumping")
# x=55
# c1.view(x,colors)
# print("Method outside", x)  #Immutable objects (e.g., int, float, str, tuple, bool) are pass
# print("Method outside:", colors)   #Mutable objects (e.g., list, dict, set, bytearray, custo




#Method overloading => multiple method with same name but different parameter


# class my_calculator:

#     def product(self, *nums):
#         pro = 1
#         for x in nums:
#             pro = pro*x

#         print(pro)


# c1 = my_calculator()
# c1.product(2,3,4,5,6)
```

```
# from multipledispatch import dispatch
# class my_calculator:

#     @dispatch(int,int)
#     def product(self,a,b):
#         print(a*b)

#     @dispatch(int,int,int)
#     def product(self,a,b,c):
#         print(a*b*c)

#     @dispatch(int)
#     def product(self,a):
#         print(a)

#     @dispatch(str,str)
#     def product(self,a,b):
#         print(int(a) * int(b))

#     @dispatch(str,int)
#     def product(self,a,b):
#         print(int(a) * b)




# c1 =my_calculator()
# c1.product(9)
# c1.product(4,5)
# c1.product(4,5,6)
# c1.product("5","6")
# c1.product("7",8)




# class Student:
#     def __init__(self,name,id):
#         self.name = name
#         self.id = id
#     def __init__(self,name,id,cg):
#         self.name = name
#         self.id = id
#         self.CGPA = cg


#         #Last_method is working with same name
# #----------------------------------------

# s1 = Student("Carol",33)
# s2 = Student("Nazmul",11,3.58)
```

```
#Constructor_overloading
# class Student:

#     def __init__(self, *info):
#         if(len(info)==3):
#             self.name = info[0]
#             self.id = info[1]
#             self.CGPA = info[2]

#         elif(len(info)==2):
#             self.name = info[0]
#             self.id = info[1]


#         print("A student object is created")









#----------------------------------------

# s1 = Student("Carol",33)
# s2 = Student("Nazmul",11,3.58)
# s3 = Student("Make")
# s4 = Student() #Unknown_number_arguemnet




# class Student:

#     def __init__(self, **info):
#         if(len(info)==3):
#             self.name = info['name']
#             self.id = info['id']
#             self.CGPA = info['cg']

#         elif(len(info)==2):
#             self.name = info['name']
#             self.id = info['id']


#         print("A student object is created")
```

```
# #-------------------------------------

# s1 = Student(name="Carol",id=33)
# s2 = Student(name="Nazmul",id=11,cg=3.58)
# s3 = Student(name="Mike")
# s4 = Student() #Unknown_keyword_arguemnet







#Operator_overloading
# class data:
#     def __init__(self,x):
#         self.x = x

#         #adding_two_object

#     def __add__(self,other):
#         return self.x + other.x



# num1 = data(10)
# num2 = data(20)

# print(num1 + num2)  #(num1).__add__(num2)


# str1 = data('cse')
# str2 = data('111')

# print(str1 + str2) #(str1).__add__(str2)









# class data:
#     def __init__(self,x):
#         self.x = x

#     def __lt__(self,other):
#         if(self.x < other.x):
#             return "num1 is less than num2"

#         else:
#             return "num2 is less than num1"
```

```
# num1 = data(10)
# num2 = data(20)

# print(num1 < num2)




# n1 = 15
# n2 = 25
# print(n1 + n2)

# print((n1).__add__(n2))
# print(type(n2))




# class House:
#     def __init__(self,w,d):
#         self.window = w
#         self.door= d

#     def view(self):
#         print(f"The house has {self.window} windows amd {self.door} door's ")

#     def __add__(self,other):
#         new_window = self.window + other.window
#         new_door = self.door + other.door
#         obj = House(new_window,new_door)
#         return obj
#         # return f"New house has {new_window} window and {new_door} door"

# #-----------------------------------------------------------------
# h1 = House(6,2)
# h2 = House(4,1)
# h1.view()
# h2.view()
# h3 = h1 + h2
# h3.view()
# print(h1 + h2) #(h1).___add__(h2)




# class Student:
#     counter = 0
```

```
#     def __init__(self,name,id):
#         self.name = name
#         self.id = id
#         Student.counter+=1

#     def details(self):
#         print("Name:",self.name,"ID:",self.id,"Student count",Student.counter)

# #---------------------------------------------------------------------

# s1 = Student("Nazmul",11)
# s2 = Student("Carol",22)
# s3 = Student("Mike", 33)
# s2.details()




# Encapsulation(accesee with get and set method of private variable)
# class Student:

#     def __init__(self,name,id):
#         self.name = name
#         self.__id = id #.__id__ ---> Encapsulation

#     def details(self):
#         print("Name:",self.name,"ID:",self.__id)

#     def set_id(self,id):
#         if (id>0):
#             self.__id = id
#         else:
#             print("Invalid ID!")

#     def get_id(self):
#         return self.__id

#     def set_name(self,name):
#         self.name = name
#     def get_name(self):
#         return self.name
# #---------------------------------------------------------------------

# s1 = Student("Bob",11)
# s2 = Student("Carol",24)
# s2.set_id(33)
# s1.set_name("Mike")
# s1.details()
# s2.details()
```

```
#Method_encapsulation

# class ABC:

#     def __init__(self, x, y):
#         self.x = x
#         self._y = y

#     def details(self):
#         print("X:",self.x,"Y:",self._y)
#         self.__method()

#     def __method(self):
#         print("Private method executed")

# #-------------------------------------------------------------------------

# s1 = ABC(5, 6)
# s2=ABC(15, 17)

# s1.details()

# # s1.__method()




# Class or Static variable

# class Player:
#     team_run = 0  # Class or Static variable , team_run--->class property

#     def __init__(self,run):
#         self.run = run   #instance variabel


#     def hit_four(self):
#         self.run += 4
#         Player.team_run += 4

#     def hit_six(self):
#         self.run += 6
#         Player.team_run += 6


#-------------------------------------------------------------------------

# print("Team Run:",Player.team_run)
```

```
# Shakib = Player(0)
# Tamim = Player(0)



# Tamim.hit_four()
# Tamim.hit_four()
# Tamim.hit_four()

# Shakib.hit_six()

# print("Shakib:", Shakib.run)
# print("Tamim:", Tamim.run)


# print(Player.team_run)  #called w.r.to class
# print("Team Run:",Player.team_run)
# print("Team Run:",Tamim.team_run)
# print("Team Run:",Shakib.team_run)



# print("Tamim", Tamim.__dict__)
# print("Shakib", Shakib.__dict__)









# class Student:
#     counter = 0


#     def __init__(self,name,id):
#         self.name = name
#         self.id = id
#         Student.counter += 1



#     def details(self):
#         print("Name:",self.name,"ID:",self.id,"Student count",Student.counter )

# #---------------------------------------------------------------------------

# print("Student Count",Student.counter)
# s1 = Student("Bob",11)
# s2 = Student("Carol",22)
# s3 = Student("Mike",33)
```

```
# s2.details()



'''Types of methods:
1. Instance method
2. Class method
3. Static method'''



# Class_Method
# class  Employee:
#     org_name = "Google"
#     def __init__(self,name):
#         self.name = name

#     @classmethod
#     def info(cls):
#         return cls.org_name

# print(Employee.info())  #No need to create object







#Staric_Method
# class  Employee:
#     org_name = "Google"
#     def __init__(self,name):
#         self.name = name

#     @staticmethod
#     def details(): #not take class or self
#         print("This is an Employe class")

# Employee.details()  #No need to create object







# class Student:
#     uni_name= "KUET"


#     def __init__(self,name,id):
#         self.name = name
#         self.id = id
```

```
#     def details(self):   #instance method
#         print("Name:",self.name,"ID:",self.id,Student.uni_name )




#     @classmethod
#     def up_uni_name(cls,u_name):
#         cls.uni_name = u_name




# #-------------------------------------------------------------------------------

# s1 = Student("Bob" , 11)
# s2 = Student("Carol" , 22)
# s1.details()
# s2.details()
# Student.up_uni_name("Brac University")
# s1.details()
# s2.details()
```

```
# class Student:
#     uni_name= "KUET"


#     def __init__(self,name,id):
#         self.name = name
#         self.id = id



#     def details(self):   #instance method
#         print("Name:",self.name,"ID:",self.id,Student.uni_name )




#     @classmethod
```

```
#     def up_uni_name(cls,u_name):
#         cls.uni_name = u_name

#     @classmethod
#     def from_string(cls,info):
#         name,id = info.split('-')
#         obj = Student(name,id)
#         return obj




# #-------------------------------------------------------------------------------

# s1 = Student("Bob" , 11)
# s2 = Student.from_string("Carol-47")
# s1.details()
# s2.details()









# class Student:
#     uni_name= "KUET"


#     def __init__(self,name,id):
#         self.name = name
#         self.__id = id



#     def details(self):   #instance method
#         print("Name:",self.name,"ID:",self.__id,Student.uni_name )






#     @classmethod
#     def up_uni_name(cls,u_name):
#         cls.uni_name = u_name

#     @staticmethod
#     def check_department(id):
#         if id[3:5] == "01":print("CSE")
#         elif id[3:5] == "41": print("CS")
```

```
# #-------------------------------------------------------------------------------

# s1 = Student("Bob" , 11)
# s2 = Student("Carol",47)
# s1.details()
# s2.details()
# Student.check_department("15341007")









#Inheritence

#Single_Inheritence
# class Animal:
#     def __init__(self,name):
#         self.name = name

#     def eat(self):
#         print(self.name, "is eating")

#-------------------------------------------------------------------------

# class Dog(Animal):

#     def bark(self):
#         print(self.name, "is barking")

#-------------------------------------------------------------------------
#parent class can not access child class

# a1 =  Animal("Jack")
# d1 = Dog("Rover")
# d1.bark()
# d1.eat()
# a1.eat()

# isinstance(Object, ClassName)
# print(isinstance(a1,Dog))
# issubclass(Class, ClassName)
# print(issubclass(Dog,Animal))




#Multilevel
# class ParentClass:
#     def method1(self):
```

```
#         print("This method1 is in ParentClass")

# class ChildClass1(ParentClass):
#     def method2(self):
#         print("This method2 is in ChildClass1")

# class ChildClass2(ChildClass1):
#     def method3(self):
#         print("This method3 is in ChildClass2")

# ch1 = ChildClass2()
# ch1.method1()
# ch1.method2()
# ch1.method3()




#Hierarchical Inheritence --->Multipe class is derived from a base class
#Multiple Iheritence --->Where a class derived from multiple parent class: ChildClass(ParentCl
#Hybrid Inheritence ---> Mixed
#super() is used in child class to call parent class
#Method_Overriding : Method with same name don't execute parent method  but it's own new metho




#Hierarchical Inheritance

# class Student:
#     def __init__(self,name,id):
#         self.name = name
#         self.id = id

#     def details(self):
#         print("Name:",self.name,"ID:",self.id)
# #-------------------------------------------------------------------------
# class CSEstudent(Student):
#     def __init__(self,name,id,labs):
#         super().__init__(name,id)
#         self.no_of_labs = labs

#     def cry(self):
#         print("CSE student is crying because of", self.no_of_labs,"labs")
# #-------------------------------------------------------------------------
# class BBAstudent(Student):
#     def party(self):
#         print("All day party")
# #-------------------------------------------------------------------------
# s1 = CSEstudent("Bob",11,3)
# s2 = BBAstudent("Carol",36)
# print(s1.__dict__)
```

```
# s1.details()
# s2.details()

# s1.cry()
# s2.party()

# print(help(s1))




# Multilevel Inheritance
# class Student:
#     def __init__(self,name,id):
#         self.name = name
#         self.id = id

#     def details(self):
#         print("Name:",self.name,"ID:",self.id)
# #-----------------------------------------------------------------------
# class CSEstudent(Student):
#     def __init__(self,name,id,labs):
#         super().__init__(name,id)
#         self.no_of_labs = labs

#     def cry(self):
#         print("CSE student is crying because of", self.no_of_labs,"labs")
# #-----------------------------------------------------------------------
# class CSEfresher(CSEstudent):
#     def enroll_CSE110(self):
#         print(self.name,"Enrolled in CSE110")


# #-------------------------------------------------
# s1 = CSEstudent("Bob",11,3)
# s2 = CSEfresher("Carol",55,1)
# s2.details()
# s1.details()
# s2.enroll_CSE110()




#Multiple Inheritance
# class A:
#     def __init__(self):
#         print("__init__ of class A")

#     def method1(self):
#         print("Method1 of class A")
```

```
# #-------------------------------------------------------
# class B:
#     def __init__(self):
#         print("__init__ of class B")

#     def method1(self):
#         print("Method1 of class B")
# #-------------------------------------------------------
# class C(A,B):   #left is called first
#     def __init__(self):
#         #super().__init__()
#         #B.__init__(self)
#         print("__init__ of class C class")

#     def method2(self):
#         print("Method2 of class C")
# #-------------------------------------------------------
# c1 = C()
# B.method1(c1)
```

```
#Variable Overriding
# class Animal:

#     def __init__(self,name):
#         self.name = name
#         self.color = "White"

#     def eat(self):
#         print(self.color,self.name,"is eating")
# #-----------------------------------------------------------------
# class Dog(Animal):

#     def __init__(self,name,color):
#         super().__init__(name)
#         self.color = color

#     def bark(self):
#         print(self.color,self.name,"is barking")
# #-----------------------------------------------------------------
# d1 = Dog("Rover","Brown")
# print(d1.__dict__)
# d1.bark()
# d1.eat()
```

```
#Method Overriding

# class A:
#     def __init__(self):
#         print("__init__ of class A")

#     def method1(self):
#         print("Olpo study")

#     def method2(self):
#         print("You will get all of my property and methods")


# #-----------------------------------------------------
# class B(A):
#     def __init__(self):
#         pass
#         #print("__init__ of class B")

#     def method1(self): #Method name will same in method overriding,parameter equal or not

#         print("Always party")
#         super().method1()
# #-----------------------------------------------------
# b1 = B()
# b1.method1()
# b1.method2()




# __str__() method overriding
# class Student:

#     def __init__(self,name,id):
#         self.name = name
#         self.id = id
#         print(self)  #print location __str__() [method]

#     def __str__(self):
#         return "My name is " + self.name #must return string

# #-------------------------------------------------------------------

# s1 = Student("Bob",11)
# s2 = Student("Carol",22)

# # print(s1)  #by default call s1.__str__()
# #print(s1.__str__())
# # print(s2)
```

```
# Composition (has a relationship)
# class Engine:
#     def __init__(self,cc):
#         self.capacity = cc

#     def start(self):
#         print("Engine started")

#     def stop(self):
#         print("Engine stopped")
# #-----------------------------------------------------------------
# class Car(Engine):
#     def __init__(self,name,cc):
#         self.name = name
#         self.engine = Engine(cc) #Object is created



#     def run(self):
#         print(self.engine)
#         self.engine.start()
#         print("Car is running")
# #-----------------------------------------------------------------
# c1 = Car("BMW", 2000)
# c1.run()






#Abstract Class & Method
# from abc import ABC,abstractmethod
# class A(ABC):
#     @abstractmethod
#     def method1(self):
#         pass        #Abstract method don't have implementation

# # a =A()    #we will not able to create object from abstract class
# # -----------------------------------------------------------------

# class B(A):
#     @abstractmethod
#     def method2(self):
#         pass
# #-----------------------------------------------------------------
# class C(B):
#     def method1(self):
#         print("Method1 is overridden")
```

```
#     def method2(self):
#         print("Method2 is overridden")
# c = C()
# c.method1()
# c.method2()




# from abc import ABC,abstractmethod

# class Animal(ABC):
#     @abstractmethod
#     def make_sound(self):
#         pass
#         #print("AB") it will work but there should not be any body

#     def eat(self):
#         print("I am eating")

# #-----------------------------------------------------------------
# class Dog(Animal):
#     def make_sound(self):
#         print("Dog is barking")
# #-----------------------------------------------------------------
# class Cat(Animal):
#     def make_sound(self):
#         print("Meow Meow")
# #-----------------------------------------------------------------
# class Snake(Animal):
#     def make_sound(self):
#         print("Hiss Hiss")
# #-----------------------------------------------------------------

# d1 = Dog()
# d1.eat()
# d1.make_sound()
# c1 = Cat()
# c1.eat()
# c1.make_sound()
# s1 = Snake()
# s1.make_sound()
```