Method overriding

Sometimes you may wanna change the method which is defined on parent class, you can do as following,

class Shape:

    def \_\_init\_\_(self,x,y):

        self.x = x

        self.y = y

    def area(self):

        return self.x \* self.y

class Circle(Shape):

    def \_\_init\_\_(self,radius):

        self.radius = radius

    def area(self):

        return 3.14 \* (self.radius\*\*2)

rec = Shape(3,5)

print(rec.area())

c = Circle(5)

print(c.area())

you can do something like this also:

class Shape:

    def \_\_init\_\_(self,x,y):

        self.x = x

        self.y = y

    def area(self):

        return self.x \* self.y

class Circle(Shape):

    def \_\_init\_\_(self,radius):

        self.radius = radius

        super().\_\_init\_\_(radius,radius)

    def area(self):

        return 3.14 \* super().area()

rec = Shape(3,5)

print(rec.area())

c = Circle(4)

print(c.area())

this will also work I used super() for init method from parent class and calling area function from upper class.

Operator Overloading

class Vector:

    def \_\_init\_\_(self,i,j,k):

        self.i = i

        self.j = j

        self.k = k

    def \_\_str\_\_(self):

        return f"{self.i}i + {self.j}j + {self.k}k"

v = Vector(3,6,5)

print(str(v))

v1 = Vector(1,2,9)

print(str(v1))

print(v + v1) # this will throw an error cause no add operation for these two are defined

lets define the add function for them:

class Vector:

    def \_\_init\_\_(self,i,j,k):

        self.i = i

        self.j = j

        self.k = k

    def \_\_str\_\_(self):

        return f"{self.i}i + {self.j}j + {self.k}k"

    def \_\_add\_\_(self,x):

        return f"{self.i + x.i}i + {self.j + x.j}j + {self.k + x.k}k"

v = Vector(3,6,5)

print(str(v))

v1 = Vector(1,2,9)

print(str(v1))

print(v + v1) # 4i + 8j + 14k

now the error is that v + v1 is returning a string but we want vector on the output:

class Vector:

    def \_\_init\_\_(self,i,j,k):

        self.i = i

        self.j = j

        self.k = k

    def \_\_str\_\_(self):

        return f"{self.i}i + {self.j}j + {self.k}k"

    def \_\_add\_\_(self,x):

        # return f"{self.i + x.i}i + {self.j + x.j}j + {self.k + x.k}k"

        return Vector(self.i + x.i , self.j + x.j , self.k + x.k)

v = Vector(3,6,5)

print(str(v) + ": V")

v1 = Vector(1,2,9)

print(str(v1) + ": V1")

addVV1 = v + v1

print(addVV1) # now it will return vector object not str so you can apply operations on it

print(addVV1+v1+v)  # now you can add as many vectors you want

how this addition is working ? read out the comments you will get to know

class Vector:

    def \_\_init\_\_(self,i,j,k):

        self.i = i

        self.j = j

        self.k = k

    def \_\_str\_\_(self):

        return f"{self.i}i + {self.j}j + {self.k}k"

    def \_\_add\_\_(self,x):

        # return f"{self.i + x.i}i + {self.j + x.j}j + {self.k + x.k}k"

        print("I ran..")

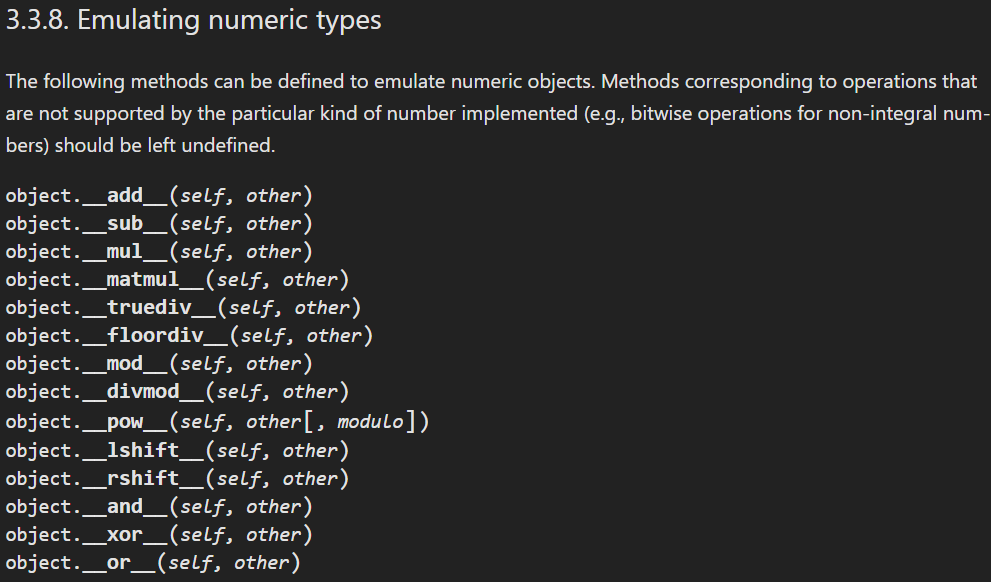
        return Vector(self.i + x.i , self.j + x.j , self.k + x.k)

v = Vector(3,6,5)

v1 = Vector(1,2,9)

print(v+v+v1) # two times I ran will print because first it will add v+v then it will return a vector object and then it will take that vector object and add v1 to it and then it will return a vector object

you can visit <https://docs.python.org/3/reference/datamodel.html> and see ke konsa dender method kis operator ko overwrite krta hai.



You can read out through this link and same as \_\_add\_\_ you can overwrite other methods.

Single level inheritance

Ek class se dosri class bangayi yeh hai single level inheritance.

class Animal:

    def \_\_init\_\_(self, name, species):

        self.name = name

        self.species = species

    def make\_sound(self):

        print("Sound made by the animal")

class Dog(Animal):

    def \_\_init\_\_(self, name, breed):

        Animal.\_\_init\_\_(self, name, species="Dog")

        self.breed = breed

    def make\_sound(self):

        print("Bark!")

Multiple inheritance

Ek child class hogi or uska ek se ziada parent hoga.

Multiple inheritance is a powerful feature in object-oriented programming that allows a class to inherit attributes and methods from multiple parent classes. This can be useful in situations where a class needs to inherit functionality from multiple sources.

class Employee:

    def \_\_init\_\_(self,name,id):

        self.name = name

        self.id = id

    def show(self):

        print(f"The name is {self.name}")

class Dancer:

    def \_\_init\_\_(self,dance):

        self.dance = dance

    def show(self):

        print(f"The dance is {self.dance}")

class DancerEmployee(Employee,Dancer):

    def \_\_init\_\_(self,dance,name,id):

        self.dance = dance

        self.name = name

        self.id = id

    def \_\_str\_\_(self):

        return f"Dance = {self.dance}, Name = {self.name}, ID = {self.id}"

o = DancerEmployee("Break","zain",42804)

print(o)

print(o.show()) # joh phele parent class given hogi uska show method call hojayega

mro() (Method Resolution Order) in Python shows the order in which classes are searched for methods and attributes when using inheritance. It helps Python determine which method to call when multiple classes are involved.

class Employee:

    def \_\_init\_\_(self,name,id):

        self.name = name

        self.id = id

    def show(self):

        print(f"The name is {self.name}")

class Dancer:

    def \_\_init\_\_(self,dance):

        self.dance = dance

    def show(self):

        print(f"The dance is {self.dance}")

class DancerEmployee(Dancer,Employee):

    def \_\_init\_\_(self,dance,name,id):

        self.dance = dance

        self.name = name

        self.id = id

    def \_\_str\_\_(self):

        return f"Dance = {self.dance}, Name = {self.name}, ID = {self.id}"

o = DancerEmployee("Break","zain",42804)

print(DancerEmployee.mro()) # mro will return the method resolution order which means which class is first and after that which class has priority

Multilevel inheritance

Multilevel inheritance is a type of inheritance in object-oriented programming where a derived class inherits from another derived class. This type of inheritance allows you to build a hierarchy of classes where one class builds upon another, leading to a more specialized class.

class Animal:

    def \_\_init\_\_(self, name, species):

        self.name = name

        self.species = species

    def show\_details(self):

        print(f"Name: {self.name}")

        print(f"Species: {self.species}")

class Dog(Animal):

    def \_\_init\_\_(self, name, breed):

        Animal.\_\_init\_\_(self, name, species="Dog")

        self.breed = breed

    def show\_details(self):

        Animal.show\_details(self)

        print(f"Breed: {self.breed}")

class GoldenRetriever(Dog):

    def \_\_init\_\_(self, name, color):

        Dog.\_\_init\_\_(self, name, breed="Golden Retriever")

        self.color = color

    def show\_details(self):

        Dog.show\_details(self)

        print(f"Color: {self.color}")

Hybrid inheritance

**Hybrid inheritance** is a combination of **two or more types of inheritance**

class Baseclass:

    pass

class derived1(Baseclass): # till now you can see that it is a single inheritance

    pass

class derived2(Baseclass):

    pass

class derived3(derived1,derived2): # from derived1 to till now you can say multiple inheritance where two classes are used as parent of derived class.

    pass

Hierarchical inheritance

Bohot sari classes single base class se inherit horahi hein

Hierarchical Inheritance is a type of inheritance in Object-Oriented Programming where multiple subclasses inherit from a single base class. In other words, a single base class acts as a parent class for multiple subclasses.

Just creating a hierarchy of classes , One parent, multiple children.

class Animal:

    def \_\_init\_\_(self, name):

        self.name = name

    def show\_details(self):

        print("Name:", self.name)

class Dog(Animal):

    def \_\_init\_\_(self, name, breed):

        Animal.\_\_init\_\_(self, name)

        self.breed = breed

    def show\_details(self):

        Animal.show\_details(self)

        print("Species: Dog")

        print("Breed:", self.breed)

class Cat(Animal):

    def \_\_init\_\_(self, name, color):

        Animal.\_\_init\_\_(self, name)

        self.color = color

    def show\_details(self):

        Animal.show\_details(self)

        print("Species: Cat")

        print("Color:", self.color)