1. **Multipath Inheritance**

**Problem Statement:**

You are tasked with modelling a hierarchy of vehicles with a focus on both land and water transportation. Create a class hierarchy involving three classes: **LandVehicle**, **WaterVehicle**, and **AmphibiousVehicle**, demonstrating multipath inheritance.

1. Define the **LandVehicle** class with the following attributes and methods:
   * Attributes:
     + **name** (string): the name or model of the land vehicle.
   * Methods:
     + **\_\_init\_\_(self, name)**: a constructor to initialize the **name** attribute.
     + **drive(self)**: a method that simulates the land vehicle's movement.
2. Define the **WaterVehicle** class with the following attributes and methods:
   * Attributes:
     + **name** (string): the name or model of the water vehicle.
   * Methods:
     + **\_\_init\_\_(self, name)**: a constructor to initialize the **name** attribute.
     + **sail(self)**: a method that simulates the water vehicle's movement on water.
3. Define the **AmphibiousVehicle** class, which inherits from both **LandVehicle** and **WaterVehicle** classes. The **AmphibiousVehicle** class should have an additional attribute and method:
   * Attribute:
     + **propulsion\_type** (string): the type of propulsion the vehicle uses (e.g., "Wheels" for land and "Propeller" for water).
   * Methods:
     + **\_\_init\_\_(self, name, propulsion\_type)**: a constructor that calls the constructors of both parent classes and sets the **propulsion\_type** attribute.
     + **travel(self)**: override the **travel** method from the parent classes to simulate the movement of the amphibious vehicle on both land and water.

Create an instance of the **AmphibiousVehicle** class, set its name and propulsion type, and demonstrate its functionality by calling the **travel** method, which should simulate movement on both land and water.

class LandVehicle:

    #instance attribute

    def \_\_init\_\_(self, name):

        self.name=name

    #Method

    def drive(self):

        print(f"{self.name} is moving on land")

class WaterVehicle:

    #instance attribute

    def \_\_init\_\_(self, name):

        self.name=name

    #Method

    def sail(self):

        print(f"{self.name} is moving on water")

class AmphibiousVehicle(LandVehicle,WaterVehicle):

    #instance attribute

    def \_\_init\_\_(self, name, propulsion\_type):

        self.name=name

        self.propulsion\_type=propulsion\_type

    #Method

    def travel(self):

        print(f"{self.name} is using {self.propulsion\_type} for propulsion")

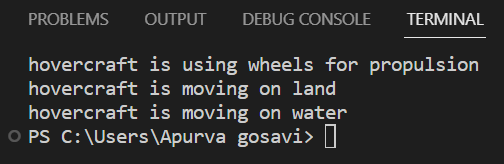
        self.drive()       #call parents method here

        self.sail()

amphibious\_vehicle = AmphibiousVehicle("hovercraft","wheels")     #inverted commas

amphibious\_vehicle.travel()

Top of Form



1. The **LandVehicle** and **WaterVehicle** classes are defined with their attributes and methods for land and water vehicles, respectively.
2. The **AmphibiousVehicle** class inherits from both **LandVehicle** and **WaterVehicle**. It has an additional attribute **propulsion\_type** and an overridden **travel** method that simulates movement on both land and water.
3. An instance of the **AmphibiousVehicle** class is created, and its **travel** method is called to demonstrate the vehicle's movement on both land and water using the specified propulsion type.

In this output:

* The first line indicates the name of the amphibious vehicle ("AmphiCar") and the type of propulsion it is using ("Wheels and Propeller").
* The second line shows that the amphibious vehicle is moving on land using wheels.
* The third line indicates that the amphibious vehicle is sailing on water using a propeller.

This demonstrates the functionality of the **AmphibiousVehicle** class, which can simulate movement on both land and water, depending on the specified propulsion type.

Q. Create three classes: "Person," "Student," and "Teacher." "Student" and "Teacher" should inherit from "Person." Implement a method in "Person" to display the name and age, and add unique methods to "Student" and "Teacher" to display their respective roles.

# Question 3 - multiple inheritance

class Person():

    def \_\_init\_\_(self,name,age):

        self.name=name

        self.age=age

    def display(self):       #self inside the brackets

        print(f"{self.name} and age is {self.age}")

class Teacher(Person):

    def \_\_init\_\_(self,name,age,role):

        Person.\_\_init\_\_(self,name,age)

        self.role=role

    def display(self):

        print(f"Teacher Info: {self.name} and age is {self.age} and she is a {self.role}")

class Student(Person):

    def \_\_init\_\_(self,name,age,role):

        Person.\_\_init\_\_(self,name,age)

        self.role=role

    def display(self):

        print(f"Student Info: {self.name} and age is {self.age} and she is a {self.role}")

student=Student('Apurva','23','Student')

teacher=Teacher('Aparna','50','Teacher')

student.display()

teacher.display()



Instances of the **Student** and **Teacher** classes are created, and their information and roles are displayed using the methods defined in each class. The **super()** function is used to call the constructor of the base class, ensuring that the name and age attributes are initialized correctly.

2. Encapsulation

**Problem Statement:**

You are tasked with designing a class to model a basic bank account. The account should have a balance that can be deposited into, withdrawn from, and checked.

1. Define a class **BankAccount** with the following attributes and methods:
   * Attributes:
     + **balance** (float): the current balance of the bank account. Initially set to 0.0.
   * Methods:
     + **\_\_init\_\_(self)**: a constructor that initializes the balance attribute.
     + **deposit(self, amount)**: a method that allows you to deposit a specified amount into the account. Ensure that the amount is a positive number and update the balance accordingly.
     + **withdraw(self, amount)**: a method that allows you to withdraw a specified amount from the account if the balance is sufficient. Ensure that the amount is a positive number and that the balance does not go negative. If the withdrawal is allowed, update the balance.
     + **get\_balance(self)**: a method that returns the current balance of the account.
2. Demonstrate the usage of the **BankAccount** class by creating an instance, making deposits, withdrawals, and checking the balance.
3. Overloading and Overriding

**Problem Statement:**

You are tasked with modeling a hierarchy of shapes. Create a base class **Shape** and two derived classes, **Circle** and **Rectangle**. In this problem, you will explore method overloading and method overriding.

1. Define the **Shape** class with the following methods:
   * **area(self)**: a method that calculates and returns the area of a generic shape. Initially, return 0.
2. Define the **Circle** class, which inherits from the **Shape** class, with the following attributes and methods:
   * Attributes:
     + **radius** (float): the radius of the circle.
   * Methods:
     + **\_\_init\_\_(self, radius)**: a constructor to initialize the **radius** attribute.
     + **area(self)**: override the **area** method from the **Shape** class to calculate and return the area of a circle using the formula **3.14 \* radius \* radius**.
3. Define the **Rectangle** class, which also inherits from the **Shape** class, with the following attributes and methods:
   * Attributes:
     + **length** (float): the length of the rectangle.
     + **width** (float): the width of the rectangle.
   * Methods:
     + **\_\_init\_\_(self, length, width)**: a constructor to initialize the **length** and **width** attributes.
     + **area(self)**: override the **area** method from the **Shape** class to calculate and return the area of a rectangle using the formula **length \* width**.

Create instances of both the **Circle** and **Rectangle** classes and demonstrate the use of the **area** method to calculate and display the areas of these shapes.

**Method Overriding:**

Define a base class "Shape" with a method "area." Create two derived classes, "Circle" and "Rectangle," which inherit from "Shape" and override the "area" method to calculate the area specific to each shape.

#Question 2

class Shape():

    pass

class Circle():

    def \_\_init\_\_(self, radius):

        self.radius=radius

    def area(self):

        return 3.142\* self.radius\* self.radius

class Rectangle():

    def \_\_init\_\_(self, l, w):

        self.l=int(l)

        self.w=int(w)

    def area(self):

        return self.l\* self.w

circle = Circle(4)

rectangle= Rectangle(4,5)

print(f"Area of circle: {circle.area()}")

print(f"Area of rectangle: {rectangle.area()}")

In this example:

1. The base class **Shape** has a method **area()** with a **pass** statement, indicating that it's a placeholder method to be overridden by derived classes.
2. The **Circle** class inherits from **Shape** and has an additional attribute **radius**. It overrides the **area()** method to calculate the area of a circle using the formula πr², where **r** is the radius.
3. The **Rectangle** class also inherits from **Shape** and has attributes for length and width. It overrides the **area()** method to calculate the area of a rectangle using the formula length × width.
4. Instances of the **Circle** and **Rectangle** classes are created, and the **area()** method is called to calculate and display the areas of the shapes. The overridden **area()** method in each derived class is used to perform the specific area calculations for circles and rectangles.

#Q3  overloading and overridding:

class Shape():

    def area():

        return 0

class Circle(Shape):

    def \_\_init\_\_(self,radius):

        self.radius=float(radius)

    def area(self):

        Shape.area()

        return 3.142\*self.radius\*self.radius

class Rectangle(Shape):

    def \_\_init\_\_(self,l,w):

        self.l=float(l)

        self.w=float(w)

    def area(self):

        Shape.area()

        return self.l\*self.w

circle=Circle(8.1425056)

rec=Rectangle(5.35698789,9.17849613)

print(f"Area of circle is {circle.area()}")

print(f"Area of rectangle is {rec.area()}")



1. Abstraction

**Problem Statement:**

You are tasked with creating a basic library management system. In this system, you'll implement an abstract class for library items and two concrete subclasses, **Book** and **DVD**. This problem will help you understand how to use abstraction to define a common interface and share certain behaviors among related classes.

1. Define an abstract class **LibraryItem** with the following methods:
   * **\_\_init\_\_(self, title, item\_id)**: a constructor to initialize the title and item ID.
   * **check\_out(self)**: an abstract method that represents the action of checking out an item.
   * **return\_item(self)**: an abstract method that represents the action of returning an item.
   * **display\_details(self)**: an abstract method to display item details.
2. Define a concrete class **Book**, which inherits from **LibraryItem**, with the following additional attributes and methods:
   * Attributes:
     + **author** (string): the author of the book.
     + **genre** (string): the genre of the book.
   * Methods:
     + **check\_out(self)**: override the **check\_out** method to indicate that a book has been checked out.
     + **return\_item(self)**: override the **return\_item** method to indicate that a book has been returned.
     + **display\_details(self)**: override the **display\_details** method to display the book's title, author, and genre.
3. Define a concrete class **DVD**, which also inherits from **LibraryItem**, with the following additional attributes and methods:
   * Attributes:
     + **director** (string): the director of the DVD.
     + **duration** (int): the duration of the DVD in minutes.
   * Methods:
     + **check\_out(self)**: override the **check\_out** method to indicate that a DVD has been checked out.
     + **return\_item(self)**: override the **return\_item** method to indicate that a DVD has been returned.
     + **display\_details(self)**: override the **display\_details** method to display the DVD's title, director, and duration.

Create instances of both **Book** and **DVD**, demonstrate the use of their methods, and display the details of items that have been checked out and returned.

