TASK 1:

Imagine you are building an application for managing employee records within a company. Implement a class in Python that encapsulates employee details such as name, ID, and salary. The class should have:

* Private variables to store the employee's details.
* Public getters and setters to access and update these details, ensuring that the salary cannot be set to a negative value.
* Provide a brief explanation of your code and how encapsulation is implemented to protect the data.
* class Employee:
* def \_\_init\_\_(self, name, emp\_id, salary):
* self.\_\_name = name
* self.\_\_emp\_id = emp\_id
* self.\_\_salary = salary
* # Getter for name
* def get\_name(self):
* return self.\_\_name
* # Setter for name
* def set\_name(self, name):
* self.\_\_name = name
* # Getter for emp\_id
* def get\_emp\_id(self):
* return self.\_\_emp\_id
* # Setter for emp\_id
* def set\_emp\_id(self, emp\_id):
* self.\_\_emp\_id = emp\_id
* # Getter for salary
* def get\_salary(self):
* return self.\_\_salary
* # Setter for salary with validation
* def set\_salary(self, salary):
* if salary >= 0:
* self.\_\_salary = salary
* else:
* print("Salary cannot be negative.")
* # Example usage
* employee = Employee("John Doe", 12345, 50000)
* # Accessing and updating employee details
* print(employee.get\_name())  # Output: John Doe
* employee.set\_name("Jane Doe")
* print(employee.get\_name())  # Output: Jane Doe
* print(employee.get\_salary())  # Output: 50000
* employee.set\_salary(60000)
* print(employee.get\_salary())  # Output: 60000
* employee.set\_salary(-1000)  # Output: Salary cannot be negative.

Explanation:

 **Private Variables**:

* The variables \_\_name, \_\_emp\_id, and \_\_salary are private. In Python, private variables are denoted by prefixing the variable name with two underscores (\_\_). This makes these variables inaccessible from outside the class directly.

 **Public Getters and Setters**:

* We use the @property decorator to define getter methods. For instance, @property def name(self) allows us to retrieve the value of the private variable \_\_name.
* We use the @name.setter decorator to define the setter method. This method allows us to set a new value for the \_\_name variable.
* Similarly, we define getters and setters for emp\_id and salary.

 **Validation in Setter**:

* For the salary setter, we include a validation check to ensure that the salary cannot be set to a negative value. If an attempt is made to set a negative salary, a ValueError is raised.

 **Encapsulation**:

* Encapsulation is implemented by using private variables to hide the internal state of the object. Access to these private variables is controlled through public getter and setter methods. This design allows us to protect the internal state of the Employee class and control how it is accessed and modified.

TASK2:

**Implementing Inheritance and Polymorphism**

You are tasked with creating a simple simulation for a vehicle management system where different types of vehicles (e.g., Cars, Trucks) are managed. Implement the following in Python:

* A base class Vehicle with common attributes like make, model, and a method start\_engine.
* Derived classes Car and Truck that override the start\_engine method.
* Demonstrate polymorphism by creating a function that takes any vehicle object and calls its start\_engine method.
* Provide a brief explanation of your code and how inheritance and polymorphism are implemented to handle different types of vehicles.

# Base class Vehicle

class Vehicle:

    def \_\_init\_\_(self, make, model):

        self.make = make

        self.model = model

    def start\_engine(self):

        print(f"The engine of the {self.make} {self.model} is starting.")

# Derived class Car

class Car(Vehicle):

    def start\_engine(self):

        print(f"The car {self.make} {self.model} is starting its engine with a roar!")

# Derived class Truck

class Truck(Vehicle):

    def start\_engine(self):

        print(f"The truck {self.make} {self.model} is starting its engine with a deep rumble!")

# Function to demonstrate polymorphism

def start\_vehicle\_engine(vehicle):

    vehicle.start\_engine()

# Creating instances of Car and Truck

car = Car("Toyota", "Corolla")

truck = Truck("Ford", "F-150")

# Demonstrating polymorphism

start\_vehicle\_engine(car)

start\_vehicle\_engine(truck)

# Base class Vehicle

class Vehicle:

    def \_\_init\_\_(self, make, model):

        self.make = make

        self.model = model

    def start\_engine(self):

        print(f"The engine of the {self.make} {self.model} is starting.")

# Derived class Car

class Car(Vehicle):

    def start\_engine(self):

        print(f"The car {self.make} {self.model} is starting its engine with a roar!")

# Derived class Truck

class Truck(Vehicle):

    def start\_engine(self):

        print(f"The truck {self.make} {self.model} is starting its engine with a deep rumble!")

# Function to demonstrate polymorphism

def start\_vehicle\_engine(vehicle):

    vehicle.start\_engine()

# Creating instances of Car and Truck

car = Car("Toyota", "Corolla")

truck = Truck("Ford", "F-150")

# Demonstrating polymorphism

start\_vehicle\_engine(car)

start\_vehicle\_engine(truck)

**Explanation:**

**Explanation:**

1. **Base Class**Vehicle: This class has common attributes make and model, and a method start\_engine that prints a generic message.
2. **Derived Classes**Car**and**Truck: These classes inherit from Vehicle and override the start\_engine method to provide specific messages for cars and trucks.
3. **Polymorphism**: The function start\_vehicle\_engine takes any Vehicle object and calls its start\_engine method. This demonstrates polymorphism, as the same function can handle different types of vehicles and call the appropriate method based on the object’s class.