**Section A – Short Answer Questions (2 marks each)**

1. Define **Object-Oriented Programming**. What are its main principles?

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects", which contain data and methods.  
**Main principles:**

* **Encapsulation**: Binding data and methods into a single unit.
* **Abstraction**: Hiding complex implementation details.
* **Inheritance**: Reusing code by inheriting features from existing classes.
* **Polymorphism**: Same interface behaves differently in different contexts.

1. Differentiate between **class** and **object** with an example.

 **Class**: Blueprint or template for creating objects.

 **Object**: Instance of a class.  
**Example**:

class Dog:

def bark(self):

print("Woof!")

d = Dog() # d is an object of class Dog

1. What are the three types of **access specifiers** in Python? Explain each briefly.

Public (name): Accessible from anywhere.

Protected (\_name): Meant for internal use; accessible within class and subclasses.

Private (\_\_name): Strictly for internal use within the class.

1. What is a **constructor** in Python? What is its special method name?

A constructor is used to initialize the object’s attributes at the time of object creation.  
**Special method name**: \_\_init\_\_().

1. What is the purpose of the super() function in inheritance?

It allows access to methods from the parent class in the child class.  
Useful for extending the functionality of inherited methods.

1. Explain the difference between **method overloading** and **method overriding**.

Overloading: Same method name, different parameters (not directly supported in Python).

Overriding: Redefining a parent method in a child class with the same signature.

1. What is **polymorphism** in OOP? Give a simple example.

Polymorphism means performing a single action in different ways.  
**Example**:

class Bird:

def sound(self):

print("Chirp")

class Dog:

def sound(self):

print("Bark")

for animal in (Bird(), Dog()):

animal.sound()

1. Can you call a **private method** outside its class in Python? If yes, how?

Yes, using name mangling:

obj.\_ClassName\_\_methodName()

**Section B – Coding-Based Questions (5 marks each)**

Write a Python class Student with attributes name and marks. Include a constructor to initialize the attributes and a method to display student details.

Expected Output:

Name: Alice

Marks: 90

class Student:

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

self.name = name

self.marks = marks

def display(self):

print("Name:", self.name)

print("Marks:", self.marks)

s = Student("Alice", 90)

s.display()

Create a class Vehicle with a method start(). Inherit a class Car from it and override the start() method. Call both parent and child methods using an object of Car.

class Vehicle:

def start(self):

print("Vehicle started")

class Car(Vehicle):

def start(self):

super().start()

print("Car started")

c = Car()

c.start()

Demonstrate the use of **public**, **protected**, and **private** variables in a class using appropriate naming conventions and access.

class AccessDemo:

def \_\_init\_\_(self):

self.public\_var = "Public"

self.\_protected\_var = "Protected"

self.\_\_private\_var = "Private"

def display(self):

print(self.public\_var)

print(self.\_protected\_var)

print(self.\_\_private\_var)

obj = AccessDemo()

obj.display()

print(obj.public\_var) # Accessible

print(obj.\_protected\_var) # Accessible but not recommended

print(obj.\_AccessDemo\_\_private\_var) # Accessing private using name mangling

Write a class hierarchy where Animal is the parent class and Dog and Cat are derived classes. Implement a method make\_sound() in each class to demonstrate **polymorphism**.

class Animal:

def make\_sound(self):

print("Animal sound")

class Dog(Animal):

def make\_sound(self):

print("Bark")

class Cat(Animal):

def make\_sound(self):

print("Meow")

for pet in (Dog(), Cat()):

pet.make\_sound()

**Section C – Application-Based Questions (8 marks each)**

Create a class BankAccount with:

* private variable \_\_balance
* methods to deposit, withdraw, and display balance
* prevent withdrawal if amount exceeds balance

Demonstrate the functionality by creating an object and calling methods.

class BankAccount:

def \_\_init\_\_(self, balance=0):

self.\_\_balance = balance

def deposit(self, amount):

self.\_\_balance += amount

def withdraw(self, amount):

if amount > self.\_\_balance:

print("Insufficient funds")

else:

self.\_\_balance -= amount

def display\_balance(self):

print("Current Balance:", self.\_\_balance)

acc = BankAccount(1000)

acc.deposit(500)

acc.withdraw(200)

acc.withdraw(2000)

acc.display\_balance()

Write a program to create a class Employee with attributes name, id, and salary. Use inheritance to create a subclass Manager that adds an additional attribute department. Use a constructor to initialize all attributes. Override a method display() to show complete info.

class Employee:

def \_\_init\_\_(self, name, emp\_id, salary):

self.name = name

self.emp\_id = emp\_id

self.salary = salary

def display(self):

print(f"Name: {self.name}, ID: {self.emp\_id}, Salary: {self.salary}")

class Manager(Employee):

def \_\_init\_\_(self, name, emp\_id, salary, department):

super().\_\_init\_\_(name, emp\_id, salary)

self.department = department

def display(self):

super().display()

print(f"Department: {self.department}")

m = Manager("John", 101, 80000, "HR")

m.display()