**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:**

1. **Encapsulation** – Bundling data and methods together.
2. **Abstraction** – Hiding complex details and showing only essentials.
3. **Inheritance** – One class can inherit features from another.
4. **Polymorphism** – One interface, multiple implementations.
5. **Differentiate between class and object with an example.**

**Class** is a blueprint or template for creating objects.

**Object** is an instance of a class that holds actual data.

class Car:

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

self.brand = brand

my\_car = Car("Toyota")

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

**Public**

Accessible from anywhere in the program.

* Example: self.name

**Protected**

Meant to be accessed within the class and its subclasses.

* Example: self.\_age

**Private**

Accessible only within the class .

* Example: self.\_\_salary

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

A constructor is a special method used to initialize objects when a class is instantiated. It sets initial values for object properties.

* The special method name for a constructor is \_\_init\_\_().
* Example:

class Person:

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

self.name = name

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

The super() function is used to call methods from the parent (super) class in a child class. It helps in accessing the parent’s constructor or methods without explicitly naming the parent class.

Example:

class Parent:

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

print("Parent constructor")

class Child(Parent):

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

super().\_\_init\_\_()

print("Child constructor")

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

**Method Overloading**

* Same method name with different number or types of parameters.
* Not directly supported in Python, but can be mimicked using default arguments or \*args.

**Method Overriding**

* A child class provides a **new implementation** of a method defined in the parent class.
* Supported in Python through inheritance.

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

Polymorphism means "many forms". It allows the same method or function to behave differently based on the object or context.

Example:

class Dog:

def sound(self):

print("Barks")

class Cat:

def sound(self):

print("Meows")

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

animal.sound()

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

Yes, you can call a private method outside its class in Python using name mangling.

Private methods in Python are defined with double underscores, like \_\_method().

Example:

class Test:

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

pass

def \_\_private\_method(self):

print("This is private")

obj = Test()

obj.\_Test\_\_private\_method() # Accessing private method using name mangling

**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(f"Name: {self.name}")

print(f"Marks: {self.marks}")

s1 = Student("Alice", 90)

s1.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.**

Solution:

class Vehicle:

def start(self):

print("Vehicle is starting")

class Car(Vehicle):

def start(self):

print("Car is starting")

super().start()

my\_car = Car()

my\_car.start()

Output:

Car is starting

Vehicle is starting

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

class Demo:

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

self.public\_var = "I am Public"

self.\_protected\_var = "I am Protected"

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

def display(self):

print(self.public\_var)

print(self.\_protected\_var)

print(self.\_\_private\_var)

obj = Demo()

obj.display()

print("\nAccessing variables outside the class:")

print(obj.public\_var)

print(obj.\_protected\_var)

print(obj.\_Demo\_\_private\_var)

output:

I am Public

I am Protected

I am Private

Accessing variables outside the class:

I am Public

I am Protected

I am Private

**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 makes a sound")

class Dog(Animal):

def make\_sound(self):

print("Dog barks")

class Cat(Animal):

def make\_sound(self):

print("Cat meows")

animals = [Dog(), Cat(), Animal()]

for animal in animals:

animal.make\_sound()

output:

Dog barks

**Cat** meows

Animal makes a 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, initial\_balance=0):

self.\_\_balance = initial\_balance

def deposit(self, amount):

if amount > 0:

self.\_\_balance += amount

print(f"Deposited: {amount}")

else:

print("Invalid deposit amount.")

def withdraw(self, amount):

if amount <= self.\_\_balance:

self.\_\_balance -= amount

print(f"Withdrawn: {amount}")

else:

print("Insufficient balance!")

def display\_balance(self):

print(f"Current Balance: {self.\_\_balance}")

account = BankAccount(1000)

account.display\_balance()

account.deposit(500)

account.withdraw(300)

account.withdraw(1500)

account.display\_balance()

Output:

Current Balance: 1000

Deposited: 500

Withdrawn: 300

Insufficient balance!

Current Balance: 1200

**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.**

**Solution:**

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}")

print(f"ID: {self.emp\_id}")

print(f"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}")

emp1 = Manager("John Doe", 101, 75000, "HR")

emp1.display()

output:

Name: John Doe

ID: 101

Salary: 75000

Department: HR