January 2, 2025

Assignment of OOPS (Object oriented programming)

Theory questions

2.0.1) What is Object-Oriented Programming (OOP)

A programming paradigm based on the concept of "objects," which contain data in the form of fields (attributes) and code in the form of methods (functions). It emphasizes reusable code and modularity.

2.0.2) What is a class in OOP?

A blueprint or template for creating objects. It defines the attributes and methods that the objects created from the class will have.

2.0.3) What is an object in OOP?

An instance of a class. It is a specific realization of the class with actual values assigned to the class's attributes.

2.0.4) What is the difference between abstraction and encapsulation?

Abstraction vs. Encapsulation

(i)Abstraction: Hiding implementation details and showing only essential features to the user.

(ii)Encapsulation: Bundling of data and methods that operate on that data into a single unit (class) and restricting access to some components using access specifiers (e.g., private).

2.0.5) What are dunder methods in Python?

Double-underscore methods like init, str, repr, etc., are special methods that provide functionality for built-in operations. They're also known as "magic methods."

2.0.6) Explain the concept of inheritance in OOP.

A mechanism by which one class (child) can inherit the attributes and methods of another class (parent), promoting code reuse.

2.0.7) What is polymorphism in OOP?

The ability to present the same interface for different underlying data types or classes. For example, a single method name may work differently depending on the object that calls it.

2.0.8) How is encapsulation achieved in Python?

Achieved using private (___attr) or protected (_attr) attributes, restricting direct access and providing controlled access via getter and setter methods.

2.0.9) What is a constructor in Python?

A special method (init) automatically called when an object is created, used to initialize attributes of the class.

2.0.10) What are class and static methods in Python?

(i)Class Methods: Defined using @classmethod. They take cls as the first parameter and operate on the class rather than an instance.

(ii)Static Methods: Defined using @staticmethod. They do not take self or cls and are used for utility functions related to the class.

2.0.11) What is method overloading in Python?

Python does not natively support method overloading, but you can achieve similar behavior using default arguments or variable-length arguments.

2.0.12) What is method overriding in OOP?

A child class provides a specific implementation for a method that is already defined in its parent class.

2.0.13) What is a property decorator in Python?

The @property decorator allows you to define a method that can be accessed like an attribute, useful for implementing getter, setter, and deleter functionalities.

2.0.14) Why is polymorphism important in OOP?

Promotes flexibility and maintainability by allowing the same interface to be used for different types, simplifying code and enabling easier extensions.

2.0.15) What is an abstract class in Python?

A class that cannot be instantiated and is used as a blueprint for other classes. It often contains abstract methods, defined using the @abstractmethod decorator.

2.0.16) What are the advantages of OOP?

Advantages of OOP

(i) Modularity

(ii) Reusability

(iii) Scalability

(iv) Maintainability

(v) Improved collaboration and debugging

2.0.17) What is multiple inheritance in Python?

A feature that allows a class to inherit attributes and methods from more than one parent class.

2.0.18) What is the difference between a class variable and an instance variable?

(i) Class Variable: Shared across all instances of the class.

(ii)Instance Variable: Unique to each object/instance of the class.

2.0.19) Explain the purpose of ''str' and 'repr' ' methods in Python.

(i) str: Provides a readable, user-friendly string representation of an object.

(ii)repr: Provides an unambiguous string representation, often used for debugging.

2.0.20) What is the significance of the 'super()' function in Python?

Used to call a method of the parent class, commonly used in the context of inheritance to avoid code duplication.

2.0.21) What is the significance of the del method in Python?

A destructor method automatically called when an object is deleted, used for cleanup tasks like closing files or releasing resources.

2.0.22) What is the difference between @staticmethod and @classmethod in Python?

@staticmethod: No access to the instance (self) or class (cls).

@classmethod: Can access and modify class-level attributes via cls.

2.0.23) How does polymorphism work in Python with inheritance?

Allows methods in child classes to override methods in parent classes, enabling objects of different classes to be treated uniformly.

2.0.24) What is method chaining in Python OOP?

The practice of calling multiple methods on the same object sequentially in a single statement, made possible by returning self.

2.0.25) What is the purpose of the call method in Python?

Allows an instance of a class to be called like a function. This is useful for creating callable objects.

PRACTICAL Questions

1. Create a parent class Animal with a method speak() that prints a generic message. Create a child class Dog that overrides the speak() method to print "Bark!".

[76]:
```
class Animal:
    def speak(self):
        print("Animal makes a sound.")

class Dog(Animal):
    def speak(self):
        print("Bark!")

animal = Animal()
dog = Dog()

animal.speak()
dog.speak()
```

```
Animal makes a sound.
Bark!
```

2. Write a program to create an abstract class Shape with a method area(). Derive classes Circle and Rectangle from it and implement the area() method in both.

[78]:
```
from abc import ABC, abstractmethod
class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius
```

```
    def area(self):
        return 3.14 * (self.radius ** 2)

class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

circle = Circle(5)
rectangle = Rectangle(4, 6)

print(f"Circle Area: {circle.area()}")
print(f"Rectangle Area: {rectangle.area()}")
```

```
Circle Area: 78.5
Rectangle Area: 24
```

3. Implement a multi-level inheritance scenario where a class Vehicle has an attribute type. Derive a class Car and further derive a class ElectricCar that adds a battery attribute.

[72]:
```
class Vehicle:
    def __init__(self, vehicle_type):
        self.vehicle_type = vehicle_type
    def get_type(self):
        return f"Vehicle Type: {self.vehicle_type}"

class Car(Vehicle):
    def __init__(self, vehicle_type, brand, model):
        super().__init__(vehicle_type)
        self.brand = brand
        self.model = model
    def get_details(self):
        return f"{self.get_type()}, Brand: {self.brand}, Model: {self.model}"

class ElectricCar(Car):
    def __init__(self, vehicle_type, brand, model, battery_capacity):
        super().__init__(vehicle_type, brand, model)
        self.battery_capacity = battery_capacity
    def get_battery_info(self):
        return f"Battery Capacity: {self.battery_capacity} kWh"

electric_car = ElectricCar("Electric", "Tesla", "Model S", 100)
```

```
print(electric_car.get_details())
print(electric_car.get_battery_info())
```

```
Vehicle Type: Electric, Brand: Tesla, Model: Model S
Battery Capacity: 100 kWh
```

4. Implement a multi-level inheritance scenario where a class Vehicle has an attribute type. Derive a class Car and further derive a class ElectricCar that adds a battery attribute.

[70]:
```
class Vehicle:
    def __init__(self, vehicle_type):
        self.vehicle_type = vehicle_type
    def get_type(self):
        return f"Vehicle Type: {self.vehicle_type}"

class Car(Vehicle):
    def __init__(self, vehicle_type, brand, model):
        super().__init__(vehicle_type)
        self.brand = brand
        self.model = model
    def get_details(self):
        return f"{self.get_type()}, Brand: {self.brand}, Model: {self.model}"

class ElectricCar(Car):
    def __init__(self, vehicle_type, brand, model, battery_capacity):
        super().__init__(vehicle_type, brand, model)
        self.battery_capacity = battery_capacity
    def get_battery_info(self):
        return f"Battery Capacity: {self.battery_capacity} kWh"

electric_car = ElectricCar("Electric", "Tesla", "Model S", 100)

print(electric_car.get_details())
print(electric_car.get_battery_info())
```

```
Vehicle Type: Electric, Brand: Tesla, Model: Model S
Battery Capacity: 100 kWh
```

5. Write a program to demonstrate encapsulation by creating a class BankAccount with private attributes balance and methods to deposit, withdraw, and check balance.

[68]:
```
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("Deposit amount must be positive.")

    def withdraw(self, amount):
        if amount > 0 and amount <= self.__balance:
            self.__balance -= amount
            print(f"Withdrew: ${amount}")
        else:
            print("Invalid withdrawal amount or insufficient balance.")

    def check_balance(self):
        return f"Current Balance: ${self.__balance}"

account = BankAccount(1000)
account.deposit(500)
account.withdraw(400)
print(account.check_balance())
```

```
Deposited: $500
Withdrew: $400
Current Balance: $1100
```

6. Demonstrate runtime polymorphism using a method play() in a base class Instrument. Derive classes Guitar and Piano that implement their own version of play().

[54]:
```
class Instrument:
    def play(self):
        pass

class Guitar(Instrument):
    def play(self):
        print("Playing the guitar: Strum strum!")

class Piano(Instrument):
    def play(self):
        print("Playing the piano: Plink plink!")

def play_instrument(instrument):
    instrument.play()

guitar = Guitar()
```

```
piano = Piano()

play_instrument(guitar)
play_instrument(piano)
```

```
Playing the guitar: Strum strum!
Playing the piano: Plink plink!
```

4.0.7 7. Create a class MathOperations with a class method add_numbers() to add two numbers and a static method subtract_numbers() to subtract two numbers.

[52]:
```
class MathOperations:
    @classmethod
    def add_numbers(cls, num1, num2):
        return num1 + num2

    @staticmethod
    def subtract_numbers(num1, num2):
        return num1 - num2

sum_result = MathOperations.add_numbers(10, 5)
print(f"Sum: {sum_result}")

difference_result = MathOperations.subtract_numbers(10, 5)
print(f"Difference: {difference_result}")
```

```
Sum: 15
Difference: 5
```

8. Implement a class Person with a class method to count the total number of persons created.

[21]:
```
class Person:
    count = 0
    def __init__(self, name):
        self.name = name
        Person.count += 1
    @classmethod
    def total_persons(cls):
        return cls.count

p1 = Person("Alice")
p2 = Person("Bob")
p3 = Person("Charlie")

print("Total Persons Created:", Person.total_persons())
```

```
Total Persons Created: 3
```

9. Write a class Fraction with attributes numerator and denominator. Override the str method to display the fraction as "numerator/denominator".

[23]:
```
class Fraction:
    def __init__(self, numerator, denominator):
        if denominator == 0:
            raise ValueError("Denominator cannot be zero.")
        self.numerator = numerator
        self.denominator = denominator

    def __str__(self):
        return f"{self.numerator}/{self.denominator}"

fraction1 = Fraction(3, 4)
print(fraction1)

fraction2 = Fraction(7, 8)
print(fraction2)
```

```
3/4
7/8
```

10. Demonstrate operator overloading by creating a class Vector and overriding the add method to add two vectors.

[25]:
```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        if not isinstance(other, Vector):
            raise TypeError("Operands must be instances of Vector.")
        return Vector(self.x + other.x, self.y + other.y)

    def __str__(self):
        return f"({self.x}, {self.y})"


v1 = Vector(3, 4)
v2 = Vector(1, 2)

v3 = v1 + v2
print(v3)
```

```
(4, 6)
```

11. Create a class Person with attributes name and age. Add a method greet() that prints "Hello, my name is {name} and I am {age} years old."

[33]:
```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def greet(self):
        print(f"Hello, my name is {self.name} and I am {self.age} years old.")

# Example Usage
person = Person("Ravi kumar yadav", 18)
person.greet()
```

```
Hello, my name is Ravi kumar yadav and I am 18 years old.
```

12. Implement a class Student with attributes name and grades. Create a method average_grade() to compute the average of the grades.

[31]:
```
class Student:
    def __init__(self, name, grades):
        self.name = name
        self.grades = grades

    def average_grade(self):
        if not self.grades:
            return 0
        return sum(self.grades) / len(self.grades)

# Example Usage
student = Student("Ravi", [90, 85, 88, 92, 87])
print(f"{student.name}'s average grade is: {student.average_grade()}")
```

```
Ravi's average grade is: 88.4
```

13. Create a class Rectangle with methods set_dimensions() to set the dimensions and area() to calculate the area.

[36]:
```
class Rectangle:
    def __init__(self):
        self.length = 0
        self.width = 0

    def set_dimensions(self, length, width):
        self.length = length
        self.width = width
```

```
    def area(self):
        return self.length * self.width

# Example Usage
rectangle = Rectangle()
rectangle.set_dimensions(5, 3)
print(f"Area of the rectangle is: {rectangle.area()}")
```

```
Area of the rectangle is: 15
```

14. Create a class Employee with a method calculate_salary() that computes the salary based on hours worked and hourly rate. Create a derived class Manager that adds a bonus to the salary.

[38]:
```
class Employee:
    def __init__(self, name, hours_worked, hourly_rate):
        self.name = name
        self.hours_worked = hours_worked
        self.hourly_rate = hourly_rate

    def calculate_salary(self):
        return self.hours_worked * self.hourly_rate

class Manager(Employee):
    def __init__(self, name, hours_worked, hourly_rate, bonus):
        super().__init__(name, hours_worked, hourly_rate)
        self.bonus = bonus
    def calculate_salary(self):
        return super().calculate_salary() + self.bonus

employee = Employee("Ravi", 40, 15)
print(f"Employee Salary: ${employee.calculate_salary()}")

manager = Manager("Ajay", 40, 20, 500)
print(f"Manager Salary: ${manager.calculate_salary()}")
```

```
Employee Salary: $600
Manager Salary: $1300
```

15. Create a class Product with attributes name, price, and quantity. Implement a method total_price() that calculates the total price of the product.

[40]:
```
class Product:
    def __init__(self, name, price, quantity):
        self.name = name
        self.price = price
        self.quantity = quantity
```

```
    def total_price(self):
        return self.price * self.quantity

product = Product("Laptop", 1000, 3)
print(f"Total price of {product.name}: ${product.total_price()}")
```

```
Total price of Laptop: $3000
```

16. Create a class Animal with an abstract method sound(). Create two derived classes Cow and Sheep that implement the sound() method.

[46]:
```
from abc import ABC, abstractmethod
class Animal(ABC):
    @abstractmethod
    def sound(self):
        pass

class Cow(Animal):
    def sound(self):
        print("Moo!")

class Sheep(Animal):
    def sound(self):
        print("Baa!")

cow = Cow()
cow.sound()

sheep = Sheep()
sheep.sound()
```

```
Moo!
Baa!
```

17. Create a class Book with attributes title, author, and year_published. Add a method get_book_info() that returns a formatted string with the book's details.

[48]:
```
class Book:
    def __init__(self, title, author, year_published):
        self.title = title
        self.author = author
        self.year_published = year_published

    def get_book_info(self):
```

```
        return f"Title: {self.title}, Author: {self.author}, Year Published:␣
 ↪{self.year_published}"

book = Book("The Great Gatsby", "F. Scott Fitzgerald", 1925)
print(book.get_book_info())
```

```
Title: The Great Gatsby, Author: F. Scott Fitzgerald, Year Published: 1925
```

18. Create a class House with attributes address and price. Create a derived class Mansion that adds an attribute number_of_rooms.

[50]:
```
class House:
    def __init__(self, address, price):
        self.address = address
        self.price = price

    def get_details(self):
        return f"Address: {self.address}, Price: ${self.price}"

class Mansion(House):
    def __init__(self, address, price, number_of_rooms):
        super().__init__(address, price)
        self.number_of_rooms = number_of_rooms

    def get_details(self):
        base_details = super().get_details()
        return f"{base_details}, Number of Rooms: {self.number_of_rooms}"

house = House("123 Main St", 250000)
print(house.get_details())

mansion = Mansion("456 Luxury Blvd", 5000000, 10)
print(mansion.get_details())
```

```
Address: 123 Main St, Price: $250000
Address: 456 Luxury Blvd, Price: $5000000, Number of Rooms: 10
```

[ ]:
```

```