Assignment of OOPS (Moduel 6)

Theory Questions

Q1. What is Object-Oriented Programming (OOP)?

Ans-> Object-Oriented Programming (OOP) is a programming paradigm that organizes software design around data (objects) and their behaviors (methods), rather than functions and logic, promoting modularity, reusability, and maintainability. OOP is a way of structuring software by grouping data and the procedures that operate on that data into "objects".

Q2. What is a class in OOP?

Ans-> In Python's Object-Oriented Programming (OOP), a class is a blueprint or template for creating objects, defining their attributes (data) and methods (functions).

- Here's a more detailed explanation:
- Blueprint for Objects: Think of a class as a recipe or a template for creating something. It outlines what that thing (the object) will be like, including its characteristics and actions.
- Attributes (Data): Attributes are variables that store data associated with an object of that class. For example, a Dog class might have attributes like name, breed, and age.
- Methods (Functions): Methods are functions that define the behavior or actions that an object of that class can perform. For example, a Dog class might have methods like bark(), eat(), or play().
- Creating Objects (Instances): You create an object (also called an instance) by calling the class name as if it were a function. The object will then have all the attributes and methods defined in the class.

Q3. What is an object in OOP?

Ans-> In object-oriented programming (OOP), an object is a fundamental unit that encapsulates data (attributes) and behaviors (methods) representing a real-world entity or concept, and is an instance of a class.

Q4. What is the difference between abstraction and encapsulation?

Ans-> In Python, abstraction focuses on hiding complex implementation details and presenting a simplified interface, while encapsulation bundles data and methods within a single unit (like a class) and controls access to them.

Q5. What are dunder methods in Python?

Ans-> In Python, "dunder" methods (short for "double underscore" methods) are special methods, also known as "magic methods", that have double underscores at the beginning and end of their names (e.g., **init**, **str**) and are used to define specific behaviors for built-in operations or functionalities within classes.

Q6. Explain the concept of inheritence in OOP?

Ans-> In Object-Oriented Programming (OOP), inheritance is a mechanism where a new class (subclass or child class) inherits properties and behaviors from an existing class (superclass or parent class), promoting code reuse and establishing hierarchical relationships between classes.

Q7. What is Polymorphism in OOP?

Ans-> In object-oriented programming (OOP), polymorphism means "many forms" and allows objects of different classes to be treated as objects of a common type, enabling them to respond differently to the same method call.

Q8. How is encapsulation achieved in Python?

Ans-> In Python, encapsulation, which bundles data and methods within a class and restricts access to some members, is achieved through naming conventions: using a single underscore _ for protected members (accessible within the class and its subclasses) and a double underscore __ for private members (accessible only within the class).

Q9. What is a constructor in Python?

Ans-> In Python, a constructor, also known as the **init** method, is a special method within a class that's automatically called when a new object (instance) of that class is created, used to initialize the object's attributes.

Q10. What are class and static methods in Python?

Ans-> In Python, class methods are bound to the class itself and receive the class (cls) as their first argument, while static methods are associated with the class but don't receive any automatic arguments.

- Here's a more detailed explanation:

1. Class Methods:

- Binding: Class methods are bound to the class, not to instances of the class.
- First Argument: They receive the class (cls) as their first argument, allowing them to access and modify class-level attributes.

- Usage: Class methods are often used for creating alternative constructors or performing actions that affect the class as a whole.

2. Static Methods:

- Binding: Static methods are associated with a class but are not bound to either the class or any instance of the class.
- No Automatic Arguments: They don't receive any automatic arguments (like self for instance methods or cls for class methods).
- Usage: Static methods are used for utility functions that are related to the class but don't require access to instance-specific or class-specific attributes.

Q11. What is method overloading in Python?

Ans-> In Python, while it's not a direct feature like in some other languages, "method overloading" is achieved through the use of default arguments, variable-length arguments (*args, *kwargs), or conditional logic within a single function definition, allowing a function to handle different inputs and behaviors.

Q12. What is method overriding in OOP?

Ans-> In Object-Oriented Programming (OOP), method overriding allows a subclass to provide a specific implementation of a method that is already defined in its superclass, enabling dynamic method dispatch and polymorphism.

Q13. What is a property decorator in Python?

Ans-> The Basics of @property

The decorator is a built-in Python decorator that allows you to turn class methods into properties in a way that's both elegant and user-friendly.

Q14. Why is polymorphism important in OOP?

Ans-> Polymorphism is crucial in Object-Oriented Programming (OOP) because it enables code reusability, flexibility, and extensibility by allowing different objects to respond to the same method call in their own unique ways, promoting a more organized and maintainable codebase.

Q15. What is an abstract class in Python?

Ans-> In Python, an abstract class is a blueprint for other classes that cannot be instantiated directly; it's meant to be subclassed and contains abstract methods (methods without implementation) that derived classes must implement.

Q16. What are the advantages of OOP?

Ans-> Object-Oriented Programming (OOP) in Python offers advantages like code reusability, modularity, and maintainability, making large projects easier to manage and debug.

- Here's a breakdown of the key advantages:
- Code Reusability: OOP promotes code reusability through inheritance, allowing developers to create new classes that inherit functionality from existing ones, reducing redundancy and saving time.
- Modularity: OOP structures code into modules (classes) that can be developed and maintained independently, making it easier to manage complex projects.
- Maintainability: The modular nature of OOP makes it easier to understand, debug, and update code, as changes in one part of the code are less likely to affect other parts.
- Flexibility and Extensibility: OOP's features like polymorphism and inheritance allow for flexible and extensible code, enabling developers to easily adapt to changing requirements and add new features without modifying existing code.
- Problem-Solving: OOP provides a structured approach to problem-solving by breaking down complex problems into smaller, manageable units (classes and objects).
- Data Security: Encapsulation, a core OOP principle, helps protect data by controlling access to it, improving data security.
- Enhanced Productivity: OOP practices can lead to increased productivity by streamlining data management, reducing redundancy, and facilitating code reuse.

Q17. What is the difference between a class variable and an instance variable?

Ans-> In Python, class variables are shared among all instances of a class, while instance variables are unique to each instance.

- Here's a breakdown:
- Class Variables:
- Defined within the class, outside of any methods (including **init**). Shared by all instances of the class; a change to a class variable affects all instances.
- Can be accessed using the class name (e.g., MyClass.my_variable) or an instance of the class (e.g., instance.my_variable). Useful for storing data that is common to all instances, like constants or counters.
- Instance Variables: Defined within methods (typically the **init** method).
- Unique to each instance; each instance has its own copy of the variable.
- Accessed using the instance name (e.g., instance.my_variable). Useful for storing data that is specific to each instance, like attributes.

Q18. What is multiple inheritance in Python?

Ans-> In Python, multiple inheritance occurs when a class (the child or derived class) inherits attributes and methods from multiple parent or base classes, allowing it to combine functionalities from those parents.

- Here's a breakdown:
- Single Inheritance: A class inheriting from only one parent class is considered single inheritance.
- Multiple Inheritance: A class inheriting from two or more parent classes is known as multiple inheritance.
- Example: Imagine a class Multimedia inheriting from Video and Audio classes. The Multimedia class would then have access to the methods and attributes of both Video and Audio.

Q19. Explain the purpose of "**str**'and'**repr**"" methods in Python?

Ans-> In Python, **str**() provides a user-friendly, readable string representation of an object, while **repr**() offers an unambiguous, developer-focused representation for debugging and object reconstruction.

- Here's a more detailed explanation:
- **str**() (for end-users):
- This method is used to create a string representation of an object that is easy for humans to read and understand.
- It's typically called when you use the print() function or the str() function on an object.
- The output of **str**() should be informative and easy to understand for the end-user.
- **repr**() (for developers/debugging):
- This method provides a more detailed and unambiguous string representation of an object, suitable for developers and debugging.
- It's called by the repr() function or when you use backticks ( ) around an object in the Python interpreter.
- The output of **repr**() should be precise and allow you to reconstruct the object from its string representation.

Q20. What is the significance of the 'super()' functin in Python?

Ans-> In Python, super() is a built-in function that allows access to methods and properties of a parent or superclass from a child or subclass. This is useful when working with inheritance in object-oriented programming.

Q21. What is the significance of the **del** method in Python?

Ans-> In Python, the **del** method, also known as a destructor, is called by the garbage collector when an object is about to be destroyed, allowing you to perform cleanup tasks like closing files or releasing resources before the object is fully deallocated.

- Here's a more detailed explanation: Purpose: The **del** method is used for object cleanup, ensuring that any external resources held by the object are released before the object is garbage collected.
- When it's called: Python's garbage collector automatically calls **del** when it determines that an object is no longer needed and has no remaining references.

Q22. What is the difference between @staticmethod and @classmethod in Python?

Ans-> In Python, @classmethod binds a method to the class itself, receiving the class (cls) as the first argument, while @staticmethod binds a method to the class without any implicit arguments, making it suitable for utility functions within a class.

- Here's a more detailed explanation: @classmethod
- Binding: A class method is bound to the class, meaning it can access and modify class-level data (attributes) using the cls parameter.
- Usage: Class methods are often used for: Creating alternative constructors for the class.
- Accessing or modifying class-level data.
- Implementing factory methods that create instances of the class in a specific way.

  @staticmethod
- Binding: A static method is bound to the class but does not receive any implicit arguments (like self or cls).
- Usage: Static methods are often used for:
- Utility functions that are logically related to the class but don't depend on class or instance state.
- Functions that don't need to access or modify any class or instance attributes.

Q23. How does polymorphism work in Python with inheritance?

Ans-> In Python, polymorphism with inheritance is achieved through method overriding, where a child class redefines a method inherited from its parent class, allowing objects of different classes to be treated as objects of a common base class.

- Here's a breakdown:
- Inheritance: A child class (subclass) inherits attributes and methods from a parent class (superclass).
- Method Overriding: If a child class needs a specific implementation of a method that already exists in the parent class, it can redefine (override) that method in the child class.

- Polymorphism: This allows you to call the same method on different objects, and each object will execute the method's implementation that is specific to its class.

Q24. What is method chaining in Python OOP?

Ans-> In Python's object-oriented programming, method chaining allows you to call multiple methods on the same object in a single line, where each method returns the object itself, enabling a fluent and readable code style.

Q25. What is the purpose of the **call** method in Python?

Ans-> The **call** method in Python allows you to make instances of a class callable, meaning you can use them as if they were functions, by defining a **call** method within the class.

- Here's a breakdown:
- Purpose: To enable instances of a class to be called like functions.
- How it works: When you define a **call** method in a class, any instance of that class can then be called using the () operator, as if it were a function.

Practical Questions

Q1.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!'.?

```
class Animal:
    def speak(self):
        print("Generic animal sound")

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

Q2. 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?

```
# prompt: Write a program to create an abstract class shape with a method area(). Derive

from abc import ABC, abstractmethod
import math
```

```
class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

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

    def area(self):
        return math.pi * 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
```

Q3.Implement a multi-level inheritence scenario where a class Vehicle has an attrubute type. Derive a class car and further derive a class Electric car that adds a battery attribute?

```
# prompt: Implement a multi-level inheritence scenario where a class Vehicle has an attru

class Car(Vehicle):
    def __init__(self, vehicle_type, model):
        super().__init__(vehicle_type)
        self.model = model

class ElectricCar(Car):
    def __init__(self, vehicle_type, model, battery_capacity):
        super().__init__(vehicle_type, model)
        self.battery = battery_capacity
```

Q4. Demonstrate polymorphism by creating a base class bird with a method fly() create two derived classes sparrow and penguin that override the fly() method?

```
# prompt: Demonstrate polymorphism by creating a base class bird with a method fly() crea

class Bird:
    def fly(self):
        print("Generic bird flying")

class Sparrow(Bird):
    def fly(self):
        print("Sparrow is flying high")

class Penguin(Bird):
    def fly(self):
        print("Penguin can't fly")
```

Q5. Write a program to demostrate encapsulation by creating a class Bank Account with private attributes balance and methods to deposit, withdraw, and check balance?

```
# prompt: Write a program to demostrate encapsulation by creating a class Bank Account wi

class BankAccount:
    def __init__(self, initial_balance=0):
        self.__balance = initial_balance  # Private attribute

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
            print(f"Deposited ${amount}. New balance: ${self.__balance}")
        else:
            print("Invalid deposit amount.")

    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount
            print(f"Withdrew ${amount}. New balance: ${self.__balance}")
        else:
            print("Insufficient funds or invalid withdrawal amount.")

    def check_balance(self):
        print(f"Current balance: ${self.__balance}")
```

Q6. Demonstrate runtime polymorphism using a method play() in a base class Instrument. Derive classes Guitar and Piano that implement their own version of play()

```
# prompt: Demonstrate runtime polymorphism using a method play() in a base class Instrume

class Instrument:
    def play(self):
        print("Generic instrument sound")

class Guitar(Instrument):
    def play(self):
        print("Guitar is being played")

class Piano(Instrument):
    def play(self):
        print("Piano is being played")

# Example usage
instruments = [Instrument(), Guitar(), Piano()]
for instrument in instruments:
    instrument.play()
```

```
Generic instrument sound
Guitar is being played
Piano is being played
```

Q7. Create a class MathOperations with a class method add_numbers () to add two numbers and a static method subtract_numbers() to subtract two numbers.

```
# prompt: Create a class MathOperations with a class method add_numbers () to add two num

class MathOperations:
    @classmethod
    def add_numbers(cls, x, y):
        return x + y

    @staticmethod
    def subtract_numbers(x, y):
        return x - y
```

Q8. Implement a class Person with a class method to count the total number of persons created

```
# prompt: Implement a class Person with a class method to count the total number of perso

class Person:
    count = 0

    def __init__(self, name):
        self.name = name
        Person.count += 1

    @classmethod
    def get_total_persons(cls):
        return cls.count
```

Q9. Write a class Fraction with attributes numerator and denominator. Override the str method to display the fraction as "numerator/denominator".

```
# prompt: Write a class Fraction with attributes numerator and denominator. Override the

class Fraction:
    def __init__(self, numerator, denominator):
        self.numerator = numerator
        self.denominator = denominator

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

Q10. Demonstrate operator overloading by creating a class Vector and overriding the add method to add two vectors.

```
# prompt: Demonstrate operator overloading by creating a class Vector and overriding the

class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)

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

Q11. 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."

```
# prompt: Create a class Person with attributes name and age. Add a method greet() that p

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.")
greet = Person("Amol", 34)
greet.greet()
```

```
Hello my name is Amol and I am 34 years old.
```

Q12. Implement a class Student with attributes name and grades. Create a method average_grade() to compute the average of the grades.

```
# prompt: Implement a class Student with attributes name and grades. Create a method aver

class Student:
    def __init__(self, name, grades):
        self.name = name
        self.grades = grades

    def average_grade(self):
        if not self.grades:
            return 0  # Handle empty grades list
# prompt: Implement a class Student with attributes name and grades. Create a method aver

class Student:
    def __init__(self, name, grades):
        self.name = name
        self.grades = grades

    def average_grade(self):
        if not self.grades:
            return 0  # Handle empty grades list
        return sum(self.grades) / len(self.grades)
```

Q13. Create a class Rectangle with methods set_dimensions() to set the dimensions and area() to calculate the area.

```
# prompt: Create a class Rectangle with methods set_dimensions() to set the dimensions an
```

```
class Rectangle:
    def set_dimensions(self, width, height):
        self.width = width
        self.height = height

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

Q14. 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.

```
# prompt: Create a class Employee with a method calculate_salary() that computes the sala

class Employee:
    def __init__(self, hours_worked, hourly_rate):
        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, hours_worked, hourly_rate, bonus):
        super().__init__(hours_worked, hourly_rate)
        self.bonus = bonus

    def calculate_salary(self):
        return super().calculate_salary() + self.bonus
```

Q15. Create a class product with attributes name, price, and quantity. Implement a method total_price() that calculates the total price of the product.

```
# prompt: Create a class product with attributes name, price, and quantity. Implement a m

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
```

Q16. Create a class Animal with an abstract method sound(). Create two derived classes Cow and Sheep that implement the sound() method.

```
# prompt: Create a class Animal with an abstract method sound(). Create two derived class

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")
```

Q17. 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.

```
# prompt: Create a class Book with attributes title, author, and year_published. Add a me

class Book:
    def __init__(self, title, author, year_published):
        self.title = title #Fire of Wings
        self.author = author #Dr. APJ Abdul Kalam
        self.year_published = year_published #1999

    def get_book_info(self):
        return f"Title: {self.title}, Author: {self.author}, Year Published: {self.year_p
```

Q18. Create a class House with attributes address and price. Create a derived class Mansion that adds an attribute number_of_rooms.

```
# prompt: Create a class House with attributes address and price. Create a derived class

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

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

Start coding or generate with AI.