Python Oops assignment

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

OOP is a programming paradigm that organizes code into objects, which are instances of classes, combining data and methods to model real-world entities.

2. What is a class in OOP?

A class in OOP is a blueprint for creating objects. It defines attributes (data) and methods (functions) that the objects created from the class can have.

3. What is an object in OOP?

An object is an instance of a class in OOP. It represents a specific entity with the properties and behaviors defined by the class.

4. What is the difference between abstraction and encapsulation?

Abstraction hides complex implementation details and shows only the necessary features, while encapsulation bundles data and methods into a single unit (class) and restricts direct access to some components.

5. What are dunder methods in Python?

Dunder (double underscore) methods in Python are special methods with names like `__init__`, `__str__`, etc., used to define how objects behave with built-in operations (e.g., initialization, string representation).

6. Explain the concept of inheritance in OOP.

Inheritance in OOP allows a class (child) to inherit attributes and methods from another class (parent), promoting code reuse and establishing a hierarchical relationship.

7. What is polymorphism in OOP?

Polymorphism in OOP allows different classes to be treated as instances of the same class through a common interface, typically by overriding methods in subclasses

8. How is encapsulation achieved in Python?

Encapsulation in Python is achieved using access modifiers like single underscore (_) for protected and double underscore (__) for private (via name mangling) to restrict access to attributes and methods.

9. What is a constructor in Python?

A constructor in Python is the `__init__` method, which is automatically called when an object is created to initialize its attributes.

10. What are class and static methods in Python?

Class methods are defined with `@classmethod` and take `cls` as the first parameter, operating on the class itself. Static methods are defined with `@staticmethod` and don't take `self` or `cls`, behaving like regular functions within a class.

11. What is method overloading in Python?

Python does not support traditional method overloading (multiple methods with the same name but different parameters). Instead, you can achieve similar functionality using default arguments or variable-length arguments (`*args`, `**kwargs`).

12. What is method overriding in OOP?

Method overriding in OOP occurs when a subclass provides a specific implementation of a method that is already defined in its parent class, allowing customization of behavior.

13. What is a property decorator in Python?

The `@property` decorator in Python allows you to define methods that can be accessed like attributes, providing a way to control access to an object's data (getter, setter, deleter).

14. Why is polymorphism important in OOP?

Polymorphism is important in OOP because it enables flexibility and extensibility, allowing different classes to be used interchangeably through a common interface, simplifying code maintenance.

15. What is an abstract class in Python?

An abstract class in Python is a class that cannot be instantiated and is meant to be subclassed. It is created using the `abc` module with the `ABC` class and `@abstractmethod` decorator.

16. What are the advantages of OOP?

Advantages of OOP include modularity, reusability (via inheritance), flexibility (via polymorphism), maintainability, and better organization of code through encapsulation and abstraction.

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

A class variable is shared among all instances of a class and defined outside any method. An instance variable is unique to each object and typically defined in the **init** method.

18. What is multiple inheritance in Python?

Multiple inheritance in Python allows a class to inherit from more than one parent class, combining their attributes and methods. Python resolves method conflicts using the Method Resolution Order (MRO).

19.Explain the purpose of `__str__`, and `__repr__` methods in Python.

str `defines a user-friendly string representation of an object (for` print() `).` **repr** `provides a detailed, unambiguous representation, often for debugging (used by` repr()`).

20. What is the significance of the `super()` function in Python?

The `super()` function in Python is used to call a method from a parent class, often in the context of inheritance, to extend or modify the parent's behavior in a subclass.

21. What is the significance of the `__del__` method in Python?

The `__del__` method in Python is a destructor, called when an object is about to be destroyed (garbage collected), allowing cleanup actions like closing files or releasing resources.

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

A `@staticmethod` doesn't take `self` or `cls` and behaves like a regular function within a class. A `@classmethod` takes `cls` as its first parameter and can access or modify class-level data.

23. How does polymorphism work in Python with inheritance?

Polymorphism in Python with inheritance works by allowing subclasses to override methods of a parent class, so objects of different subclasses can be treated as instances of the parent class while executing their specific method implementations.

24. What is method chaining in Python OOP?

Method chaining in Python OOP is a technique where multiple methods are called on the same object in a single line, achieved by having each method return `self`.

25. What is the purpose of the `__call__` method in Python?

The `__call__` method in Python makes an object callable like a function. When defined in a class, it allows an instance of the class to be invoked with parentheses, e.g., `obj()`.

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".

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

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

# Test
dog = Dog()
dog.speak()  # Output: Bark
```

```
Bark
```

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

```
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, length, width):
        self.length = length
        self.width = width

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

# Test
```

```
circle = Circle(5)
rectangle = Rectangle(4, 6)
print(circle.area())  # Output: ~78.54
print(rectangle.area())  # Output: 24
```

```
78.53981633974483
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.

```
class Vehicle:
    def __init__(self, type):
        self.type = type

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

class ElectricCar(Car):
    def __init__(self, type, model, battery):
        super().__init__(type, model)
        self.battery = battery

# Test
electric_car = ElectricCar("Sedan", "Tesla Model 3", "100kWh")
print(electric_car.type, electric_car.model, electric_car.battery)  # Output: Sedan Tesla M
```

```
Sedan Tesla Model 3 100kWh
```

4. Demonstrate polymorphism by creating a base class Bird with a method fly(). Create two derived classes Sparrow and Penguin that override the fly() method.

```
class Bird:
    def fly(self):
        print("Bird can fly")

class Sparrow(Bird):
    def fly(self):
        print("Sparrow flies high")
```

```
class Penguin(Bird):
    def fly(self):
        print("Penguin cannot fly")

# Test
sparrow = Sparrow()
penguin = Penguin()
sparrow.fly()  # Output: Sparrow flies high
penguin.fly()  # Output: Penguin cannot fly
```

```
Sparrow flies high
Penguin cannot fly
```

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

```
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance  # Private attribute

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount

    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount

    def check_balance(self):
        return self.__balance

# Test
account = BankAccount(1000)
account.deposit(500)
account.withdraw(200)
print(account.check_balance())  # Output: 1300
```

```
1300
```

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().

```
class Instrument:
    def play(self):
        print("Instrument plays")

class Guitar(Instrument):
    def play(self):
        print("Guitar strums")

class Piano(Instrument):
    def play(self):
        print("Piano keys play")

# Test
guitar = Guitar()
piano = Piano()
guitar.play()  # Output: Guitar strums
piano.play()  # Output: Piano keys play
```

```
Guitar strums
Piano keys play
```

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.

```
class MathOperations:
    @classmethod
    def add_numbers(cls, a, b):
        return a + b

    @staticmethod
    def subtract_numbers(a, b):
        return a - b

# Test
print(MathOperations.add_numbers(5, 3))  # Output: 8
print(MathOperations.subtract_numbers(5, 3))  # Output: 2
```

```
8
2
```

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

```
class Person:
    count = 0  # Class attribute to track number of persons
    def __init__(self, name):
        self.name = name
        Person.count += 1

    @classmethod
    def get_count(cls):
        return cls.count

# Test
p1 = Person("Alice")
p2 = Person("Bob")
print(Person.get_count())  # Output: 2
```

```
2
```

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

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

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

# Test
fraction = Fraction(3, 4)
print(fraction)  # Output: 3/4
```

```
3/4
```

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

```
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"Vector({self.x}, {self.y})"
```

```
# Test
v1 = Vector(2, 3)
v2 = Vector(4, 5)
v3 = v1 + v2
print(v3)  # Output: Vector(6, 8)
```

```
Vector(6, 8)
```

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".

```
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")

# Test
person = Person("Alice", 30)
person.greet()  # Output: Hello, my name is Alice and I am 30 years old
```

```
Hello, my name is Alice and I am 30 years old
```

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

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

    def average_grade(self):
        return sum(self.grades) / len(self.grades) if self.grades else 0

# Test
student = Student("Alice", [85, 90, 95])
print(student.average_grade())  # Output: 90.0
```

```
90.0
```

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

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

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

    def get_area(self):
        return self.length * self.width

# Test
rect = Rectangle()
rect.set_dimensions(4, 5)
print(rect.get_area())  # Output: 20
```

```
20
```

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

```
class Employee:
    def __init__(self, hours, rate):
        self.hours = hours
        self.rate = rate

    def calculate_salary(self):
        return self.hours * self.rate

class Manager(Employee):
    def __init__(self, hours, rate, bonus):
        super().__init__(hours, rate)
        self.bonus = bonus

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

# Test
emp = Employee(40, 20)
```

```
mgr = Manager(40, 20, 500)
print(emp.calculate_salary())  # Output: 800
print(mgr.calculate_salary())  # Output: 1300
```

```
800
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.

```
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

# Test
product = Product("Laptop", 1000, 2)
print(product.total_price())  # Output: 2000
```

```
2000
```

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

```
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")

# Test
```

```
cow = Cow()
sheep = Sheep()
cow.sound()  # Output: Moo
sheep.sound()  # Output: Baa
```

```
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 details.

```
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: {self.year_published}"

# Test
book = Book("Python 101", "John Doe", 2020)
print(book.get_book_info())  # Output: Title: Python 101, Author: John Doe, Year: 2020
```

```
Title: Python 101, Author: John Doe, Year: 2020
```

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

```
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

# Test
mansion = Mansion("123 Main St", 500000, 10)
print(mansion.address, mansion.price, mansion.number_of_rooms)  # Output: 123 Main St 50000
```

```
123 Main St 500000 10
```