fqbwue56j

January 15, 2025

THEORY QUESTIONS

1.What is Object-Oriented Programming (OOP)? - OOPS stand for object-oriented programming system. Object-oriented programming is a computer programming model that organizes software design around objects, or data, instead of logic and functions. In OOP, objects are created from templates called classes, which define the behavior and properties of the objects they create.

2.what is class in oops? - A user-defined data type that describes the nature of an object. A class includes a declaration and definition, which can be split into separate files. A class can define the types of operations, or methods, that can be performed on an object.

3.what is an object in oops? - A single instance of a class, created with specifically defined data. Objects can represent real-world entities, such as a car or book, or abstract entities

4.what is a difference between abstraction and encapsulation? - Abstraction is a fundamental concept in Python programming that allows us to simplify complex concepts and focus on the essential details. It involves hiding unnecessary details and exposing only the relevant information to the users. - ENCAPSULATION:encapsulation is one of the fundamental concept of object oriented programming system which means bundling of data and method of a classs.

5.what are dunder methods in python? - Dunder methods, also known as magic methods or special methods, are predefined methods in Python that have double underscores at the beginning and end of their names. These methods provide a way to define specific behaviors for built-in operations or functionalities in Python classes.

6.explain the concept of inheritance in python. - Inheritance in Python is a feature that allows a class to inherit the properties and methods of another class. This allows to reuse code and write it once instead of copying it for each class that needs it.

7.what is polymorphism in python? - Polymorphism is a fundamental concept in object-oriented programming in Python that allows a single type of entity to represent many types in different contexts.poly means many nad phormism means forms.

8.how is encapsulation achieved in python? - in Python, encapsulation is achieved by using access modifiers to restrict access to a class's variables and methods: Public By default, all attributes and methods in a Python class are public and can be accessed from outside the class. Private To define a private variable, add two underscores as a prefix at the start of a variable name. Private members are only accessible within the class. Protected To indicate that an attribute is meant for internal use only within the class and its subclass, use a single underscore as a prefix.

9.what is a constructor in python? - A constructor in Python is a special method that initializes an object by assigning values to its data members when it's created.

10.what are class and static methods in python? - A class method takes cls as the first parameter while a static method needs no specific parameters. A class method can access or modify the class state while a static method can't access or modify it. In general, static methods know nothing about the class state.

11.what is method overloading in python? - Two or more methods have the same name but different numbers of parameters or different types of parameters, or both. These methods are called overloaded methods and this is called method overloading.

12.what is method overriding in oops? - Method overriding in Python occurs when a child class defines a method that has the same name and parameters as a method in its parent class. The child class method overrides or replaces the parent class method when called on an instance of the child class.

13.what is property decorator in python? - 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

14.why is polymorphism important in oops? - Polymorphism allows for flexible code design and code reuse. It also enables developers to write clean, readable, and resilient code.

15.what is an abstract class in python? - an abstract class is a class that cannot be instantiated on its own and is designed to be a blueprint for other classes: Abstract classes provide a common interface for subclasses while allowing them to provide specific implementations.

16.what are advantage of oops? - Encapsulation :Encapsulation combines data and code together, which allows developers to modify the internal details of an object without affecting the external code.

- Inheritance:Inheritance allows children classes to inherit properties and behaviors from the parent class, which promotes code reusability and reduces duplicate code.
- Code reusability:Objects and their behaviors can be reused in different parts of an application or even across multiple projects.
- Abstraction:abstraction provides only essential information about the data to the outside world, hiding the background details or implementation.
- Modularity:Modularity is achieved through encapsulation, which enhances code organization and makes it easier to debug and maintain.
- Code maintenance:OOP makes it easy for developers to incorporate new changes into the code. Scalability:OOP promotes scalability by making it easy to add new features or functionalities without changing existing code

17.What is the difference between a class variable and an instance variable? - Class variables:Class variables are useful for sharing data between all instances of a class, such as keeping track of the number of instances created or storing a constant value.

- Instance variables:Each instantiated object of a class has its own copy of an instance variable. Instance variables are useful for storing data that is specific to each instance, such as a person's name or age.

18.what is multiple inheritance in python. - If a child class is inheriting the properties of a single other class, we call it single inheritance. However, if a child class inherits from more than one class, i.e. this child class is derived from multiple classes, we call it multiple inheritance in Python.

19. Explain the purpose of ‘’**str**’ and ‘**repr**’ ‘ methods in Python?

- str:Returns a human-readable string representation of an object. It’s called by the print(), str(), and format() functions. The goal of **str** is to make the output readable and user-friendly.
- repr:Returns an unambiguous string representation of an object. It’s called by the repr() function. The goal of **repr** is to make the output clear and informative, often including enough information to recreate the object.

20.What is the significance of the ‘super()’ function in Python? - The super() function in Python is a built-in function that allows you to access methods and properties of a parent class from a child class. This is useful when working with inheritance in object-oriented programming.

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

- The del method in Python is a destructor method that’s called when an object is about to be destroyed. It’s used to: Perform cleanup actions, such as closing files or releasing locks.

22.What is the difference between @staticmethod and @classmethod in Python? - While a static method requires no specific parameters, a class method takes cls as its first argument. While a static method cannot access or modify the class state, a class method can. Static methods are typically unaware of the class state.

23. How does polymorphism work in Python with inheritance?

- Polymorphism, a child class method is allowed to have the same name as the class methods in the parent class. In inheritance, the methods belonging to the parent class are passed down to the child class. It’s also possible to change a method that a child class has inherited from its parent

24.What is method chaining in Python OOP? - Method chaining is a powerful technique in Python that enables us to perform a sequence of operations on an object in a concise and readable manner.

25.What is the purpose of the **call** method in Python? - The call method makes instances of a class callable, allowing them to be treated and executed like functions. This enhances the versatility of Python’s object-oriented programming paradigm

PRACTICAL QUESTIONS

[ ]:
```python
#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("this is generic message")
class dog(animal):
  def speak(self):
    print("bark!")
```

[ ]:
```python
obj=animal()
obj.speak()
```

```
this is generic message
```

[ ]:
```
obj2=dog()
obj2.speak()
```

```
bark!
```

[10]:
```
#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.
from abc import ABC, abstractmethod
class shape(ABC):
  def area(self):
    print("calculate the area")
class cirle(shape):
  def area(self,radius):
    return 3.14*radius*radius
class rectangle(shape):
  def area(self,length,width):
    return length*width
```

[9]:
```
#3.implement a multilevel inheritance where a class vehicle has an attribute␣
 ↪type.derive 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,battery):
    super().__init__(type)
    self.battery=battery
    class electriccar(car):
     def __init__(self,type,battery,range):
      super().__init__(type,battery)
      self.range=range
```

[8]:
```
#4.implement a multilevel inheritance where a class vehicle has an attribute␣
 ↪type.derive 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,battery):
    super().__init__(type)
    self.battery=battery
    class electriccar(car):
     def __init__(self,type,battery,range):
      super().__init__(type,battery)
      self.range=range
```

```
[ ]: #5.write a program to demostrate encapsulation by creating a class bankaccount␣
      ↪with private attributes balance and methods to deposits withdraw and check␣
      ↪balance.
     class bankaccount:
       def __init__(self,balance):
         self.__balance=balance
         def deposit(self,amount):
           self.__balance+=amount
       def withdraw(self,amount):
         if amount<=self.__balance:
           self.__balance-=amount
         else:
           print("insufficient balance")
```

```
[ ]: #6.demonstrate runtime polymorphism using a method pay() in a base class␣
      ↪instrument .derive classes guitar and piano that implements their owm␣
      ↪version of play().
     class instrument:
      def __init__(self,name):
       self.name=name
       def play(self):
         pass
     class guitar(instrument):
       def play(self):
         print("guitar is playing")
     class piano(instrument):
       def play(self):
         print("piano is playing")
```

```
[ ]: #7.create a class mathoperation with a class method add_numbers to add two␣
      ↪numbers and a static method substract_numbers() to substract two numbers.
     class mathoperation:
       @classmethod
       def add_numbers(cls,a,b):
         return a+b
         @staticmethod
         def substract_numbers(a,b):
           return a-b
```

```
[ ]: #8.implement a class person with a class method to count the total numbers of␣
      ↪person created.
     class person:
       count=0
       def __init__(self,name):
         self.name=name
```

```
    person.count+=1
    @classmethod
    def get_count(cls):
      return person.count
```

[ ]:
```
#9.write a class fraction with attrubute 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}"
```

[1]:
```
#10.demostrate operator overloading by creating a class vector and overirde 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)
```

[6]:
```
#11.create a class person with attributs 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")
```

[3]:
```
#12.implement a class student with attributs 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)
```

[11]:
```
#13.create a class rectangle with methods set_dimension() to set the dimension␣
↪and area() to calculate the area
class rectange:
  def __init__(self,length,width):
    self.length=length
    self.width=width
    def set_dimension(self,length,width):
     self.length=length
     self.width=width
    def area(self):
      return self.length*self.width
```

[12]:
```
#14.create a employee with method calculate_salary() that compute the salary␣
↪based om hours worked and hourly rate .create a derived class manager that␣
↪add bonus to the salary
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
```

[13]:
```
#15.create a class product with attributes name price quantity implement a␣
↪method totsl_price() thst calculate the total pricce 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
```

[15]:
```
#16.create a class animal with an abstract method sound().create two dervied␣
↪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")
```

[16]:
```
#17.create a class book with attribute titlr author and year_publshes add a␣
↪method get_book_info() that returns a formatted strng with books detild
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"{self.title} by {self.author} ({self.year_published})"
```

[20]:
```
#18.create a class house with attributes address and price create a deived␣
 ↪class mansion that adds an attributes numbers_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
        def get_house_info(self):
          return f"{self.address} for ${self.price} with {self.number_of_rooms}␣
 ↪rooms"
```

[ ]:
```

```