# **Python OOPs Questions**

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

* Object-Oriented Programming (OOP) is a method of programming where the focus is on objects rather than functions. An object is an entity that contains both data (attributes) and behavior (methods). OOP helps to represent real-world problems in code by grouping related data and functions together. The four major principles of OOP are Encapsulation (data hiding), Abstraction (hiding complexity), Inheritance (reusing code), and Polymorphism (many forms of functions/objects). This makes OOP more structured, reusable, and easier to maintain compared to procedural programming.

In [ ]:

class Car:

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

self.brand = brand

def drive(self):

print(f"{self.brand} is driving")

car1 = Car("Tesla")

car1.drive()

Tesla is driving

# **2. What is a class in OOP?**

* A class is the blueprint or template for creating objects. It defines attributes (variables) and methods (functions) but does not consume memory until an object is created. Classes provide structure to programs and allow multiple objects to be created from the same definition. For example, a Student class can define what every student has (name, roll number) and does (study).

In [ ]:

class Student:

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

self.name = name

self.roll = roll

def study(self):

print(f"{self.name} is studying")

s1 = Student("Akanksha", 101)

s1.study()

Akanksha is studying

# **3. What is an object in OOP?**

* An object is an instance of a class. It is the actual entity that holds real values and can perform actions defined in the class. While a class is only a definition, an object is the usable form of that class. Multiple objects can be created from one class, and each object can hold different data.

In [ ]:

class Dog:

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

self.name = name

def sound(self):

print(f"{self.name} barks")

dog1 = Dog("Tommy")

dog2 = Dog("Sheru")

dog1.sound()

dog2.sound()

Tommy barks

Sheru barks

# **4. What is the difference between abstraction and encapsulation?**

* Abstraction is the process of hiding implementation details and showing only essential features. For example, when we use a print() function, we don’t know how it works internally, we only know the output.
* Encapsulation means bundling data and methods inside a class and restricting direct access. This ensures security by preventing unauthorized modifications.

In [ ]:

class Bank:

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

self.\_\_balance = 1000 *# encapsulated*

def show\_balance(self):

print("Balance:", self.\_\_balance)

b = Bank()

b.show\_balance()

Balance: 1000

# **5. What are dunder methods in Python?**

* Dunder (double underscore) methods are special methods in Python used to perform specific operations. They start and end with **. Examples include \_\_init** (constructor), **str** (string representation), **len** (length). These methods make objects behave like built-in types.

In [ ]:

class Book:

def \_\_str\_\_(self):

return "Book object"

print(Book())

Book object

# **6. Explain the concept of inheritance in OOP.**

* Inheritance allows a class (child) to use the properties and methods of another class (parent). This avoids code duplication and supports reusability. Python supports single, multiple, and multilevel inheritance.

In [ ]:

class Animal:

def sound(self): print("Animal sound")

class Dog(Animal):

def sound(self): print("Dog barks")

d = Dog()

d.sound()

Dog barks

# **7. What is polymorphism in OOP?**

* Polymorphism means "many forms". In OOP, it allows the same function or method name to be used in different ways depending on the object or data type. This improves flexibility and code reusability. For example, the + operator works differently for integers and strings. In class-based polymorphism, methods of different classes can share the same name but perform different actions. This is powerful in real-world programming because it allows one interface to be used for different types of objects.

In [ ]:

print(len("hello")) *# String*

print(len([1,2,3])) *# List*

5

3

# **8. How is encapsulation achieved in Python?**

* Encapsulation means binding data (variables) and methods into a single unit and restricting direct access to some variables. In Python, this is achieved by making variables private using \_ or \_\_. Encapsulation increases security by preventing external modification of sensitive data. Getter and setter methods are often used to access or update private variables safely.

In [ ]:

class Bank:

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

self.\_\_balance = 1000

def get\_balance(self):

return self.\_\_balance

b = Bank()

print(b.get\_balance())

1000

# **9. What is a constructor in Python?**

* A constructor is a special method **init** in Python that is called automatically when an object is created. It initializes the object’s attributes with default or passed values. Without a constructor, we would need to set values manually after creating the object. This makes object creation easy and automatic.

In [ ]:

class Student:

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

self.name = name

s1 = Student("Akanksha")

print(s1.name)

Akanksha

# **10. What are class and static methods in Python?**

* Class methods are methods that work on the class rather than an object. They are defined using @classmethod and take cls as a parameter.
* Static methods are general functions inside a class that don’t need self or cls. They are defined using @staticmethod

In [ ]:

class Test:

@classmethod

def show\_cls(cls):

print("Class method called")

@staticmethod

def show\_stat():

print("Static method called")

Test.show\_cls()

Test.show\_stat()

Class method called

Static method called

# **11. What is method overloading in Python?**

* Method overloading means defining multiple methods with the same name but different parameters. Python doesn’t support it directly, but it can be done using default arguments or \*args. This makes methods more flexible to handle different input cases.

In [ ]:

def add(a, b=0):

return a+b

print(add(5))

print(add(5, 10))

5

15

# **12. What is method overriding in OOP?**

* Method overriding occurs when a child class provides a new version of a method that already exists in its parent class. This is used when we want different behavior for the same method in different classes. It’s key in achieving polymorphism in inheritance.

In [ ]:

class Animal:

def sound(self):

print("Animal makes sound")

class Dog(Animal):

def sound(self):

print("Dog barks")

d = Dog()

d.sound()

Dog barks

# **13. What is a property decorator in Python?**

* The @property decorator is used to define getters in Python. It allows methods to be accessed like attributes without parentheses. It is useful for controlled access to variables while still keeping code simple.

In [ ]:

class Person:

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

self.\_name = name

@property

def name(self):

return self.\_name

p = Person("Akanksha")

print(p.name)

Akanksha

# **14. Why is polymorphism important in OOP?**

* Polymorphism is important because it allows different objects to be treated through the same interface. It reduces code duplication and increases flexibility. For example, we can call the same method on multiple objects without worrying about their type, making code more general and reusable.

In [ ]:

for x in [len("Hi"), len([1,2,3,4])]:

print(x)

2

4

# **15. What is an abstract class in Python?**

* An abstract class in Python is a class that cannot be instantiated directly and is used as a blueprint for other classes. It contains one or more abstract methods (methods declared but not implemented). Abstract classes are useful when you want to define a common interface for all subclasses, ensuring that every subclass provides an implementation for those methods. This is achieved using the abc module and @abstractmethod decorator. Abstract classes promote code reusability and enforce rules across different subclasses.

In [1]:

from abc import ABC, abstractmethod

class Animal(ABC):

@abstractmethod

def sound(self):

pass

class Dog(Animal):

def sound(self):

return "Bark"

dog = Dog()

print(dog.sound())

Bark

# **16. What are the advantages of OOP?**

* Code reusability (through inheritance).
* Encapsulation (data hiding for better security).
* Polymorphism (different implementations for the same interface).
* Modularity (code is structured and easier to maintain).
* Scalability (adding new features is easier). In Python, OOP makes programs more structured, readable, and closer to real-world problem modeling.

In [2]:

class Car:

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

self.brand = brand

car1 = Car("BMW")

car2 = Car("Audi")

print(car1.brand, car2.brand)

BMW Audi

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

* A class variable is shared across all objects of a class, while an instance variable is unique for each object. Class variables are defined inside the class but outside methods, whereas instance variables are defined inside the constructor (**init**).

In [3]:

class Student:

school = "ABC School" *# class variable*

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

self.name = name *# instance variable*

s1 = Student("Amit")

s2 = Student("Riya")

print(s1.school, s2.school)

print(s1.name, s2.name)

ABC School ABC School

Amit Riya

# **18. What is multiple inheritance in Python?**

* Multiple inheritance means that a class can inherit from more than one parent class. This allows a child class to combine features of multiple base classes. However, it may create complexity when methods with the same name appear in parent classes (handled by Method Resolution Order - MRO).

In [4]:

class Father:

def skill(self):

return "Driving"

class Mother:

def skill(self):

return "Cooking"

class Child(Father, Mother):

pass

c = Child()

print(c.skill()) *# Follows MRO*

Driving

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

* In Python, the **str** and **repr** methods are special methods used to represent objects as strings, but they serve different purposes. The **str** method is intended to give a human-readable description of the object, making it more user-friendly when printing. On the other hand, the **repr** method is meant for developers, returning an “official” string representation of the object that ideally could be used to recreate the same object. If **str** is not defined, Python automatically falls back to **repr**. Together, these methods enhance debugging and improve how objects are displayed in logs or console output, ensuring clarity for both users and developers.

In [5]:

class Student:

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

self.name = name

def \_\_str\_\_(self):

return f"Student Name: {self.name}"

def \_\_repr\_\_(self):

return f"Student('{self.name}')"

s = Student("Amit")

print(str(s))

print(repr(s))

Student Name: Amit

Student('Amit')

# **20. What is the significance of the ‘super()’ function in Python?**

* The super() function plays a critical role in object-oriented programming by allowing child classes to access methods or constructors of their parent classes without directly naming them. This makes the code more maintainable and avoids errors if the parent class name changes. It is especially important in the case of multiple inheritance, as Python uses the Method Resolution Order (MRO) to decide which parent’s method to call. By using super(), developers ensure consistent and predictable behavior across complex class hierarchies. It also promotes code reuse by avoiding duplicate logic in child classes when some functionality already exists in the parent.

In [6]:

class Parent:

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

print("Parent Constructor")

class Child(Parent):

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

super().\_\_init\_\_() *# Calls Parent constructor*

print("Child Constructor")

c = Child()

Parent Constructor

Child Constructor

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

* The **del** method is known as the destructor in Python and is automatically called when an object is about to be destroyed by the garbage collector. Its main purpose is to handle cleanup tasks such as releasing resources, closing files, disconnecting from databases, or freeing memory that the object may be holding. While it provides control over the final stage of an object’s lifecycle, it should be used carefully because the exact time of destruction is not always guaranteed (depends on garbage collection). Thus, **del** is best reserved for essential cleanup operations that cannot be left to automatic memory management.

In [7]:

class Test:

def \_\_del\_\_(self):

print("Destructor called, object deleted")

t = Test()

del t

Destructor called, object deleted

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

* Both @staticmethod and @classmethod are special decorators used to define methods inside a class, but they differ in how they interact with the class. A static method does not take self or cls as its first argument—it behaves like a normal function but is grouped logically inside the class. It is useful for utility functions that do not depend on class or instance data. A class method, on the other hand, takes cls as the first argument, giving it access to class-level variables and allowing it to modify shared state across all objects. In short, static methods are independent of class data, while class methods are tied to the class itself.

In [8]:

class Demo:

school = "ABC"

@staticmethod

def greet():

return "Hello!" *# Independent of class*

@classmethod

def change\_school(cls, new):

cls.school = new *# Modifies class variable*

print(Demo.greet())

Demo.change\_school("XYZ")

print(Demo.school)

Hello!

XYZ

# **23. How does polymorphism work in Python with inheritance?**

* Polymorphism in Python means “many forms,” and in the context of OOP, it allows the same method name to behave differently depending on the object that calls it. With inheritance, polymorphism is achieved when a child class overrides methods from the parent class while keeping the same method signature. This ensures that when a method is called on an object, the appropriate version (parent or child) executes. This feature promotes flexibility and allows a single interface to be used for different types of objects, making programs easier to extend and scale without modifying existing code.

In [9]:

class Animal:

def speak(self):

return "Some sound"

class Dog(Animal):

def speak(self):

return "Bark"

class Cat(Animal):

def speak(self):

return "Meow"

a = Animal()

d = Dog()

c = Cat()

print(a.speak(), d.speak(), c.speak())

Some sound Bark Meow

# **24. What is method chaining in Python OOP?**

* Method chaining is a design technique where multiple methods are called on the same object in a single line. This is achieved when each method returns self (the object itself) instead of None. It improves code readability by reducing the number of lines and making operations look like a sequence of steps. Method chaining is widely used in frameworks like Pandas, Django, and SQLAlchemy, where multiple operations can be performed fluently. It makes programs elegant, but developers should ensure clarity, as chaining too many methods may make debugging harder.

In [10]:

class Number:

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

self.value = value

def add(self, n):

self.value += n

return self

def subtract(self, n):

self.value -= n

return self

n = Number(10).add(5).subtract(3)

print(n.value)

12

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

* The **call** method in Python makes an object behave like a function, meaning you can “call” an object with parentheses as if it were a function. This feature is particularly useful in scenarios like decorators, function wrappers, or machine learning models, where objects need to be invoked multiple times with varying inputs. By defining **call**, you allow a class instance to maintain state (like storing data or parameters) while also being directly executable. This combination of object state + function behavior makes it very powerful for designing reusable and flexible components in Python.

In [11]:

class Adder:

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

self.num = num

def \_\_call\_\_(self, x):

return self.num + x

add5 = Adder(5)

print(add5(10)) *# Object called like a function*

15

## **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!"**

In [12]:

class Animal:

def speak(self):

print("This animal makes a sound.")

class Dog(Animal):

def speak(self):

print("Bark!")

*# Example*

d = Dog()

d.speak()

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.**

In [13]:

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 \* self.radius

class Rectangle(Shape):

def \_\_init\_\_(self, length, width):

self.length = length

self.width = width

def area(self):

return self.length \* self.width

*# Example*

c = Circle(5)

r = Rectangle(4, 6)

print("Circle Area:", c.area())

print("Rectangle Area:", r.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.**

In [14]:

class Vehicle:

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

self.type = type

class Car(Vehicle):

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

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

self.brand = brand

class ElectricCar(Car):

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

super().\_\_init\_\_(type, brand)

self.battery = battery

*# Example*

e = ElectricCar("Car", "Tesla", "100 kWh")

print(e.type, e.brand, e.battery)

Car Tesla 100 kWh

# **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.**

In [15]:

class Bird:

def fly(self):

print("Some birds can fly.")

class Sparrow(Bird):

def fly(self):

print("Sparrow flies high!")

class Penguin(Bird):

def fly(self):

print("Penguins cannot fly.")

*# Example*

for b in [Sparrow(), Penguin()]:

b.fly()

Sparrow flies high!

Penguins 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.**

In [16]:

class BankAccount:

def \_\_init\_\_(self, balance=0):

self.\_\_balance = balance *# private attribute*

def deposit(self, amount):

self.\_\_balance += amount

def withdraw(self, amount):

if amount <= self.\_\_balance:

self.\_\_balance -= amount

else:

print("Insufficient balance!")

def check\_balance(self):

return self.\_\_balance

*# Example*

acc = BankAccount(1000)

acc.deposit(500)

acc.withdraw(200)

print("Balance:", acc.check\_balance())

Balance: 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().**

In [17]:

class Instrument:

def play(self):

print("Instrument is playing.")

class Guitar(Instrument):

def play(self):

print("Guitar is strumming.")

class Piano(Instrument):

def play(self):

print("Piano is playing melody.")

*# Example*

for i in [Guitar(), Piano()]:

i.play()

Guitar is strumming.

Piano is playing melody.

# **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.**

In [18]:

class MathOperations:

@classmethod

def add\_numbers(cls, a, b):

return a + b

@staticmethod

def subtract\_numbers(a, b):

return a - b

*# Example*

print(MathOperations.add\_numbers(10, 5))

print(MathOperations.subtract\_numbers(10, 5))

15

5

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

In [19]:

class Person:

count = 0

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

self.name = name

Person.count += 1

@classmethod

def total\_persons(cls):

return cls.count

*# Example*

p1 = Person("A")

p2 = Person("B")

print("Total persons:", Person.total\_persons())

Total persons: 2

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

In [20]:

class Fraction:

def \_\_init\_\_(self, numerator, denominator):

self.numerator = numerator

self.denominator = denominator

def \_\_str\_\_(self):

return f"{self.numerator}/{self.denominator}"

*# Example*

f = Fraction(3, 4)

print(f)

3/4

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

In [21]:

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})"

*# Example*

v1 = Vector(2, 3)

v2 = Vector(4, 5)

print(v1 + v2)

(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."**

In [22]:

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*

p = Person("Akanksha", 20)

p.greet()

Hello, my name is Akanksha and I am 20 years old.

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

In [23]:

class Student:

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

self.name = name

self.grades = grades

def average\_grade(self):

return sum(self.grades) / len(self.grades)

*# Example*

s = Student("Ravi", [80, 90, 70])

print("Average:", s.average\_grade())

Average: 80.0

# **13. Create a class Rectangle with methods set\_dimensions() to set the dimensions and area() to calculate the area.**

In [24]:

class Rectangle:

def set\_dimensions(self, length, width):

self.length = length

self.width = width

def area(self):

return self.length \* self.width

*# Example*

r = Rectangle()

r.set\_dimensions(5, 3)

print("Area:", r.area())

Area: 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.**

In [25]:

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

*# Example*

m = Manager(40, 50, 500)

print("Manager Salary:", m.calculate\_salary())

Manager Salary: 2500

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

In [26]:

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

*# Example*

p = Product("Laptop", 50000, 2)

print("Total Price:", p.total\_price())

Total Price: 100000

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

In [27]:

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!")

*# Example*

Cow().sound()

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.**

In [28]:

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}, published in {self.year\_published}"

*# Example*

b = Book("The Alchemist", "Paulo Coelho", 1988)

print(b.get\_book\_info())

'The Alchemist' by Paulo Coelho, published in 1988

# **18. Create a class House with attributes address and price. Create a derived class Mansion that adds an attribute number\_of\_rooms.**

In [29]:

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

*# Example*

m = Mansion("123 Street, Mumbai", 50000000, 10)

print(m.address, m.price, "Rooms:", m.number\_of\_rooms)

123 Street, Mumbai 50000000 Rooms: 10