INTERVIEW PREPARATION GUIDE | 2026 EDITION

100 OOPs Interview Questions and Answers

For freshers, intermediate and experienced developers.
Pillars, SOLID, design patterns and low-level design, in Python.

40
Freshers questions

35
Intermediate questions

25
Experienced questions

Code and Debug

codeanddebug.in | YouTube: @codeanddebug | Free to share for learning

How to use this guide

The 100 questions are grouped by level: **Freshers** (1 to 40), **Intermediate** (41 to 75) and **Experienced** (76 to 100). Freshers should master the first section completely and read the second. Experienced candidates are expected to answer all three.

Do not memorise answers word for word. Interviewers follow up with "give me a real example" and "how is this different in Java?". Practise explaining each answer out loud with an example of your own.

Every example is in Python and was run; the output shown is real. Notes explain where Java or C++ behave differently.

The interactive version of this guide has a class and object builder, an MRO explorer, a "spot the OOP principle" game and a SOLID refactoring walkthrough: codeanddebug.in/blog/oops-interview-questions.

Contents

Freshers (1 to 40)

1. What is object-oriented programming (OOP)?
2. What is the difference between procedural programming and OOP?
3. What are the four pillars of OOP?
4. What are a class and an object?
5. Does a class take memory? Does an object?
6. What is the difference between instance attributes and class attributes?
7. What is self in Python?
8. What is a constructor? What does __init__ do?
9. What are the types of constructors?
10. What is a destructor? Does Python have one?
11. What is encapsulation? Give an example.
12. What are access modifiers? How does Python handle public, protected and private?
13. What are getters and setters? How does @property work in Python?
14. What is abstraction?
15. What is the difference between abstraction and encapsulation?
16. What is inheritance?
17. What are the types of inheritance?
18. What does super() do?
19. What is method overriding?
20. What is polymorphism? What are compile-time and runtime polymorphism?
21. What is method overloading? Does Python support it?
22. What is the difference between method overloading and method overriding?
23. What is operator overloading?
24. What is duck typing?
25. What is an abstract class? How do you create one in Python?
26. What is an interface?
27. What is the difference between an abstract class and an interface?
28. What is the difference between instance methods, class methods and static methods?
29. What goes wrong with a mutable class attribute?
30. What is the difference between composition and inheritance?
31. What are association, aggregation and composition?
32. What are coupling and cohesion?
33. What is the difference between is and ==?
34. What is the difference between __str__ and __repr__?
35. What do isinstance() and issubclass() do?
36. What are the advantages and disadvantages of OOP?
37. Is Python a fully object-oriented language?
38. What is dynamic binding (late binding)?
39. Can you create an object of an abstract class? Can an abstract class have a constructor?
40. Design a simple class for a real-world object, such as a library book.

Intermediate (41 to 75)

41. What is the diamond problem? How does Python solve it with the MRO?
42. How does super() work with multiple inheritance?
43. What is a mixin?
44. What are magic (dunder) methods? Name the important ones.
45. Why must __eq__ and __hash__ be consistent?
46. What is the difference between __new__ and __init__?
47. What is the difference between a shallow copy and a deep copy of an object?
48. What are descriptors?
49. What is __slots__ and when would you use it?
50. What are dataclasses? When should you use them?
51. How do you make your own class iterable?
52. How do you write a class that works with the with statement?
53. What is a callable object (__call__)?
54. What is a Protocol (structural typing) in Python?
55. How do you create custom exceptions? Why use a class hierarchy for them?
56. What are the SOLID principles?
57. Explain the Single Responsibility Principle with an example.
58. Explain the Open/Closed Principle with an example.
59. Explain the Liskov Substitution Principle. Why is Square-inherits-Rectangle a problem?
60. Explain the Interface Segregation Principle.
61. Explain the Dependency Inversion Principle and dependency injection.
62. Why is composition usually preferred over inheritance?
63. What is the difference between tight coupling and loose coupling?
64. What are virtual functions? What is a vtable?
65. How do you prevent a class from being inherited or a method from being overridden?
66. What is the object class?
67. What is an immutable object? How do you make one in Python?
68. What are nested (inner) classes? When are they useful?
69. What is monkey patching?
70. How do you make objects comparable and sortable?
71. What is object slicing in C++?
72. What are generics? How do you write a generic class in Python?
73. What is the difference between __getattr__ and __getattribute__?
74. What is method chaining (a fluent interface)?
75. What is an Enum and why use it instead of plain constants?

Experienced (76 to 100)

76. What are design patterns? What are their main categories?
77. What is the Singleton pattern? How do you implement it in Python, and what are its problems?
78. What is the Factory pattern?
79. What is the difference between Factory Method and Abstract Factory?
80. What is the Builder pattern?
81. What is the Strategy pattern?
82. What is the Observer pattern?
83. What is the Decorator pattern? How is it different from a Python @decorator?
84. What is the Adapter pattern?
85. What are the Template Method and Facade patterns?
86. What is the State pattern?
87. What is a metaclass?
88. What is __init_subclass__? How do you build a plugin registry with it?
89. What are common OOP anti-patterns?
90. Design a parking lot system (low-level design).
91. Design an LRU cache class.
92. How do you approach any low-level design (LLD) interview question?
93. What is the difference between OOP and functional programming?
94. When should you avoid OOP or inheritance?
95. How do you make a class thread-safe?
96. What is the difference between an entity and a value object?
97. How do you unit test a class that depends on other classes?
98. When should you use a regular class, a dataclass, a NamedTuple or Pydantic?
99. What is single dispatch? How is it different from method overloading?
100. How would you refactor a large, messy class?

Freshers level

Classes and objects, constructors, the four pillars, access modifiers, overloading and overriding, abstract classes and interfaces, and common Python OOP traps.

Q1. What is object-oriented programming (OOP)?

FRESHERS · BASICS

Object-oriented programming is a way of writing programs by modelling them as a set of **objects**. Each object bundles some **data** (attributes) together with the **behaviour** that works on that data (methods). A banking app might have Account, Customer and Transaction objects, each responsible for its own data.

The goal is to make large programs easier to understand, reuse and change. Objects hide their internal details, share behaviour through inheritance, and can be swapped for one another through polymorphism. Python, Java, C++, C# and Kotlin all support OOP.

Q2. What is the difference between procedural programming and OOP?

FRESHERS · BASICS

Point	Procedural	Object-oriented
Program is built from	Functions that act on data	Objects that own data and behaviour
Data	Often global or passed around freely	Kept inside objects and accessed through methods
Focus	Steps to perform	Things in the problem and how they interact
Reuse	By calling functions	Through inheritance and composition
Security of data	Weak; any function can change it	Stronger through encapsulation
Best for	Small scripts, simple tasks	Large applications that change over time
Examples	C, early BASIC	Java, C++, Python, C#

Python supports both styles, and real projects often mix them: plain functions for small utilities, classes for the main entities.

Q3. What are the four pillars of OOP?

FRESHERS · BASICS

- **Encapsulation:** keep data and the methods that change it together inside a class, and control access to the data.
- **Abstraction:** show only what an object does and hide how it does it. You call `car.start()` without knowing how the engine works.
- **Inheritance:** a new class reuses and extends an existing class. A SavingsAccount *is an* Account.
- **Polymorphism:** the same call behaves differently depending on the object. `shape.area()` works for a circle and a square, each with its own formula.

The "spot the OOP principle" game on this page has short snippets for you to classify.

Q4. What are a class and an object?

FRESHERS · CLASSES AND OBJECTS

A **class** is a blueprint that defines what data and behaviour its objects will have. An **object** (or instance) is a real thing created from that blueprint, with its own values.

The class `Student` says every student has a name and marks. Asha and Rohan are two objects of that class, each with their own name and marks.

PYTHON (TESTED)

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

    def result(self):
        return "Pass" if self.marks >= 40 else "Fail"

asha = Student("Asha", 82)
rohan = Student("Rohan", 35)

print(asha.name, asha.result())
print(rohan.name, rohan.result())
print(type(asha).__name__, isinstance(asha, Student))
```

OUTPUT

```
Asha Pass
Rohan Fail
Student True
```

Q5. Does a class take memory? Does an object?

FRESHERS · CLASSES AND OBJECTS

In languages like Java and C++, a class definition by itself does not allocate memory for data. Memory for fields is allocated when an **object** is created (with `new` in Java, or on the stack or heap in C++). The class's code and metadata are loaded once and shared by all objects.

In Python, a class is itself an object (of type `type`), so it does take some memory: it holds its methods and class attributes in a dictionary. Each instance then takes extra memory for its own attributes. The usual interview answer is still: the class is the blueprint, and each object gets its own memory for its data.

Q6. What is the difference between instance attributes and class attributes?

FRESHERS · CLASSES AND OBJECTS

- **Instance attributes** belong to one object and are usually set in `__init__` with `self.x = ...`. Each object has its own copy.
- **Class attributes** are defined in the class body and shared by all objects of that class. They suit constants and counters.

Reading `obj.x` first looks in the object, then in the class. Assigning `obj.x = ...` always creates or changes the instance attribute, and hides the class attribute for that object only. Java calls class attributes **static fields**.

PYTHON (TESTED)

```
class Employee:
    company = "Code and Debug"      # class attribute: shared
    count = 0

    def __init__(self, name):
        self.name = name            # instance attribute: per object
        Employee.count += 1

a = Employee("Asha")
b = Employee("Rohan")
print(a.company, b.company, Employee.count)

a.company = "Other Co"              # creates an instance attribute on a only
print(a.company, b.company, Employee.company)
```

OUTPUT

```
Code and Debug Code and Debug 2
Other Co Code and Debug Code and Debug
```

Q7. What is self in Python?

FRESHERS · CLASSES AND OBJECTS

`self` is the first parameter of every instance method. It refers to the object the method was called on, so the method can read and change that object's attributes.

When you write `acc.deposit(500)`, Python actually calls `Account.deposit(acc, 500)`. The name `self` is only a convention, but you should always use it.

Java and C++ note: the equivalent is `this`, which is passed automatically and does not appear in the parameter list.

PYTHON (TESTED)

```
class Account:
    def __init__(self, balance):
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount
        return self.balance

acc = Account(1000)
print(acc.deposit(500))
print(Account.deposit(acc, 500))   # the same call, written out
```

OUTPUT

```
1500
2000
```

Q8. What is a constructor? What does __init__ do?

FRESHERS · CONSTRUCTORS

A constructor is a special method that runs automatically when an object is created. It sets up the object's starting state.

In Python, `__init__` is the initialiser: it receives the new, empty object as `self` and sets its attributes. (Strictly, `__new__` creates the object and `__init__` initialises it; see the intermediate section.) `__init__` must return `None`.

Java and C++ note: a constructor has the same name as the class and no return type, and it is called with `new ClassName(...)` in Java.

PYTHON (TESTED)

```
class Course:
    def __init__(self, title, price=0):
        self.title = title
        self.price = price
        print(f"Created course: {title}")

c1 = Course("Python Full Course")
c2 = Course("Master DSA", 4999)
print(c1.price, c2.price)
```

OUTPUT

```
Created course: Python Full Course
Created course: Master DSA
0 4999
```

Q9. What are the types of constructors?

FRESHERS · CONSTRUCTORS

- **Default constructor:** takes no arguments. If you write no constructor, Java and C++ give you one automatically.
- **Parameterised constructor:** takes arguments to set the initial values.
- **Copy constructor (C++):** creates a new object as a copy of an existing one, `Point(const Point& other)`.

Python note: a class has only one `__init__`. You get the effect of several constructors with default arguments, or with **class methods as alternative constructors**, such as `datetime.fromtimestamp()`. Copies are made with the `copy` module.

PYTHON (TESTED)

```
class Point:
    def __init__(self, x=0, y=0):          # default and parameterised in one
        self.x, self.y = x, y

    @classmethod
    def from_string(cls, text):            # alternative constructor
        x, y = map(int, text.split(","))
        return cls(x, y)

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

print(Point(), Point(3, 4), Point.from_string("7,2"))
```

OUTPUT

```
Point(0, 0) Point(3, 4) Point(7, 2)
```

Q10. What is a destructor? Does Python have one?

FRESHERS · CONSTRUCTORS

A destructor runs when an object is destroyed, to release resources such as files or network connections.

- **C++:** `~ClassName()` runs at a predictable time: when the object goes out of scope or is deleted.
- **Java:** has no destructor. Objects are cleaned up by the garbage collector at an unpredictable time, so resources are closed with `try-with-resources`.
- **Python:** `__del__` is called when the object's reference count drops to zero. In CPython that is usually immediate, but it is not guaranteed in every situation (for example reference cycles, or interpreter shutdown).

So in Python, do not rely on `__del__` for important cleanup. Use a context manager (`with`) instead.

PYTHON (TESTED)

```
class TempFile:
    def __init__(self, name):
        self.name = name
        print("open", name)
    def __del__(self):
        print("cleanup", self.name)

f = TempFile("report.txt")
del f                       # last reference removed
print("after del")
```

OUTPUT

```
open report.txt
cleanup report.txt
after del
```

Q11. What is encapsulation? Give an example.

FRESHERS · ENCAPSULATION

Encapsulation means bundling data with the methods that work on it, and controlling how that data can be changed from outside. Other code interacts with the object through its methods instead of editing its internals directly.

In a bank account, the balance should never go negative. If any code could write `acc.balance = -5000`, that rule would break. With encapsulation, the balance can only change through `deposit` and `withdraw`, which check the rule.

PYTHON (TESTED)

```
class BankAccount:
    def __init__(self, owner, balance=0):
        self.owner = owner
        self.__balance = balance            # private by name mangling

    def deposit(self, amount):
        if amount <= 0:
            raise ValueError("Deposit must be positive")
        self.__balance += amount

    def withdraw(self, amount):
        if amount > self.__balance:
            raise ValueError("Insufficient balance")
        self.__balance -= amount

    def get_balance(self):
        return self.__balance

acc = BankAccount("Asha", 1000)
acc.deposit(500)
try:
    acc.withdraw(5000)
except ValueError as e:
    print("Error:", e)
print(acc.get_balance())
```

OUTPUT

```
Error: Insufficient balance
1500
```

Q12. What are access modifiers? How does Python handle public, protected and private?

FRESHERS · ENCAPSULATION

Modifier	Java and C++	Python convention
Public	Accessible everywhere	`name`
Protected	Class and subclasses (Java: also the same package)	`_name`: a hint, "internal, do not use from outside"
Private	Only inside the class; enforced by the compiler	`__name`: renamed to `_ClassName__name` (name mangling)
Default (package-private)	Java only: same package	No equivalent

Python has no enforced access control. A single underscore is a convention, and a double underscore only makes accidental access and clashes in subclasses less likely. The Python saying is "we are all consenting adults here".

PYTHON (TESTED)

```
class User:
    def __init__(self):
        self.name = "Asha"        # public
        self._role = "admin"      # protected by convention
        self.__password = "s3cr3t" # private: name-mangled

u = User()
print(u.name, u._role)
try:
    print(u.__password)
except AttributeError as e:
    print("AttributeError:", e)
print(u._User__password)          # still reachable, but you should not do this
```

OUTPUT

```
Asha admin
AttributeError: 'User' object has no attribute '__password'
s3cr3t
```

Q13. What are getters and setters? How does @property work in Python?

FRESHERS · ENCAPSULATION

Getters and setters are methods that read and change an attribute, so the class can validate values or compute them. Java code is full of `getAge()` and `setAge()`.

In Python you start with a plain attribute. If you later need validation, you turn it into a **property**: code still writes `p.age = 30`, but your setter runs. So you never have to write getters and setters in advance.

PYTHON (TESTED)

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age               # goes through the setter below

    @property
    def age(self):
        return self._age

    @age.setter
    def age(self, value):
        if not 0 <= value <= 150:
            raise ValueError("Age must be between 0 and 150")
        self._age = value

p = Person("Rohan", 25)
p.age = 26
print(p.age)
try:
    p.age = -5
except ValueError as e:
    print("Error:", e)
```

OUTPUT

```
26
Error: Age must be between 0 and 150
```

Q14. What is abstraction?

FRESHERS · ABSTRACTION

Abstraction means exposing only the essential features of an object and hiding the complex details of how they work. Users of the object depend on **what** it does, not **how**.

When you call `payment.pay(500)`, you do not need to know whether it goes through UPI, a card network or a wallet. The details are hidden behind a simple interface, and they can change without breaking the code that calls it.

In code, abstraction is achieved with abstract classes and interfaces (which define what must be done), and with well-chosen public methods that hide internal steps.

Q15. What is the difference between abstraction and encapsulation?

FRESHERS · ABSTRACTION

Point	Abstraction	Encapsulation
Purpose	Hide complexity; show only what matters	Protect data; control how it is accessed
Focus	Design: what an object does	Implementation: how data is kept safe
Achieved with	Abstract classes, interfaces, simple public methods	Private attributes, getters, setters, properties
Question it answers	"What should the user see?"	"Who can change this data?"
Example	A `Payment` interface with `pay()`	A private `__balance` changed only by `deposit()`

They work together: encapsulation is one of the tools that makes abstraction possible.

Q16. What is inheritance?

FRESHERS · INHERITANCE

Inheritance lets a new class (the **child**, subclass or derived class) reuse the attributes and methods of an existing class (the **parent**, superclass or base class), then add or change behaviour. It models an **"is-a"** relationship: a Dog *is an* Animal.

Benefits: code reuse, a clear hierarchy, and the ability to treat different subclasses through the parent type (polymorphism).

PYTHON (TESTED)

```
class Animal:
    def __init__(self, name):
        self.name = name
    def describe(self):
        return f"{self.name} is an animal"
    def sound(self):
        return "..."

class Dog(Animal):                  # Dog inherits from Animal
    def sound(self):                # and overrides one method
        return "Woof"

d = Dog("Tommy")
print(d.describe())                 # inherited
print(d.sound())                    # overridden
print(isinstance(d, Animal), issubclass(Dog, Animal))
```

OUTPUT

```
Tommy is an animal
Woof
True True
```

Q17. What are the types of inheritance?

FRESHERS · INHERITANCE

- **Single:** one child, one parent. `Dog(Animal)`.
- **Multilevel:** a chain. `Puppy(Dog)`, `Dog(Animal)`.
- **Hierarchical:** many children of one parent. `Dog(Animal)`, `Cat(Animal)`.
- **Multiple:** one child with several parents. `FlyingCar(Car, Aircraft)`.
- **Hybrid:** a mix of the above, which can create the diamond problem.

Java note: a class can extend only one class, so Java has no multiple inheritance of classes. It gets a similar effect by implementing several interfaces. **C++** and **Python** support multiple inheritance; Python resolves conflicts with the MRO (see the intermediate section and the MRO explorer on this page).

Q18. What does super() do?

FRESHERS · INHERITANCE

`super()` gives you access to methods of the parent class (strictly, the next class in the MRO). It is mostly used to call the parent's `__init__` so the parent part of the object is set up, and to extend a parent method instead of replacing it completely.

Java note: `super(...)` calls the parent constructor and must be the first statement in a constructor; `super.method()` calls the parent's version of a method.

PYTHON (TESTED)

```
class Account:
    def __init__(self, owner, balance):
        self.owner = owner
        self.balance = balance
    def summary(self):
        return f"{self.owner}: Rs {self.balance}"

class SavingsAccount(Account):
    def __init__(self, owner, balance, rate):
        super().__init__(owner, balance)   # let the parent set its part
        self.rate = rate
    def summary(self):
        return super().summary() + f" at {self.rate}% interest"

print(SavingsAccount("Asha", 5000, 7).summary())
```

OUTPUT

```
Asha: Rs 5000 at 7% interest
```

Q19. What is method overriding?

FRESHERS · POLYMORPHISM

Method overriding is when a child class provides its own version of a method that already exists in the parent class, with the same name and parameters. When the method is called on a child object, the child's version runs.

It is how **runtime polymorphism** works: code written against the parent type automatically uses the right behaviour for each child.

Java note: mark overriding methods with `@Override` so the compiler checks the signature. A `final` method cannot be overridden, and static methods are hidden, not overridden.

PYTHON (TESTED)

```
class Notification:
    def send(self, msg):
        return f"Generic: {msg}"

class Email(Notification):
    def send(self, msg):
        return f"Email: {msg}"

class SMS(Notification):
    def send(self, msg):
        return f"SMS: {msg[:20]}"

for n in [Notification(), Email(), SMS()]:
    print(n.send("Your class starts at 7 pm tonight"))
```

OUTPUT

```
Generic: Your class starts at 7 pm tonight
Email: Your class starts at 7 pm tonight
SMS: Your class starts at
```

Q20. What is polymorphism? What are compile-time and runtime polymorphism?

FRESHERS · POLYMORPHISM

Polymorphism ("many forms") means one interface can work with objects of different types, and each type responds in its own way.

Point	Compile-time (static)	Runtime (dynamic)
Achieved by	Method overloading, operator overloading	Method overriding
Decided	By the compiler, from argument types	At runtime, from the object's actual type
Also called	Early binding	Late binding, dynamic dispatch
Example	`add(int, int)` vs `add(double, double)` in Java	`shape.area()` on a Circle or a Square

Python note: Python decides everything at runtime, so all its polymorphism is runtime polymorphism, including duck typing.

Q21. What is method overloading? Does Python support it?

FRESHERS · POLYMORPHISM

Method overloading means defining several methods with the **same name but different parameters** (number or types) in one class. Java and C++ support it, and the compiler picks the right version.

Python does not: if you define two methods with the same name, the second one replaces the first. You get the same effect with default arguments, `*args`, or `functools.singledispatchmethod` to choose by type.

PYTHON (TESTED)

```
class Calculator:
    def add(self, a, b):
        return a + b
    def add(self, a, b, c):          # replaces the first add
        return a + b + c

try:
    Calculator().add(1, 2)
except TypeError as e:
    print("TypeError:", e)

class BetterCalculator:
    def add(self, *nums):             # one method, any number of arguments
        return sum(nums)

print(BetterCalculator().add(1, 2), BetterCalculator().add(1, 2, 3))
```

OUTPUT

```
TypeError: Calculator.add() missing 1 required positional argument: 'c'
3 6
```

Q22. What is the difference between method overloading and method overriding?

FRESHERS · POLYMORPHISM

Point	Overloading	Overriding
Where	Same class	Parent and child classes
Method signature	Same name, different parameters	Same name and same parameters
Type of polymorphism	Compile-time	Runtime
Inheritance needed	No	Yes
Return type (Java)	Can differ	Same, or a subtype (covariant return)
Python support	Not directly	Yes

Q23. What is operator overloading?

FRESHERS · POLYMORPHISM

Operator overloading lets your own classes work with operators like `+`, `==`, `<` or `len()`. In Python you do it by defining special ("dunder") methods: `__add__` for `+`, `__eq__` for `==`, `__lt__` for `<`, `__len__` for `len()`.

Java note: Java does not allow operator overloading (apart from the built-in `+` for strings). C++ does, with functions such as `operator+`.

PYTHON (TESTED)

```
class Money:
    def __init__(self, rupees):
        self.rupees = rupees
    def __add__(self, other):
        return Money(self.rupees + other.rupees)
    def __lt__(self, other):
        return self.rupees < other.rupees
    def __repr__(self):
        return f"Rs {self.rupees}"

fees, books = Money(4999), Money(800)
print(fees + books)
print(books < fees)
print(sorted([fees, books]))
```

OUTPUT

```
Rs 5799
True
[Rs 800, Rs 4999]
```

Q24. What is duck typing?

FRESHERS · POLYMORPHISM

"If it walks like a duck and quacks like a duck, it is a duck." In duck typing, Python does not check an object's class; it only checks that the object has the method you call. Any object with the right methods works, even with no shared parent class.

This is why `len()` works on lists, strings, dicts and your own classes that define `__len__`, and why any object with `read()` can be used where a file is expected.

Java note: Java needs an explicit shared type, such as an interface, for this kind of polymorphism. Python's `typing.Protocol` adds type-checker support for duck typing.

PYTHON (TESTED)

```
class Dog:
    def speak(self): return "Woof"

class Robot:
    def speak(self): return "Beep"

def make_it_speak(thing):           # no type check at all
    return thing.speak()

print(make_it_speak(Dog()), make_it_speak(Robot()))
```

OUTPUT

```
Woof Beep
```

Q25. What is an abstract class? How do you create one in Python?

FRESHERS · ABSTRACTION

An abstract class is a class that cannot be instantiated and exists to be inherited from. It can contain **abstract methods** (declared but not implemented) that every concrete subclass must implement, as well as normal methods with shared code.

In Python, inherit from `ABC` and mark methods with `@abstractmethod`. Python then refuses to create an object of any class that has not implemented every abstract method.

Java note: use the `abstract` keyword on the class and the methods.

PYTHON (TESTED)

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self): ...

    def describe(self):                       # shared, concrete method
        return f"{type(self).__name__} with area {self.area():.2f}"

class Circle(Shape):
    def __init__(self, r): self.r = r
    def area(self): return 3.14159 * self.r ** 2

class Square(Shape):
    def __init__(self, s): self.s = s
    def area(self): return self.s ** 2

for s in [Circle(2), Square(3)]:
    print(s.describe())

try:
    Shape()
except TypeError as e:
    print("TypeError:", e)
```

OUTPUT

```
Circle with area 12.57
Square with area 9.00
TypeError: Can't instantiate abstract class Shape without an implementation for abstract method 'area'
```

Q26. What is an interface?

FRESHERS · ABSTRACTION

An interface is a contract: a list of methods a class promises to provide, with no shared state. Code can depend on the interface, and any class that implements it can be plugged in.

- **Java:** declared with `interface`, implemented with `implements`. A class can implement many interfaces. Since Java 8, interfaces can have `default` and `static` methods.
- **C++:** no keyword; a class with only pure virtual functions acts as an interface.
- **Python:** no keyword; use an abstract base class with only abstract methods, or a `typing.Protocol` for structural (duck-typed) interfaces.

Q27. What is the difference between an abstract class and an interface?

FRESHERS · ABSTRACTION

Point	Abstract class	Interface
Methods	Abstract and concrete	Abstract (Java 8+: also default and static)
State (fields)	Can have instance variables	Only constants (Java)
Constructors	Yes	No
Inheritance	A class extends only one (Java)	A class can implement many
Access modifiers	Any	Methods public by default
Relationship	"Is-a" with shared code	"Can-do" capability
Use when	Related classes share code and state	Unrelated classes share a capability, such as Comparable or Serializable

Q28. What is the difference between instance methods, class methods and static methods?

FRESHERS · METHODS

Type	First parameter	Can access	Typical use
Instance method	`self` (the object)	Object and class data	Most behaviour
Class method (`@classmethod`)	`cls` (the class)	Class data, not a specific object	Alternative constructors, class-wide counters
Static method (`@staticmethod`)	None	Neither, unless passed in	Helper functions that belong with the class

Java note: Java has instance methods and `static` methods; there is no direct equivalent of a class method that receives the class.

PYTHON (TESTED)

```
class Pizza:
    def __init__(self, toppings):
        self.toppings = toppings

    def describe(self):                          # instance method
        return "Pizza with " + ", ".join(self.toppings)

    @classmethod
    def margherita(cls):                         # alternative constructor
        return cls(["tomato", "mozzarella"])

    @staticmethod
    def is_valid_size(inches):                   # plain helper
        return inches in (8, 10, 12)

print(Pizza.margherita().describe())
print(Pizza.is_valid_size(10), Pizza.is_valid_size(9))
```

OUTPUT

```
Pizza with tomato, mozzarella
True False
```

Q29. What goes wrong with a mutable class attribute?

FRESHERS · CLASSES AND OBJECTS

A class attribute is shared by every object. If it is mutable, such as a list or dict, changing it through one object changes it for all of them. This is one of the most common Python OOP bugs, and a favourite trick question.

The fix is to create mutable data per object in `__init__`.

PYTHON (TESTED)

```
class Batch:
    students = []                    # shared by ALL batches: a bug
    def add(self, name):
        self.students.append(name)

morning, evening = Batch(), Batch()
morning.add("Asha")
print(evening.students)              # evening sees Asha too

class FixedBatch:
    def __init__(self):
        self.students = []           # one list per object
    def add(self, name):
        self.students.append(name)

m, e = FixedBatch(), FixedBatch()
m.add("Asha")
print(m.students, e.students)
```

OUTPUT

```
['Asha']
['Asha'] []
```

Q30. What is the difference between composition and inheritance?

FRESHERS · RELATIONSHIPS

- **Inheritance** models **"is-a"**: a Car *is a* Vehicle. The child gets the parent's code.
- **Composition** models **"has-a"**: a Car *has an* Engine. One object holds another and delegates work to it.

Point	Inheritance	Composition
Coupling	Tight: child depends on parent internals	Loose: depends only on the part's public methods
Flexibility	Fixed when the class is written	Parts can be swapped, even at runtime
Risk	Deep hierarchies that are hard to change	More small classes to wire together

The common advice is "favour composition over inheritance": use inheritance only for a true "is-a" relationship where the child can stand in for the parent everywhere.

PYTHON (TESTED)

```
class Engine:
    def start(self): return "engine started"

class ElectricMotor:
    def start(self): return "motor humming"

class Car:
    def __init__(self, engine):          # Car HAS an engine
        self.engine = engine
    def drive(self):
        return f"Car: {self.engine.start()}, driving"

print(Car(Engine()).drive())
print(Car(ElectricMotor()).drive())      # swap the part, same Car class
```

OUTPUT

```
Car: engine started, driving
Car: motor humming, driving
```

Q31. What are association, aggregation and composition?

FRESHERS · RELATIONSHIPS

All three describe how objects relate. They differ in ownership and lifetime:

Relationship	Meaning	Lifetime	Example
Association	Objects use or know each other	Independent	A Teacher and a Student
Aggregation	A whole has parts, but the parts can exist alone ("weak has-a")	Parts outlive the whole	A Department has Professors; closing the department does not delete them
Composition	A whole owns its parts ("strong has-a")	Parts die with the whole	A House has Rooms; demolish the house and the rooms are gone

In UML, aggregation is drawn with a hollow diamond and composition with a filled diamond on the "whole" side.

Q32. What are coupling and cohesion?

FRESHERS · DESIGN

- **Cohesion** is how closely the responsibilities inside one class belong together. **High cohesion is good:** an `InvoicePrinter` that only prints invoices.
- **Coupling** is how much one class depends on the details of another. **Low coupling is good:** changing one class should not force changes in many others.

The goal is **high cohesion, low coupling**. You get there by giving each class one clear job, depending on interfaces instead of concrete classes, and passing dependencies in rather than creating them inside. The SOLID refactoring tool on this page shows this step by step.

Q33. What is the difference between is and ==?

FRESHERS · PYTHON SPECIFICS

- `==` checks **equality**: do the two objects have the same value? It calls `__eq__`, which you can define.
- `is` checks **identity**: are they the very same object in memory?

Use `is` only for singletons such as `None` (`if x is None`). For values, use `==`. By default, a class without `__eq__` compares by identity, so two objects with the same data are not equal until you define it.

Java note: the same trap exists: `==` compares references and `.equals()` compares values, which is why `"a" == new String("a")` is false.

PYTHON (TESTED)

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

class Point2:
    def __init__(self, x, y): self.x, self.y = x, y
    def __eq__(self, other): return (self.x, self.y) == (other.x, other.y)

print(Point(1, 2) == Point(1, 2))     # no __eq__: compares identity
print(Point2(1, 2) == Point2(1, 2))   # value equality
a = Point2(1, 2); b = a
print(a is b, Point2(1, 2) is Point2(1, 2))
```

OUTPUT

```
False
True
True False
```

Q34. What is the difference between __str__ and __repr__?

FRESHERS · PYTHON SPECIFICS

- `__str__` returns a readable string for users. It is used by `print()` and `str()`.
- `__repr__` returns an unambiguous string for developers, ideally one that looks like code to recreate the object. It is used in the console, in lists, and as a fallback when `__str__` is missing.

If you write only one, write `__repr__`. **Java note:** the equivalent is overriding `toString()`.

PYTHON (TESTED)

```
class Course:
    def __init__(self, title, price):
        self.title, self.price = title, price
    def __str__(self):
        return f"{self.title} (Rs {self.price})"
    def __repr__(self):
        return f"Course({self.title!r}, {self.price})"

c = Course("Master DSA", 4999)
print(c)          # __str__
print(repr(c))    # __repr__
print([c])        # lists use __repr__
```

OUTPUT

```
Master DSA (Rs 4999)
Course('Master DSA', 4999)
[Course('Master DSA', 4999)]
```

Q35. What do isinstance() and issubclass() do?

FRESHERS · PYTHON SPECIFICS

- `isinstance(obj, Cls)` is true if `obj` is an instance of `Cls` **or any of its subclasses**. It accepts a tuple of classes.
- `issubclass(A, B)` is true if class `A` inherits from `B` (a class counts as a subclass of itself).

Prefer `isinstance` over `type(obj) == Cls`, because the exact type check breaks for subclasses. Even better, avoid type checks where duck typing or polymorphism can do the job.

PYTHON (TESTED)

```
class Animal: pass
class Dog(Animal): pass

d = Dog()
print(isinstance(d, Dog), isinstance(d, Animal), isinstance(d, (int, str)))
print(type(d) == Animal)              # False: exact type check
print(issubclass(Dog, Animal), issubclass(Animal, Dog), issubclass(Dog, object))
```

OUTPUT

```
True True False
False
True False True
```

Q36. What are the advantages and disadvantages of OOP?

FRESHERS · BASICS

Advantages:

- Models real-world problems naturally.
- Code reuse through inheritance and composition.
- Encapsulation protects data and limits the effect of changes.
- Polymorphism lets you add new types without changing existing code.
- Easier to split work across a team, one class or module at a time.

Disadvantages:

- More code and design effort for small programs.
- Deep inheritance hierarchies become rigid and hard to understand.
- Some overhead from objects and method calls.
- Shared mutable state inside objects makes concurrency harder; functional styles avoid this.

Q37. Is Python a fully object-oriented language?

FRESHERS · PYTHON SPECIFICS

In Python, **everything is an object**: numbers, strings, functions, classes and modules all have a type and can have attributes and methods. Even `5` is an instance of the class `int`.

But Python is **multi-paradigm**: it does not force you to write classes, and it supports procedural and functional programming too. It also has no enforced private members. So the usual answer is: Python is object-based to its core and fully supports OOP, but it is not "purely" object-oriented in the strict sense some people apply to languages like Smalltalk.

Java note: Java is often called "not purely OOP" because of its primitive types (`int`, `double`) and static methods.

PYTHON (TESTED)

```
print(type(5), type("hi"), type(len), type(int))
print((5).bit_length(), "hi".upper())
def greet(): return "hello"
greet.author = "Anirudh"           # functions are objects with attributes
print(greet.author)
```

OUTPUT

```
<class 'int'> <class 'str'> <class 'builtin_function_or_method'> <class 'type'>
3 HI
Anirudh
```

Q38. What is dynamic binding (late binding)?

FRESHERS · POLYMORPHISM

Binding links a method call to the code that will run. With **static (early) binding**, the compiler decides from the declared type. With **dynamic (late) binding**, the decision is made at runtime from the object's actual type.

Dynamic binding is what makes method overriding useful: a variable of type `Shape` can hold a `Circle`, and `shape.area()` calls the circle's version.

- **Java:** instance methods use dynamic binding; `static`, `private` and `final` methods are bound statically.
- **C++:** only methods marked `virtual` use dynamic binding.
- **Python:** every method call is resolved at runtime.

Q39. Can you create an object of an abstract class? Can an abstract class have a constructor?

FRESHERS · ABSTRACTION

No, you cannot instantiate an abstract class directly. Java gives a compile error, C++ refuses a class with pure virtual functions, and Python raises `TypeError` when a class still has unimplemented abstract methods.

Yes, an abstract class can have a constructor. It runs when a concrete subclass is created, through `super().__init__()`, to set up the shared state.

PYTHON (TESTED)

```
from abc import ABC, abstractmethod

class Employee(ABC):
    def __init__(self, name):
        self.name = name
        print("Employee constructor ran for", name)
    @abstractmethod
    def salary(self): ...

class Intern(Employee):
    def salary(self): return 15000

try:
    Employee("Asha")
except TypeError as e:
    print("TypeError:", e)
i = Intern("Rohan")
print(i.name, i.salary())
```

OUTPUT

```
TypeError: Can't instantiate abstract class Employee without an implementation for abstract method 'salary'
Employee constructor ran for Rohan
Rohan 15000
```

Q40. Design a simple class for a real-world object, such as a library book.

FRESHERS · DESIGN

Interviewers ask this to check that you can choose sensible attributes and methods and protect the object's rules. Talk through it: identify the data, the actions, the rules, and how other code will use it.

For a library book: data is title, author, ISBN and whether it is issued; actions are issue and return; rules are "cannot issue an issued book" and "cannot return a book that is not issued".

PYTHON (TESTED)

```
class Book:
    def __init__(self, title, author, isbn):
        self.title = title
        self.author = author
        self.isbn = isbn
        self._issued_to = None

    @property
    def is_available(self):
        return self._issued_to is None

    def issue(self, member):
        if not self.is_available:
            raise ValueError(f"'{self.title}' is already issued to {self._issued_to}")
        self._issued_to = member

    def give_back(self):
        if self.is_available:
            raise ValueError(f"'{self.title}' was not issued")
        self._issued_to = None

    def __repr__(self):
        status = "available" if self.is_available else f"issued to {self._issued_to}"
        return f"<Book {self.title!r}, {status}>"

b = Book("Clean Code", "Robert C. Martin", "9780132350884")
b.issue("Asha")
print(b)
try:
    b.issue("Rohan")
except ValueError as e:
    print("Error:", e)
b.give_back()
print(b)
```

OUTPUT

```
<Book 'Clean Code', issued to Asha>
Error: 'Clean Code' is already issued to Asha
<Book 'Clean Code', available>
```

Intermediate level

Multiple inheritance and the MRO, magic methods, descriptors, dataclasses, protocols, the SOLID principles, composition and language differences.

Q41. What is the diamond problem? How does Python solve it with the MRO?

INTERMEDIATE · INHERITANCE

The diamond problem appears with multiple inheritance: classes B and C both inherit from A, and D inherits from both B and C. If B and C both override a method from A, which version does D get? And is A's part of the object set up once or twice?

Python solves it with the **Method Resolution Order (MRO)**, computed by the **C3 linearisation** algorithm. It produces one fixed order in which classes are searched: the child first, then parents left to right, and a shared base only after all of its children. You can read it from `ClassName.__mro__` or `ClassName.mro()`. The MRO explorer on this page lets you try different hierarchies.

Java note: Java avoids the problem for classes by banning multiple class inheritance. For interface default methods with the same signature, the class must override and choose. **C++** needs `virtual` inheritance so that A's part is shared.

PYTHON (TESTED)

```
class A:
    def hello(self): return "A"
class B(A):
    def hello(self): return "B"
class C(A):
    def hello(self): return "C"
class D(B, C):
    pass

print(D().hello())
print([cls.__name__ for cls in D.__mro__])
```

OUTPUT

```
B
['D', 'B', 'C', 'A', 'object']
```

Q42. How does super() work with multiple inheritance?

INTERMEDIATE · INHERITANCE

`super()` does not simply mean "my parent". It means **"the next class after me in the MRO of the object's actual class"**. When every class in a hierarchy calls `super()`, each method in the chain runs exactly once, in MRO order. This is called **cooperative multiple inheritance**.

That is why, in the example, B's `super()` call goes to C, not to A, even though B only inherits from A. To make it work, every class should call `super()` and accept extra keyword arguments with `**kwargs` when constructors differ.

PYTHON (TESTED)

```
class A:
    def setup(self):
        print("A.setup")
class B(A):
    def setup(self):
        print("B.setup"); super().setup()
class C(A):
    def setup(self):
        print("C.setup"); super().setup()
class D(B, C):
    def setup(self):
        print("D.setup"); super().setup()

D().setup()          # A runs once, at the end
```

OUTPUT

```
D.setup
B.setup
C.setup
A.setup
```

Q43. What is a mixin?

INTERMEDIATE · INHERITANCE

A mixin is a small class that provides one specific piece of behaviour to be "mixed into" other classes through multiple inheritance. It is not meant to be used on its own and usually has no `__init__` of its own.

Examples: a `JsonMixin` that adds `to_json()`, a `TimestampMixin` that adds created and updated times, or Django's `LoginRequiredMixin` for views. Mixins let you share behaviour across unrelated classes without building a deep hierarchy. By convention they are listed before the main base class.

PYTHON (TESTED)

```
import json

class JsonMixin:
    def to_json(self):
        return json.dumps(self.__dict__)

class ReprMixin:
    def __repr__(self):
        fields = ", ".join(f"{k}={v!r}" for k, v in self.__dict__.items())
        return f"{type(self).__name__}({fields})"

class Student(JsonMixin, ReprMixin):
    def __init__(self, name, batch):
        self.name, self.batch = name, batch

s = Student("Asha", "DSA-Nov")
print(s)
print(s.to_json())
```

OUTPUT

```
Student(name='Asha', batch='DSA-Nov')
{"name": "Asha", "batch": "DSA-Nov"}
```

Q44. What are magic (dunder) methods? Name the important ones.

INTERMEDIATE · PYTHON SPECIFICS

Magic methods (double underscore, "dunder") let your objects work with Python's built-in syntax and functions. Python calls them for you.

You write	Python calls
`Cls(...)`	`__new__`, then `__init__`
`str(x)`, `repr(x)`	`__str__`, `__repr__`
`len(x)`, `x[i]`, `v in x`	`__len__`, `__getitem__`, `__contains__`
`for v in x`	`__iter__`, `__next__`
`x == y`, `x < y`, `hash(x)`	`__eq__`, `__lt__`, `__hash__`
`x + y`, `x * y`	`__add__`, `__mul__`
`with x:`	`__enter__`, `__exit__`
`x()`	`__call__`

PYTHON (TESTED)

```
class Playlist:
    def __init__(self, songs): self.songs = list(songs)
    def __len__(self): return len(self.songs)
    def __getitem__(self, i): return self.songs[i]
    def __contains__(self, song): return song in self.songs

p = Playlist(["Intro", "Arrays", "Strings"])
print(len(p), p[1], "Arrays" in p)
print([s.upper() for s in p])        # __getitem__ alone makes it iterable
```

OUTPUT

```
3 Arrays True
['INTRO', 'ARRAYS', 'STRINGS']
```

Q45. Why must __eq__ and __hash__ be consistent?

INTERMEDIATE · PYTHON SPECIFICS

Sets and dictionaries find objects by their **hash**, then confirm with **equality**. The rule is: **if** `a == b`**, then** `hash(a) == hash(b)`. If you break it, equal objects end up as duplicates in a set or cannot be found as dict keys.

Python helps: when you define `__eq__` without `__hash__`, the class becomes **unhashable**. If the object is immutable, define `__hash__` from the same fields used in `__eq__`. Never hash mutable fields that can change while the object is inside a set.

Java note: the same contract applies to `equals()` and `hashCode()`.

PYTHON (TESTED)

```
class P1:
    def __init__(self, x): self.x = x
    def __eq__(self, o): return self.x == o.x

try:
    {P1(1)}
except TypeError as e:
    print("TypeError:", e)

class P2:
    def __init__(self, x): self.x = x
    def __eq__(self, o): return self.x == o.x
    def __hash__(self): return hash(self.x)

print(len({P2(1), P2(1), P2(2)}))
```

OUTPUT

```
TypeError: unhashable type: 'P1'
2
```

Q46. What is the difference between __new__ and __init__?

INTERMEDIATE · PYTHON SPECIFICS

- `__new__(cls, ...)` is a static method that **creates** and returns the new object. It runs first.
- `__init__(self, ...)` **initialises** the object that `__new__` returned. It must return `None`.

You rarely override `__new__`. The main reasons are subclassing immutable types (`int`, `str`, `tuple`), where the value must be set at creation, and controlling instance creation, as in a singleton or an instance cache.

PYTHON (TESTED)

```
class Celsius(float):
    def __new__(cls, value):
        print("__new__ creates the object")
        return super().__new__(cls, round(value, 1))
    def __init__(self, value):
        print("__init__ sets it up")
        self.unit = "C"

t = Celsius(36.6666)
print(t, t.unit, isinstance(t, float))
```

OUTPUT

```
__new__ creates the object
__init__ sets it up
36.7 C True
```

Q47. What is the difference between a shallow copy and a deep copy of an object?

INTERMEDIATE · PYTHON SPECIFICS

- **Assignment** (`b = a`) copies nothing; both names point to the same object.
- **Shallow copy** (`copy.copy`) creates a new outer object, but its attributes still point to the same inner objects. Changing a nested list affects both.
- **Deep copy** (`copy.deepcopy`) copies the object and, recursively, everything it contains.

You can customise both with `__copy__` and `__deepcopy__`. **Java note:** `clone()` is shallow by default; deep copies need copy constructors or serialisation. **C++:** the default copy constructor is shallow for pointer members.

PYTHON (TESTED)

```
import copy

class Batch:
    def __init__(self, name, students):
        self.name, self.students = name, students

orig = Batch("Morning", ["Asha", "Rohan"])
shallow = copy.copy(orig)
deep = copy.deepcopy(orig)

orig.students.append("Meera")
print("shallow:", shallow.students)   # shares the list
print("deep   :", deep.students)      # has its own list
```

OUTPUT

```
shallow: ['Asha', 'Rohan', 'Meera']
deep   : ['Asha', 'Rohan']
```

Q48. What are descriptors?

INTERMEDIATE · PYTHON SPECIFICS

A descriptor is an object that defines `__get__`, `__set__` or `__delete__` and is stored as a **class attribute**. When you access that attribute on an instance, Python calls the descriptor's methods instead of reading the value directly.

Descriptors are the machinery behind `@property`, methods (which is how `self` gets bound), `@classmethod`, `@staticmethod`, and Django model fields. Write your own when the same validation logic is needed on many attributes, so you do not repeat a property for each one.

PYTHON (TESTED)

```
class Positive:
    def __set_name__(self, owner, name):
        self.name = "_" + name
    def __get__(self, obj, objtype=None):
        return getattr(obj, self.name)
    def __set__(self, obj, value):
        if value <= 0:
            raise ValueError(f"{self.name[1:]} must be positive")
        setattr(obj, self.name, value)

class Product:
    price = Positive()          # one descriptor, reused
    stock = Positive()
    def __init__(self, price, stock):
        self.price, self.stock = price, stock

p = Product(499, 10)
print(p.price, p.stock)
try:
    p.stock = 0
except ValueError as e:
    print("Error:", e)
```

OUTPUT

```
499 10
Error: stock must be positive
```

Q49. What is __slots__ and when would you use it?

INTERMEDIATE · PYTHON SPECIFICS

By default, every Python object stores its attributes in a per-instance dictionary, `__dict__`. Declaring `__slots__` tells Python to store a fixed set of attributes in a compact array instead.

- **Benefits:** much less memory per object and slightly faster attribute access. Useful when you create millions of small objects.
- **Trade-off:** you cannot add attributes that are not listed, and multiple inheritance with slots needs care.

PYTHON (TESTED)

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

class PointSlots:
    __slots__ = ("x", "y")
    def __init__(self, x, y): self.x, self.y = x, y

a, b = PointDict(1, 2), PointSlots(1, 2)
print("has __dict__:", hasattr(a, "__dict__"), hasattr(b, "__dict__"))
try:
    b.z = 3
except AttributeError as e:
    print("AttributeError:", e)
```

OUTPUT

```
has __dict__: True False
AttributeError: 'PointSlots' object has no attribute 'z' and no __dict__ for setting new attributes
```

Q50. What are dataclasses? When should you use them?

INTERMEDIATE · PYTHON SPECIFICS

`@dataclass` (Python 3.7+) writes the boring parts of a class for you from type-annotated fields: `__init__`, `__repr__` and `__eq__`, and optionally ordering, immutability (`frozen=True`) and `__slots__`.

Use them for classes that mainly hold data: DTOs, configuration, API responses, value objects. Use `field(default_factory=list)` for mutable defaults, which avoids the shared-list bug. For heavy validation, libraries such as Pydantic go further.

Java note: Java 16+ has `record` for the same purpose.

PYTHON (TESTED)

```
from dataclasses import dataclass, field

@dataclass(order=True)
class Course:
    price: int
    title: str
    tags: list = field(default_factory=list)

a = Course(4999, "Master DSA", ["dsa"])
b = Course(1999, "Python Basics")
print(a)
print(a == Course(4999, "Master DSA", ["dsa"]))
print(sorted([a, b])[0].title)
```

OUTPUT

```
Course(price=4999, title='Master DSA', tags=['dsa'])
True
Python Basics
```

Q51. How do you make your own class iterable?

INTERMEDIATE · PYTHON SPECIFICS

An object is **iterable** if `__iter__` returns an **iterator**. An iterator has `__next__`, which returns the next value or raises `StopIteration` when done.

The simplest way is to write `__iter__` as a generator with `yield`: Python builds the iterator for you, and the object can be looped over many times.

PYTHON (TESTED)

```
class Countdown:
    def __init__(self, start):
        self.start = start
    def __iter__(self):             # a generator: returns a fresh iterator each time
        n = self.start
        while n > 0:
            yield n
            n -= 1

c = Countdown(3)
print(list(c), list(c))
it = iter(c)
print(next(it), next(it))
```

OUTPUT

```
[3, 2, 1] [3, 2, 1]
3 2
```

Q52. How do you write a class that works with the with statement?

INTERMEDIATE · PYTHON SPECIFICS

A **context manager** defines `__enter__`, which runs at the start of the `with` block, and `__exit__`, which always runs at the end, even if an exception is raised. It is the Pythonic way to manage resources: files, locks, database transactions and network connections.

If `__exit__` returns `True`, the exception is suppressed; normally it returns `False` or `None` so errors propagate. **Java note:** the equivalent is `try-with-resources` with `AutoCloseable`.

PYTHON (TESTED)

```
class Transaction:
    def __enter__(self):
        print("BEGIN")
        return self
    def __exit__(self, exc_type, exc, tb):
        print("ROLLBACK" if exc_type else "COMMIT")
        return False                    # do not hide errors

with Transaction():
    print("update balance")

try:
    with Transaction():
        raise ValueError("insufficient balance")
except ValueError as e:
    print("Caught:", e)
```

OUTPUT

```
BEGIN
update balance
COMMIT
BEGIN
ROLLBACK
Caught: insufficient balance
```

Q53. What is a callable object (__call__)?

INTERMEDIATE · PYTHON SPECIFICS

If a class defines `__call__`, its instances can be called like functions: `obj(args)`. This is useful when you need a function that also keeps state or configuration, such as a rate limiter, a counter, a configured validator, or a machine learning model (PyTorch modules are callable for this reason).

PYTHON (TESTED)

```
class Multiplier:
    def __init__(self, factor):
        self.factor = factor
        self.calls = 0
    def __call__(self, x):
        self.calls += 1
        return x * self.factor

double = Multiplier(2)
print(double(5), double(21), callable(double), double.calls)
```

OUTPUT

```
10 42 True 2
```

Q54. What is a Protocol (structural typing) in Python?

INTERMEDIATE · ABSTRACTION

An abstract base class requires classes to **inherit** from it (nominal typing). A `typing.Protocol` describes the methods an object must have, and any class with those methods matches, **without inheriting** (structural typing). It is duck typing that type checkers such as mypy can verify.

With `@runtime_checkable`, you can also use the protocol in `isinstance` checks (which only check that the methods exist).

PYTHON (TESTED)

```
from typing import Protocol, runtime_checkable

@runtime_checkable
class Payable(Protocol):
    def pay(self, amount: int) -> str: ...

class Upi:                                  # no inheritance
    def pay(self, amount): return f"UPI paid Rs {amount}"

class Card:
    def pay(self, amount): return f"Card paid Rs {amount}"

def checkout(method: Payable, amount: int):
    return method.pay(amount)

print(checkout(Upi(), 499), "|", checkout(Card(), 999))
print(isinstance(Upi(), Payable), isinstance("text", Payable))
```

OUTPUT

```
UPI paid Rs 499 | Card paid Rs 999
True False
```

Q55. How do you create custom exceptions? Why use a class hierarchy for them?

INTERMEDIATE · PYTHON SPECIFICS

Custom exceptions are classes that inherit from `Exception` (never directly from `BaseException`). A small hierarchy lets callers catch errors at the right level: one specific error, or every error from your module with the base class.

Exceptions can carry extra data as attributes, which is better than parsing the message text.

PYTHON (TESTED)

```
class PaymentError(Exception):
    pass

class InsufficientFunds(PaymentError):
    def __init__(self, needed, available):
        super().__init__(f"Need Rs {needed}, have Rs {available}")
        self.shortfall = needed - available

class CardDeclined(PaymentError):
    pass

for err in [InsufficientFunds(5000, 3200), CardDeclined("Bank declined")]:
    try:
        raise err
    except InsufficientFunds as e:
        print("Funds:", e, "| short by", e.shortfall)
    except PaymentError as e:
        print("Other payment error:", e)
```

OUTPUT

```
Funds: Need Rs 5000, have Rs 3200 | short by 1800
Other payment error: Bank declined
```

Q56. What are the SOLID principles?

INTERMEDIATE · SOLID

SOLID is a set of five design principles, popularised by Robert C. Martin, for classes that are easy to change and test:

- **S, Single Responsibility:** a class should have one reason to change.
- **O, Open/Closed:** open for extension, closed for modification. Add behaviour with new code, not by editing working code.
- **L, Liskov Substitution:** a subclass must be usable anywhere its parent is expected, without surprises.
- **I, Interface Segregation:** many small, focused interfaces are better than one large one.
- **D, Dependency Inversion:** depend on abstractions, not concrete classes. High-level code should not create its low-level dependencies.

The SOLID refactoring tool on this page applies each principle to one messy class, step by step.

Q57. Explain the Single Responsibility Principle with an example.

INTERMEDIATE · SOLID

A class should have **one reason to change**, meaning one job owned by one part of the business. A `Report` class that calculates figures, formats them as PDF and emails them has three reasons to change: the finance team's rules, the design team's layout and the IT team's email server.

Split it so each class has one job. Each part can then change and be tested on its own.

PYTHON (TESTED)

```
class SalesReport:                      # only the numbers
    def __init__(self, sales): self.sales = sales
    def total(self): return sum(self.sales)

class TextFormatter:                    # only the presentation
    def format(self, report): return f"Total sales: Rs {report.total():,}"

class ConsoleSender:                    # only the delivery
    def send(self, text): print("Sending ->", text)

report = SalesReport([12000, 8500, 4300])
ConsoleSender().send(TextFormatter().format(report))
```

OUTPUT

```
Sending -> Total sales: Rs 24,800
```

Q58. Explain the Open/Closed Principle with an example.

INTERMEDIATE · SOLID

Software should be **open for extension but closed for modification**. When a new requirement arrives, you should be able to add a new class instead of editing tested code.

The warning sign is a long `if/elif` on a type: every new payment method or shape means editing the same function. Replace it with polymorphism, so adding a type means adding a class.

PYTHON (TESTED)

```
from abc import ABC, abstractmethod

class Discount(ABC):
    @abstractmethod
    def apply(self, price): ...

class NoDiscount(Discount):
    def apply(self, price): return price
class StudentDiscount(Discount):
    def apply(self, price): return round(price * 0.8)
class FestivalDiscount(Discount):          # new rule = new class, nothing edited
    def apply(self, price): return price - 500

def checkout(price, discount: Discount):
    return discount.apply(price)

for d in [NoDiscount(), StudentDiscount(), FestivalDiscount()]:
    print(type(d).__name__, checkout(4999, d))
```

OUTPUT

```
NoDiscount 4999
StudentDiscount 3999
FestivalDiscount 4499
```

Q59. Explain the Liskov Substitution Principle. Why is Square-inherits-Rectangle a problem?

INTERMEDIATE · SOLID

The Liskov Substitution Principle says: **objects of a subclass must be usable wherever the parent class is expected, without breaking the program**. A subclass may add behaviour, but it must keep the parent's promises.

The classic violation: in geometry a square is a rectangle, so `Square(Rectangle)` seems natural. But a rectangle promises that setting the width does not change the height. A square must keep both equal, so it breaks that promise, and code written for rectangles gives wrong answers.

Fixes: do not inherit; make both implement a common `Shape` with `area()`, or make them immutable.

PYTHON (TESTED)

```
class Rectangle:
    def __init__(self, w, h): self.w, self.h = w, h
    def set_width(self, w): self.w = w
    def area(self): return self.w * self.h

class Square(Rectangle):
    def __init__(self, side): super().__init__(side, side)
    def set_width(self, w): self.w = self.h = w      # keeps it square

def stretch(rect):                      # written for Rectangle
    rect.set_width(10)
    return rect.area()                  # expects 10 * original height

print("Rectangle 5x4 ->", stretch(Rectangle(5, 4)), "(expected 40)")
print("Square 4     ->", stretch(Square(4)), "(expected 40)")
```

OUTPUT

```
Rectangle 5x4 -> 40 (expected 40)
Square 4     -> 100 (expected 40)
```

Q60. Explain the Interface Segregation Principle.

INTERMEDIATE · SOLID

Clients should not be forced to depend on methods they do not use. A large interface forces every implementer to provide everything, often with empty methods or `raise NotImplementedError`, which is a sign that the abstraction is wrong.

Example: a `Machine` interface with `print()`, `scan()` and `fax()`. A basic printer must now pretend to scan and fax. Split it into `Printer`, `Scanner` and `Fax` interfaces; a multifunction device implements all three, and a basic printer implements only `Printer`.

In Python, small `Protocol` classes make this natural.

Q61. Explain the Dependency Inversion Principle and dependency injection.

INTERMEDIATE · SOLID

High-level modules should not depend on low-level modules; both should depend on abstractions. An `OrderService` should not create a `MySQLDatabase` inside itself. It should depend on a `Repository` abstraction, and receive a concrete one from outside.

Dependency injection is the technique: pass dependencies in, usually through the constructor. Benefits: you can swap MySQL for PostgreSQL, and in tests you can pass a fake repository with no real database. FastAPI's `Depends` and Spring's `@Autowired` are dependency injection

frameworks.

PYTHON (TESTED)

```
class OrderService:
    def __init__(self, repo):                   # dependency injected
        self.repo = repo
    def place(self, item):
        self.repo.save(item)
        return f"Order placed: {item}"

class SqlRepo:
    def save(self, item): print("INSERT INTO orders ...", item)

class FakeRepo:                                 # for tests
    def __init__(self): self.saved = []
    def save(self, item): self.saved.append(item)

print(OrderService(SqlRepo()).place("Python course"))
fake = FakeRepo()
OrderService(fake).place("DSA course")
print("Test saw:", fake.saved)
```

OUTPUT

```
INSERT INTO orders ... Python course
Order placed: Python course
Test saw: ['DSA course']
```

Q62. Why is composition usually preferred over inheritance?

INTERMEDIATE · RELATIONSHIPS

- **Inheritance is fixed and tight.** A child depends on the parent's internals, and a change in the parent can break every child (the "fragile base class" problem).
- **Inheritance hierarchies explode.** Combining features through subclasses gives classes such as `LoggedCachedRetryingHttpClient`. With composition, you combine small parts.
- **Composition is flexible.** Parts can be swapped at runtime and tested alone.
- **Inheritance exposes everything.** A child inherits every public method, even ones that make no sense for it.

Use inheritance when there is a true "is-a" relationship that passes the Liskov test, and the parent was designed to be extended. Otherwise, compose.

PYTHON (TESTED)

```
class Logger:
    def log(self, msg): print("[log]", msg)
class Cache:
    def __init__(self): self.store = {}

class HttpClient:                               # built from parts, not from parents
    def __init__(self, logger=None, cache=None):
        self.logger, self.cache = logger, cache
    def get(self, url):
        if self.cache and url in self.cache.store:
            return self.cache.store[url] + " (cached)"
        if self.logger: self.logger.log(f"GET {url}")
        body = f"<page {url}>"
        if self.cache: self.cache.store[url] = body
        return body

c = HttpClient(logger=Logger(), cache=Cache())
print(c.get("/blog")); print(c.get("/blog"))
```

OUTPUT

```
[log] GET /blog
<page /blog>
<page /blog> (cached)
```

Q63. What is the difference between tight coupling and loose coupling?

INTERMEDIATE · DESIGN

Tight coupling: a class knows the concrete details of another, for example it creates it with `EmailSender()` inside a method, or reaches into its attributes. Changing or replacing one forces changes in the other, and testing needs the real dependency.

Loose coupling: a class depends only on a small interface and receives the dependency from outside. You can replace the implementation (email with SMS, real with fake) without touching the class.

Techniques: dependency injection, programming to interfaces, events or callbacks instead of direct calls, and the Law of Demeter ("talk only to your immediate friends": avoid chains like `order.customer.address.city.name`).

Q64. What are virtual functions? What is a vtable?

INTERMEDIATE · POLYMORPHISM

In C++, a **virtual function** is a member function that can be overridden and is resolved at runtime through a base-class pointer or reference. Without `virtual`, C++ calls the base version based on the pointer's declared type. A **pure virtual function** (`= 0`) has no implementation and makes the class abstract.

Compilers usually implement this with a **vtable**: each class with virtual functions has a table of function pointers, and each object has a hidden pointer (vptr) to its class's table. A virtual call looks up the right function in the table. Base classes used polymorphically also need a **virtual destructor**, or deleting through a base pointer leaks the derived part.

Java and Python note: all instance methods are effectively virtual by default; Java uses `final` to prevent overriding.

Q65. How do you prevent a class from being inherited or a method from being overridden?

INTERMEDIATE · INHERITANCE

- **Java:** a `final` class cannot be extended (`String` is final); a `final` method cannot be overridden. Java 17 adds `sealed` classes that list which classes may extend them.
- **C++:** the `final` specifier on a class or virtual function.
- **Python:** nothing is enforced at runtime by default. `typing.final` marks intent and type checkers report violations. To enforce it, raise an error in `__init_subclass__`.

PYTHON (TESTED)

```
class Config:
    def __init_subclass__(cls, **kwargs):
        raise TypeError(f"{cls.__name__} cannot inherit from final class Config")

try:
    class MyConfig(Config):
        pass
except TypeError as e:
    print("TypeError:", e)
```

OUTPUT

```
TypeError: MyConfig cannot inherit from final class Config
```

Q66. What is the object class?

INTERMEDIATE · INHERITANCE

In Python 3, every class implicitly inherits from `object`, the root of the class hierarchy. That is where default behaviour comes from: `__init__`, `__repr__` (which prints the memory address), `__eq__` (which compares identity) and `__hash__`.

Java note: every class extends `java.lang.Object`, which provides `toString()`, `equals()`, `hashCode()`, `getClass()` and others. **C++** has no universal root class.

PYTHON (TESTED)

```
class Empty:
    pass

print(Empty.__mro__)
print(Empty.__bases__, issubclass(Empty, object))
print(sorted(m for m in dir(object) if m in ("__eq__", "__hash__", "__init__", "__repr__", "__str__")))
```

OUTPUT

```
(<class '__main__.Empty'>, <class 'object'>)
(<class 'object'>,) True
['__eq__', '__hash__', '__init__', '__repr__', '__str__']
```

Q67. What is an immutable object? How do you make one in Python?

INTERMEDIATE · DESIGN

An immutable object cannot be changed after it is created. Python's `int`, `str`, `tuple` and `frozenset` are immutable. "Changing" one actually creates a new object.

Benefits: safe to share between threads, safe to use as dictionary keys or in sets, easier to reason about because nothing changes behind your back. **Cost:** a new object for every change.

For your own classes, use `@dataclass(frozen=True)` or `typing.NamedTuple`. **Java note:** make the class `final`, fields `private final`, no setters, and copy mutable inputs; or use a `record`.

PYTHON (TESTED)

```
from dataclasses import dataclass, replace, FrozenInstanceError

@dataclass(frozen=True)
class Money:
    amount: int
    currency: str = "INR"

price = Money(4999)
try:
    price.amount = 1
except FrozenInstanceError as e:
    print("FrozenInstanceError:", e)
discounted = replace(price, amount=3999)    # new object
print(price, discounted, hash(price) == hash(Money(4999)))
```

OUTPUT

```
FrozenInstanceError: cannot assign to field 'amount'
Money(amount=4999, currency='INR') Money(amount=3999, currency='INR') True
```

Q68. What are nested (inner) classes? When are they useful?

INTERMEDIATE · CLASSES AND OBJECTS

A nested class is a class defined inside another class. In Python it is simply a class attribute of the outer class; it has no special access to the outer instance.

Use it to group a helper class that only makes sense with its outer class, such as a `Node` inside a `LinkedList`, or Django's `class Meta` inside a model.

Java note: Java distinguishes **static nested classes** (no link to an outer object) from **inner classes** (each instance is tied to an outer instance and can access its fields), plus local and anonymous classes.

PYTHON (TESTED)

```
class LinkedList:
    class Node:
        def __init__(self, val, nxt=None):
            self.val, self.next = val, nxt

    def __init__(self, *vals):
        self.head = None
        for v in reversed(vals):
            self.head = LinkedList.Node(v, self.head)

    def __iter__(self):
        node = self.head
        while node:
            yield node.val
            node = node.next

print(list(LinkedList(1, 2, 3)))
```

OUTPUT

```
[1, 2, 3]
```

Q69. What is monkey patching?

INTERMEDIATE · PYTHON SPECIFICS

Monkey patching means changing a class or module at runtime: adding, replacing or removing methods and attributes after it has been defined. Python allows it because classes are just objects.

It is used in tests to replace slow or external calls (`unittest.mock.patch` does this safely and undoes it), and for urgent fixes to third-party libraries. In normal code it is risky: behaviour changes far from where the class is defined, which makes bugs hard to find.

PYTHON (TESTED)

```
class PaymentGateway:
    def charge(self, amount):
        return f"Charged Rs {amount} via the real bank"

gw = PaymentGateway()
print(gw.charge(100))

original = PaymentGateway.charge
PaymentGateway.charge = lambda self, amount: f"FAKE charge of Rs {amount}"
print(gw.charge(100))                   # existing objects change too
PaymentGateway.charge = original        # always undo
```

OUTPUT

```
Charged Rs 100 via the real bank
FAKE charge of Rs 100
```

Q70. How do you make objects comparable and sortable?

INTERMEDIATE · PYTHON SPECIFICS

Define the rich comparison methods: `__eq__`, `__lt__`, `__le__`, `__gt__`, `__ge__`. `sorted()` and `min()` only need `__lt__`. To avoid writing all of them, define `__eq__` and one ordering method and add `@functools.total_ordering`.

Alternatives: sort with `key=` (often simplest), or `@dataclass(order=True)`, which compares fields in order. **Java note:** implement `Comparable` with `compareTo()`, or pass a `Comparator`.

PYTHON (TESTED)

```
from functools import total_ordering

@total_ordering
class Version:
    def __init__(self, text):
        self.parts = tuple(int(p) for p in text.split("."))
    def __eq__(self, other): return self.parts == other.parts
    def __lt__(self, other): return self.parts < other.parts
    def __repr__(self): return ".".join(map(str, self.parts))

vs = [Version("3.10"), Version("3.9"), Version("3.12.1")]
print(sorted(vs), max(vs), Version("3.9") >= Version("3.10"))
```

OUTPUT

```
[3.9, 3.10, 3.12.1] 3.12.1 False
```

Q71. What is object slicing in C++?

INTERMEDIATE · POLYMORPHISM

Object slicing happens in C++ when a derived object is copied **by value** into a base-class variable. Only the base part is copied; the derived fields are "sliced off", and the copy behaves as a base object, so overridden virtual functions are no longer called.

Example: `Dog d; Animal a = d; a.speak();` calls `Animal::speak`. Passing objects by value to a function that takes `Animal` causes the same problem.

Fix: use pointers or references (`Animal&`, `Animal*`, smart pointers) for polymorphism. Java and Python do not have this problem, because variables hold references to objects, never copies of them.

Q72. What are generics? How do you write a generic class in Python?

INTERMEDIATE · PYTHON SPECIFICS

Generics let you write a class or function once and use it with different types while keeping type safety. A `Stack` of integers and a `Stack` of strings share the same code, and the type checker knows which is which.

In Python, types are only checked by tools such as mypy, so generics are type hints: `TypeVar` with `Generic`, or the newer `class Stack[T]:` syntax in Python 3.12. **Java** uses `class Stack<T>` (with type erasure at runtime), and **C++** uses templates, which generate separate code for each type.

PYTHON (TESTED)

```
from typing import Generic, TypeVar

T = TypeVar("T")

class Stack(Generic[T]):
    def __init__(self) -> None:
        self._items: list[T] = []
    def push(self, item: T) -> None:
        self._items.append(item)
    def pop(self) -> T:
        return self._items.pop()
    def __len__(self) -> int:
        return len(self._items)

s: Stack[int] = Stack()
s.push(1); s.push(2)
print(s.pop(), len(s))
```

OUTPUT

```
2 1
```

Q73. What is the difference between __getattr__ and __getattribute__?

INTERMEDIATE · PYTHON SPECIFICS

- `__getattribute__` runs for **every** attribute access. Overriding it is powerful but easy to get wrong: inside it you must call `super().__getattribute__`, or you will recurse forever.
- `__getattr__` runs **only when normal lookup fails**, as a fallback. It is the safe choice for proxies, lazy loading, and objects that forward calls to another object.

PYTHON (TESTED)

```
class ApiResponse:
    def __init__(self, data):
        self._data = data
    def __getattr__(self, name):          # only called for missing attributes
        try:
            return self._data[name]
        except KeyError:
            raise AttributeError(f"no field {name!r}") from None

r = ApiResponse({"user": "asha", "plan": "pro"})
print(r.user, r.plan)
try:
    r.email
except AttributeError as e:
    print("AttributeError:", e)
```

OUTPUT

```
asha pro
AttributeError: no field 'email'
```

Q74. What is method chaining (a fluent interface)?

INTERMEDIATE · DESIGN

Method chaining is a style where each method returns `self` (or a new object), so calls can be written one after another: `query.where(...).order_by(...).limit(10)`. It reads like a sentence and is common in builders, query libraries (Django ORM, SQLAlchemy), Pandas and jQuery.

Mutable builders return `self`. Immutable APIs (Django querysets, Pandas methods) return a new object each time, which is safer because the original is never changed.

PYTHON (TESTED)

```
class Query:
    def __init__(self, table):
        self.parts = [f"SELECT * FROM {table}"]
    def where(self, cond):
        self.parts.append(f"WHERE {cond}"); return self
    def order_by(self, col):
        self.parts.append(f"ORDER BY {col}"); return self
    def limit(self, n):
        self.parts.append(f"LIMIT {n}"); return self
    def sql(self):
        return " ".join(self.parts)

print(Query("students").where("marks > 80").order_by("marks DESC").limit(5).sql())
```

OUTPUT

```
SELECT * FROM students WHERE marks > 80 ORDER BY marks DESC LIMIT 5
```

Q75. What is an Enum and why use it instead of plain constants?

INTERMEDIATE · PYTHON SPECIFICS

An `Enum` is a class whose members are a fixed set of named constants, such as order statuses or user roles. Compared with plain strings or integers:

- Typos fail immediately (`Status.SHIPED` raises an error) instead of silently being a wrong string.
- Members are unique, comparable by identity, iterable and readable when printed.
- The set of valid values is defined in one place.

Java note: Java enums are full classes too and can have fields, constructors and methods.

PYTHON (TESTED)

```
from enum import Enum

class OrderStatus(Enum):
    PLACED = "placed"
    SHIPPED = "shipped"
    DELIVERED = "delivered"

s = OrderStatus("shipped")              # from a database value
print(s, s.name, s.value, s is OrderStatus.SHIPPED)
print([m.name for m in OrderStatus])
try:
    OrderStatus("lost")
except ValueError as e:
    print("ValueError:", e)
```

OUTPUT

```
OrderStatus.SHIPPED SHIPPED shipped True
['PLACED', 'SHIPPED', 'DELIVERED']
ValueError: 'lost' is not a valid OrderStatus
```

Experienced level

Design patterns with code, metaclasses, low-level design problems, thread safety, testing with mocks and refactoring.

Q76. What are design patterns? What are their main categories?

EXPERIENCED · DESIGN PATTERNS

Design patterns are proven, reusable solutions to common design problems in object-oriented code. They were catalogued in the 1994 "Gang of Four" book, which describes 23 patterns in three groups:

Category	Concerned with	Examples
Creational	How objects are created	Singleton, Factory Method, Abstract Factory, Builder, Prototype
Structural	How classes and objects are combined	Adapter, Decorator, Facade, Proxy, Composite
Behavioural	How objects communicate and share responsibility	Strategy, Observer, Command, State, Template Method, Iterator

Patterns are a shared vocabulary, not rules. In Python, several are simpler than in Java, because functions are objects and modules already behave like singletons.

Q77. What is the Singleton pattern? How do you implement it in Python, and what are its problems?

EXPERIENCED · DESIGN PATTERNS

Singleton ensures a class has **only one instance** and gives a global access point to it, for example one configuration object or one connection pool.

In Python the simplest singleton is a **module**: it is imported once, so a module-level object is shared everywhere. If you need a class, override `__new__`, and add a lock if threads might create it at the same time.

Problems: it is global state in disguise, it hides dependencies, it makes testing hard because state leaks between tests, and it needs care with threads. Prefer creating one instance at startup and passing it in (dependency injection).

Java note: the usual safe forms are an `enum` singleton, or a private constructor with a static holder class.

PYTHON (TESTED)

```
import threading

class Config:
    _instance = None
    _lock = threading.Lock()

    def __new__(cls):
        if cls._instance is None:
            with cls._lock:                      # thread-safe creation
                if cls._instance is None:
                    cls._instance = super().__new__(cls)
                    cls._instance.settings = {}
        return cls._instance

a = Config(); b = Config()
a.settings["env"] = "prod"
print(a is b, b.settings)
```

OUTPUT

```
True {'env': 'prod'}
```

Q78. What is the Factory pattern?

EXPERIENCED · DESIGN PATTERNS

A factory hides **which concrete class** gets created. Callers ask for "a notifier for this channel" and receive an object that follows a common interface, without knowing or importing every concrete class.

- **Simple factory:** a function or method that chooses the class from an argument.
- **Factory Method (GoF):** a base class defines a method for creating objects, and subclasses decide which class to create.

Benefits: creation logic lives in one place, and adding a new type touches only the factory and the new class (Open/Closed). In Python, a dictionary of classes is often all you need.

PYTHON (TESTED)

```
class EmailNotifier:
    def send(self, msg): return f"Email: {msg}"
class SmsNotifier:
    def send(self, msg): return f"SMS: {msg}"
class WhatsAppNotifier:
    def send(self, msg): return f"WhatsApp: {msg}"

NOTIFIERS = {"email": EmailNotifier, "sms": SmsNotifier, "whatsapp": WhatsAppNotifier}

def make_notifier(channel):
    try:
        return NOTIFIERS[channel]()
    except KeyError:
        raise ValueError(f"Unknown channel: {channel}") from None

for ch in ["email", "whatsapp"]:
    print(make_notifier(ch).send("Class starts at 7 pm"))
```

OUTPUT

```
Email: Class starts at 7 pm
WhatsApp: Class starts at 7 pm
```

Q79. What is the difference between Factory Method and Abstract Factory?

EXPERIENCED · DESIGN PATTERNS

Point	Factory Method	Abstract Factory
Creates	One product	A family of related products
Mechanism	Inheritance: subclasses override a creation method	Composition: a factory object with several creation methods
Guarantees	The right single type	Products that belong together are used together
Example	`Dialog.create_button()` returns a Windows or Mac button	`DarkThemeFactory` creates a dark button, dark checkbox and dark menu

Use Abstract Factory when mixing products from different families would be a bug: a dark button on a light form, or a MySQL connection with a PostgreSQL query builder.

Q80. What is the Builder pattern?

EXPERIENCED · DESIGN PATTERNS

Builder constructs a complex object **step by step**, instead of through one constructor with many parameters (the "telescoping constructor" problem). Each step is a readable method call, optional parts can be skipped, and the builder can validate everything before producing the final object.

In Java it is very common (Lombok's `@Builder`). In Python, keyword arguments with defaults and dataclasses remove most of the need, but builders are still useful for complex, multi-step construction such as queries, HTTP requests or reports.

PYTHON (TESTED)

```
class Pizza:
    def __init__(self, size, crust, toppings):
        self.size, self.crust, self.toppings = size, crust, toppings
    def __repr__(self):
        return f"{self.size} {self.crust} pizza with {', '.join(self.toppings) or 'cheese only'}"

class PizzaBuilder:
    def __init__(self): self._size, self._crust, self._toppings = "medium", "thin", []
    def size(self, s): self._size = s; return self
    def crust(self, c): self._crust = c; return self
    def add(self, t): self._toppings.append(t); return self
    def build(self):
        if self._size == "small" and len(self._toppings) > 3:
            raise ValueError("Too many toppings for a small pizza")
        return Pizza(self._size, self._crust, list(self._toppings))

print(PizzaBuilder().size("large").crust("cheese burst").add("paneer").add("onion").build())
print(PizzaBuilder().build())
```

OUTPUT

```
large cheese burst pizza with paneer, onion
medium thin pizza with cheese only
```

Q81. What is the Strategy pattern?

EXPERIENCED · DESIGN PATTERNS

Strategy defines a family of interchangeable algorithms, puts each in its own class (or function), and lets the client choose one at runtime. The calling code stays the same while the behaviour changes.

Examples: payment methods, discount rules, sorting or ranking algorithms, route planning by car, bike or walking. It replaces long `if/elif` chains and follows the Open/Closed Principle. In Python, a strategy can simply be a function.

PYTHON (TESTED)

```
def flat_shipping(order_total):   return 50
def free_above_999(order_total):  return 0 if order_total >= 999 else 50
def express(order_total):         return 150

class Checkout:
    def __init__(self, shipping_strategy):
        self.shipping = shipping_strategy
    def total(self, amount):
        return amount + self.shipping(amount)

for strategy in [flat_shipping, free_above_999, express]:
    print(f"{strategy.__name__:15} -> Rs {Checkout(strategy).total(1200)}")
```

OUTPUT

```
flat_shipping   -> Rs 1250
free_above_999  -> Rs 1200
express         -> Rs 1350
```

Q82. What is the Observer pattern?

EXPERIENCED · DESIGN PATTERNS

Observer sets up a **one-to-many** relationship: when one object (the subject or publisher) changes, all its registered observers (subscribers) are notified automatically. The subject does not need to know what the observers do.

It is everywhere: UI event listeners, Django signals, pub/sub messaging, webhooks, and spreadsheet cells that recalculate. Watch out for memory leaks from observers that are never unsubscribed (weak references can help), and for chains of updates that are hard to follow.

PYTHON (TESTED)

```
class Course:
    def __init__(self, title):
        self.title = title
        self._observers = []
    def subscribe(self, fn):
        self._observers.append(fn)
    def publish_lecture(self, name):
        for notify in self._observers:
            notify(self.title, name)

def email_students(course, lecture):  print(f"Email: new lecture '{lecture}' in {course}")
def update_dashboard(course, lecture): print(f"Dashboard: {course} now has '{lecture}'")

dsa = Course("Master DSA")
dsa.subscribe(email_students)
dsa.subscribe(update_dashboard)
dsa.publish_lecture("Graphs: BFS")
```

OUTPUT

```
Email: new lecture 'Graphs: BFS' in Master DSA
Dashboard: Master DSA now has 'Graphs: BFS'
```

Q83. What is the Decorator pattern? How is it different from a Python @decorator?

EXPERIENCED · DESIGN PATTERNS

The **Decorator design pattern** adds behaviour to an **object** at runtime by wrapping it in another object with the same interface. Wrappers can be stacked: a data stream can be wrapped with compression, then encryption, then buffering, in any combination, without a subclass for each combination.

A **Python** `@decorator` is a language feature that wraps a **function or class** when it is defined, for example `@lru_cache` or `@login_required`. It is inspired by the same idea, but it works on functions at definition time, not on objects at runtime.

PYTHON (TESTED)

```
class Coffee:
    def cost(self): return 100
    def desc(self): return "coffee"

class Milk:
    def __init__(self, drink): self.drink = drink
    def cost(self): return self.drink.cost() + 20
    def desc(self): return self.drink.desc() + " + milk"

class Caramel:
    def __init__(self, drink): self.drink = drink
    def cost(self): return self.drink.cost() + 35
    def desc(self): return self.drink.desc() + " + caramel"

order = Caramel(Milk(Coffee()))              # wrappers stacked at runtime
print(order.desc(), "= Rs", order.cost())
```

OUTPUT

```
coffee + milk + caramel = Rs 155
```

Q84. What is the Adapter pattern?

EXPERIENCED · DESIGN PATTERNS

Adapter converts the interface of one class into the interface that the client expects, so classes with incompatible interfaces can work together. It is like a travel plug adapter.

Typical case: your app expects every payment provider to have `pay(amount)`, but a new third-party SDK has `make_transaction(paise, currency)`. Instead of changing your app or the SDK, you write a small adapter class that translates between them.

PYTHON (TESTED)

```
class ThirdPartyGateway:                    # cannot change this code
    def make_transaction(self, paise, currency):
        return f"Processed {paise} paise in {currency}"

class GatewayAdapter:                        # what our app expects: pay(rupees)
    def __init__(self, gateway): self.gateway = gateway
    def pay(self, rupees):
        return self.gateway.make_transaction(int(rupees * 100), "INR")

def checkout(payment, amount):
    return payment.pay(amount)

print(checkout(GatewayAdapter(ThirdPartyGateway()), 499.50))
```

OUTPUT

```
Processed 49950 paise in INR
```

Q85. What are the Template Method and Facade patterns?

EXPERIENCED · DESIGN PATTERNS

Template Method (behavioural): a base class defines the skeleton of an algorithm in one method, and subclasses fill in specific steps by overriding hook methods. The order of steps is fixed, only the details vary. Example: a data import job that always does extract, validate, transform, load, where each data source implements extract differently. Django's class-based views work this way.

Facade (structural): one simple class in front of a complicated subsystem. Instead of calling the inventory, payment, invoice and shipping services separately, the client calls `OrderFacade.place_order()`. It reduces coupling between clients and the subsystem's many classes.

Q86. What is the State pattern?

EXPERIENCED · DESIGN PATTERNS

State lets an object change its behaviour when its internal state changes, by moving each state's behaviour into its own class. The object delegates to its current state object, and states decide the next state.

It replaces large `if status == ...` blocks spread across many methods. Typical uses: order lifecycles (placed, shipped, delivered, cancelled), elevators, vending machines, document approval workflows and TCP connection states. Invalid transitions, such as cancelling a delivered order, are rejected in one obvious place.

PYTHON (TESTED)

```
class Placed:
    def ship(self, order):   order.state = Shipped();   return "shipped"
    def cancel(self, order): order.state = Cancelled(); return "cancelled, refund issued"

class Shipped:
    def ship(self, order):   return "already shipped"
    def cancel(self, order): return "cannot cancel: already shipped, raise a return instead"

class Cancelled:
    def ship(self, order):   return "cannot ship a cancelled order"
    def cancel(self, order): return "already cancelled"

class Order:
    def __init__(self): self.state = Placed()
    def ship(self):   return self.state.ship(self)
    def cancel(self): return self.state.cancel(self)

o = Order()
print(o.ship(), "|", o.cancel(), "|", type(o.state).__name__)
o2 = Order()
print(o2.cancel(), "|", o2.ship())
```

OUTPUT

```
shipped | cannot cancel: already shipped, raise a return instead | Shipped
cancelled, refund issued | cannot ship a cancelled order
```

Q87. What is a metaclass?

EXPERIENCED · ADVANCED PYTHON

A metaclass is the class of a class. Just as objects are created by classes, **classes are created by metaclasses**. The default metaclass is `type`. A custom metaclass can change how classes are built: validate them, add methods, register them, or enforce naming rules.

They power Django models and older ORMs. But they are rarely needed: class decorators and `__init_subclass__` handle most cases more simply. As Tim Peters put it, if you wonder whether you need them, you don't.

PYTHON (TESTED)

```
class RequireDocstring(type):
    def __new__(mcls, name, bases, ns):
        if bases and not ns.get("__doc__"):
            raise TypeError(f"class {name} must have a docstring")
        return super().__new__(mcls, name, bases, ns)

class Base(metaclass=RequireDocstring):
    pass

class Good(Base):
    """Documented class."""

try:
    class Bad(Base):
        pass
except TypeError as e:
    print("TypeError:", e)
print(type(Good).__name__, type(int).__name__)
```

OUTPUT

```
TypeError: class Bad must have a docstring
RequireDocstring type
```

Q88. What is __init_subclass__? How do you build a plugin registry with it?

EXPERIENCED · ADVANCED PYTHON

`__init_subclass__` is a class method that Python calls on a parent class **whenever a subclass is defined**. It lets a base class customise or register its subclasses without a metaclass.

A common use is a plugin registry: every subclass registers itself under a name, so new plugins work just by defining a class. File exporters, payment providers and command handlers are typical examples.

PYTHON (TESTED)

```
class Exporter:
    registry = {}
    def __init_subclass__(cls, fmt, **kwargs):
        super().__init_subclass__(**kwargs)
        Exporter.registry[fmt] = cls
    def export(self, rows): raise NotImplementedError

class CsvExporter(Exporter, fmt="csv"):
    def export(self, rows): return "\n".join(",".join(map(str, r)) for r in rows)

class JsonExporter(Exporter, fmt="json"):
    def export(self, rows):
        import json; return json.dumps(rows)

rows = [[1, "Asha"], [2, "Rohan"]]
print(sorted(Exporter.registry))
print(Exporter.registry["csv"]().export(rows))
```

OUTPUT

```
['csv', 'json']
1,Asha
2,Rohan
```

Q89. What are common OOP anti-patterns?

EXPERIENCED · DESIGN

- **God object:** one huge class that knows and does everything (`Manager`, `Utils`, `Helper`). Split it by responsibility.
- **Deep inheritance hierarchies:** five levels of subclasses where a change at the top breaks everything. Prefer composition.
- **Anaemic domain model:** classes that are only data with getters and setters, while all the logic lives in separate "service" functions. The objects protect nothing.
- **Inheritance for code reuse only:** `Stack(list)` exposes `insert` and `sort`, which break the stack's rules.
- **Feature envy:** a method that mostly uses another object's data probably belongs to that object.
- **Getters and setters for everything:** breaks encapsulation in practice; ask the object to do the work instead ("tell, don't ask").
- **Singleton overuse:** hidden global state that makes testing hard.

Q90. Design a parking lot system (low-level design).

EXPERIENCED · LOW-LEVEL DESIGN

This is the most common low-level design question. Interviewers want requirements first, then classes, relationships and a few key methods.

1. **Requirements:** multiple levels; spot sizes for bikes, cars and trucks; issue a ticket on entry; calculate the fee on exit; show free spots.
2. **Classes:** `ParkingLot` (has levels), `Level` (has spots), `Spot` (size, occupied vehicle), `Vehicle` with subclasses, `Ticket`, and a `PricingStrategy`.
3. **Relationships:** composition from lot to levels to spots; a ticket refers to a vehicle and a spot.
4. **Design choices:** pricing as a Strategy so rules can change; spot allocation behind one method so it can be optimised; thread safety for concurrent entry gates.

A compact, working core:

PYTHON (TESTED)

```
from enum import Enum

class Size(Enum):
    BIKE = 1; CAR = 2; TRUCK = 3

class Vehicle:
    size = None
    def __init__(self, number): self.number = number
class Bike(Vehicle):  size = Size.BIKE
class Car(Vehicle):   size = Size.CAR
class Truck(Vehicle): size = Size.TRUCK

class Spot:
    def __init__(self, sid, size): self.id, self.size, self.vehicle = sid, size, None
    def fits(self, v): return self.vehicle is None and self.size.value >= v.size.value

class ParkingLot:
    RATE = {Size.BIKE: 10, Size.CAR: 30, Size.TRUCK: 60}       # Rs per hour
    def __init__(self, spots): self.spots = spots; self.tickets = {}
    def park(self, v):
        spot = next((s for s in self.spots if s.fits(v)), None)
        if not spot: raise RuntimeError(f"No spot for {v.number}")
        spot.vehicle = v; self.tickets[v.number] = spot
        return f"{v.number} -> spot {spot.id}"
    def leave(self, number, hours):
        spot = self.tickets.pop(number); v, spot.vehicle = spot.vehicle, None
        return f"{number} pays Rs {self.RATE[v.size] * max(1, hours)}"

lot = ParkingLot([Spot("B1", Size.BIKE), Spot("C1", Size.CAR), Spot("T1", Size.TRUCK)])
print(lot.park(Car("GJ05AB1234")))
print(lot.park(Bike("GJ05XY0001")))
print(lot.park(Car("GJ05CD5678")))       # takes the truck spot: larger fits
print(lot.leave("GJ05AB1234", 3))
```

OUTPUT

```
GJ05AB1234 -> spot C1
GJ05XY0001 -> spot B1
GJ05CD5678 -> spot T1
GJ05AB1234 pays Rs 90
```

Q91. Design an LRU cache class.

EXPERIENCED · LOW-LEVEL DESIGN

An LRU (Least Recently Used) cache holds a fixed number of items; when it is full, it removes the item that was used longest ago. Both `get` and `put` must be O(1).

The classic design combines a **hash map** (key to node, for O(1) lookup) with a **doubly linked list** (to move items to the front and remove from the back in O(1)). In Python, `OrderedDict` already combines both, so the class is short. In an interview, mention the hash map plus linked list design even if you use `OrderedDict`, and be ready to implement the linked list by hand.

PYTHON (TESTED)

```
from collections import OrderedDict

class LRUCache:
    def __init__(self, capacity):
        self.capacity = capacity
        self.data = OrderedDict()
    def get(self, key):
        if key not in self.data:
            return -1
        self.data.move_to_end(key)              # mark as recently used
        return self.data[key]
    def put(self, key, value):
        if key in self.data:
            self.data.move_to_end(key)
        self.data[key] = value
        if len(self.data) > self.capacity:
            self.data.popitem(last=False)       # evict least recently used

c = LRUCache(2)
c.put("a", 1); c.put("b", 2)
c.get("a")                                      # a is now most recent
c.put("c", 3)                                   # evicts b
print(c.get("b"), c.get("a"), c.get("c"), list(c.data))
```

OUTPUT

```
-1 1 3 ['a', 'c']
```

Q92. How do you approach any low-level design (LLD) interview question?

EXPERIENCED · LOW-LEVEL DESIGN

Questions such as "design a library system, an elevator, BookMyShow, Splitwise or a chess game" all follow the same steps:

1. **Clarify requirements** for 3 to 5 minutes: core features, scale, what is out of scope.
2. **Identify entities** (nouns become classes) and **actions** (verbs become methods).
3. **Define relationships:** inheritance only for real "is-a"; composition and association elsewhere. Sketch a quick class diagram.
4. **Apply SOLID and patterns where they help:** Strategy for changing rules (pricing, splitting bills), State for lifecycles (elevator, order), Observer for notifications, Factory for creation.
5. **Handle the hard parts:** concurrency (two users booking the same seat), validation, error cases.
6. **Write the core classes and methods**, then walk through one complete scenario.

Interviewers value clear trade-offs over a perfect diagram. Our Low Level System Design in Python course covers these case studies end to end.

Q93. What is the difference between OOP and functional programming?

EXPERIENCED · DESIGN

Point	Object-oriented	Functional
Main unit	Objects: data with behaviour	Pure functions
State	Mutable state inside objects	Immutable data; new values instead of changes
Reuse through	Inheritance, composition, polymorphism	Higher-order functions and function composition
Side effects	Common	Pushed to the edges
Concurrency	Needs locks for shared state	Easier: no shared mutable state
Fits well	Modelling entities with lifecycles: users, orders, UI components	Data transformations, pipelines, calculations

Modern code mixes both: classes for the domain, immutable value objects, and pure functions for transformations. Python, Kotlin, Scala and modern Java all support both styles.

Q94. When should you avoid OOP or inheritance?

EXPERIENCED · DESIGN

- **Small scripts and one-off tasks:** a few functions are clearer than a class hierarchy.
- **Pure data transformation:** functions over dicts, dataclasses or DataFrames are usually simpler.
- **A class with one method and no state** is just a function in disguise.
- **Inheritance only to reuse code** when there is no real "is-a" relationship: use composition or a helper function.
- **Deep hierarchies for variations** that could be data or configuration (a `Plan` with a price field, not `BasicPlan`, `ProPlan`, `ProPlusPlan` subclasses).

A good rule: start simple, and introduce classes and abstractions when they remove real duplication or protect real rules.

Q95. How do you make a class thread-safe?

EXPERIENCED · ADVANCED PYTHON

When several threads change the same object, operations like `self.count += 1` are not atomic: they read, add and write in separate steps, so updates can be lost. In Python the GIL does not prevent this.

- **Lock** the critical section with `threading.Lock` (or `RLock` if the same thread may re-enter).
- **Avoid shared mutable state:** use immutable objects, or give each thread its own data.
- **Use thread-safe structures** such as `queue.Queue` to pass work between threads.
- **Keep locked sections short**, and always acquire multiple locks in the same order to avoid deadlocks.

Java note: `synchronized` methods or blocks, `ReentrantLock`, and atomic classes such as `AtomicInteger`.

PYTHON (TESTED)

```
import threading

class Counter:
    def __init__(self):
        self.value = 0
        self._lock = threading.Lock()
    def increment(self):
        with self._lock:                # one thread at a time
            self.value += 1

c = Counter()
def work():
    for _ in range(100_000):
        c.increment()

threads = [threading.Thread(target=work) for _ in range(4)]
for t in threads: t.start()
for t in threads: t.join()
print(c.value)                          # always 400000
```

OUTPUT

```
400000
```

Q96. What is the difference between an entity and a value object?

EXPERIENCED · DESIGN

These come from Domain-Driven Design, and they explain when objects should compare by identity and when by value.

Point	Entity	Value object
Identity	Has a unique ID that stays the same over time	No identity; defined only by its values
Equality	Same ID means same entity, even if fields changed	Equal if all values are equal
Mutability	Changes over its lifecycle	Immutable; replace it instead of changing it
Examples	Student, Order, BankAccount	Money(500, INR), Address, DateRange, Email

Value objects remove whole classes of bugs: a `Money` class can refuse to add rupees to dollars, and an `Email` class can validate once at creation.

Q97. How do you unit test a class that depends on other classes?

EXPERIENCED · TESTING

Test the class in isolation by replacing its dependencies with **test doubles**:

- **Stub:** returns fixed answers.
- **Fake:** a working lightweight version, such as an in-memory repository.
- **Mock:** also records how it was called, so you can assert on the interaction.

This is easy only if dependencies are **injected** (passed in) rather than created inside the class, which is one of the main practical benefits of the Dependency Inversion Principle. Python's `unittest.mock` provides `Mock`, `MagicMock` and `patch`.

PYTHON (TESTED)

```
from unittest.mock import Mock

class EnrollmentService:
    def __init__(self, payments, mailer):
        self.payments, self.mailer = payments, mailer
    def enroll(self, student, course, fee):
        if not self.payments.charge(student, fee):
            return "payment failed"
        self.mailer.send(student, f"Welcome to {course}")
        return "enrolled"

payments = Mock(); payments.charge.return_value = True
mailer = Mock()
svc = EnrollmentService(payments, mailer)

print(svc.enroll("asha@example.com", "Master DSA", 4999))
mailer.send.assert_called_once_with("asha@example.com", "Welcome to Master DSA")
print("mailer called correctly:", mailer.send.call_count == 1)

payments.charge.return_value = False
print(svc.enroll("rohan@example.com", "Master DSA", 4999), "| mails sent:", mailer.send.call_count)
```

OUTPUT

```
enrolled
mailer called correctly: True
payment failed | mails sent: 1
```

Q98. When should you use a regular class, a dataclass, a NamedTuple or Pydantic?

EXPERIENCED · ADVANCED PYTHON

Option	Best for	Notes
Regular class	Objects with real behaviour and rules	Full control; you write `__init__`, `__repr__`, `__eq__` yourself
`@dataclass`	Mostly data, inside your own code	Generates boilerplate; optional frozen, ordering and slots; no runtime type validation
`NamedTuple`	Small immutable records	Tuple behaviour: unpacking, indexing, very light
Pydantic `BaseModel`	Data from outside: API requests, config files, JSON	Validates and converts types at runtime; JSON in and out; used by FastAPI

Rule of thumb: validate at the boundaries of your system (Pydantic), then use dataclasses or plain classes inside.

Q99. What is single dispatch? How is it different from method overloading?

EXPERIENCED · ADVANCED PYTHON

`functools.singledispatch` (and `singledispatchmethod` for methods) creates a function whose implementation is chosen at runtime by the **type of the first argument**. It is Python's closest feature to overloading, and it follows the Open/Closed Principle: new types are added by registering new functions, without editing existing ones.

Unlike Java overloading, which the compiler resolves from the declared types of all arguments, single dispatch looks at the actual runtime type of only one argument.

PYTHON (TESTED)

```
from functools import singledispatch

@singledispatch
def to_display(value):
    return str(value)

@to_display.register
def _(value: float):
    return f"{value:,.2f}"

@to_display.register
def _(value: list):
    return ", ".join(to_display(v) for v in value)

@to_display.register
def _(value: bool):
    return "Yes" if value else "No"

print(to_display(1234567.891), "|", to_display([1, 2.5, True]), "|", to_display("hi"))
```

OUTPUT

```
1,234,567.89 | 1, 2.50, Yes | hi
```

Q100. How would you refactor a large, messy class?

EXPERIENCED · DESIGN

This tests practical judgement more than theory. A safe, step-by-step answer:

1. **Add tests first** around the current behaviour, so you know the refactor did not break anything.
2. **Find responsibilities:** group methods and fields by what they change for. Each group is a candidate class (Single Responsibility).
3. **Extract classes one at a time** and have the original class delegate to them, so callers keep working.
4. **Replace type checks with polymorphism** where `if type == ...` chains appear (Open/Closed).
5. **Inject dependencies** such as databases and email clients instead of creating them inside (Dependency Inversion), which also makes testing easy.
6. **Remove dead code and rename** for clarity, in small commits that each keep the tests passing.

The SOLID refactoring tool near the top of this page walks through exactly this on an example class.

Keep practising

Revise one level at a time, and explain each answer out loud as if you were in the interview.

Use the class builder, MRO explorer, OOP principle game and SOLID walkthrough at codeanddebug.in/blog/oops-interview-questions to make the ideas stick.

Cover the other core subjects with 100 DBMS and 100 Computer Networks Interview Questions at codeanddebug.in/blog.

Watch the video explanations on YouTube: @codeanddebug.