**OOP in Python — Interview Q&A**

**1. What are the four pillars of OOP?**

The four pillars are:

1. **Encapsulation** → Binding data & methods into one unit.
2. **Abstraction** → Hiding implementation details, showing only essential features.
3. **Inheritance** → Reusing code by creating new classes from existing ones.
4. **Polymorphism** → Same method behaving differently depending on the object.

**2. Difference between @staticmethod and @classmethod?**

* **@staticmethod** → Does not take self or cls. It’s like a normal function inside a class. Used for utility/helper functions.
* **@classmethod** → Takes cls as a parameter. Works with **class variables**, not instance variables. Used when you need to modify class-level data.

**3. Can Python support multiple inheritance?**

👉 **Yes.**  
A class in Python can inherit from multiple parent classes. Example:

class A: pass

class B: pass

class C(A, B): pass

Python resolves conflicts using the **MRO (Method Resolution Order)**.

**4. What is method overloading in Python?**

👉 Python does **not support traditional method overloading** (like Java or C++).  
But it can be **achieved using default arguments or \*args**.

Example:

class Example:

def add(self, a=0, b=0, c=0):

return a + b + c

obj = Example()

print(obj.add(2, 3)) # 5

print(obj.add(2, 3, 4)) # 9

**5. What is method overriding?**

👉 **Method overriding** happens when a **child class** provides its own implementation of a method already defined in the **parent class**.

class Animal:

def sound(self):

print("Some sound")

class Dog(Animal):

def sound(self):

print("Bark 🐶")

d = Dog()

d.sound() # Output: Bark 🐶

**6. Difference between Encapsulation & Abstraction?**

| **Feature** | **Encapsulation** | **Abstraction** |
| --- | --- | --- |
| Meaning | Hides data by restricting direct access (private variables). | Hides complexity, shows only essential details. |
| Focus | "How to hide data?" | "What to show?" |
| Implementation | Using \_ and \_\_ prefixes (private/protected) | Using abstract classes & methods |
| Example | Bank balance hidden from outside | Vehicle start\_engine() defined, but implementation left to subclasses |

**7. What are dunder methods?**

👉 **Dunder methods** = Double Underscore methods (also called magic methods).  
They are special methods Python uses for operator overloading and object behavior.

Examples:

* \_\_init\_\_ → Constructor
* \_\_str\_\_ → String representation
* \_\_len\_\_ → Length
* \_\_add\_\_ → Add two objects

**✅ 30 More OOP Interview Questions with Answers**

**8. What is a class in Python?**

A **class** is a blueprint for creating objects. It defines attributes (data) and methods (functions).

**9. What is an object in Python?**

An **object** is an instance of a class. Example:

class Car: pass

my\_car = Car() # my\_car is an object

**10. What is self in Python OOP?**

self represents the **instance of the class**. It lets objects access attributes and methods.

**11. What is the difference between \_\_init\_\_ and \_\_new\_\_?**

* \_\_new\_\_ → Creates a new object (low-level, rarely overridden).
* \_\_init\_\_ → Initializes object attributes (most common).

**12. What is inheritance in Python?**

A mechanism where one class (**child**) can use properties and methods of another (**parent**).

**13. Explain multiple inheritance in Python with example.**

class A: pass

class B: pass

class C(A, B): pass

👉 Class C inherits from both A and B.

**14. What is the difference between is-a and has-a relationship?**

* **is-a** → Inheritance (Dog is an Animal).
* **has-a** → Composition (Car has an Engine).

**15. What is composition in OOP?**

Using objects of one class inside another class instead of inheritance.

class Engine: pass

class Car:

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

self.engine = Engine()

**16. What are access modifiers in Python?**

Python doesn’t have strict access modifiers. But uses naming conventions:

* Public → variable
* Protected → \_variable
* Private → \_\_variable

**17. What is operator overloading in Python?**

Ability to use operators (+, -, \*, etc.) on objects via dunder methods.  
Example:

class Book:

def \_\_init\_\_(self, pages): self.pages = pages

def \_\_add\_\_(self, other): return self.pages + other.pages

**18. What is polymorphism in Python?**

Same method name, different behavior depending on object.

**19. What is the difference between function overloading and overriding?**

* **Overloading** → Same function name, different parameters (not supported directly in Python).
* **Overriding** → Child class redefines parent method.

**20. What is the difference between class variable and instance variable?**

* **Class variable** → Shared by all objects (school\_name).
* **Instance variable** → Unique for each object (self.name).

**21. Can we make a constructor private in Python?**

No strict private constructors. But we can restrict using factory methods or @classmethod.

**22. What is a metaclass in Python?**

A **class of a class**. Defines how classes behave. Default metaclass = type.

**23. Can a class inherit from multiple abstract classes?**

👉 Yes. Python supports multiple inheritance with abstract classes.

**24. What is super() in Python?**

Used to call a method from the parent class inside a child class.

**25. What is the difference between \_\_str\_\_ and \_\_repr\_\_?**

* \_\_str\_\_ → User-friendly representation (for print()).
* \_\_repr\_\_ → Developer-friendly representation (for debugging).

**26. What is duck typing in Python?**

"If it walks like a duck and quacks like a duck, it’s a duck."  
👉 Object type doesn’t matter, only if it has required methods/attributes.

**27. Can we create an abstract class without abstract methods?**

👉 Yes. It just can’t be instantiated.

**28. What is the difference between shallow copy and deep copy?**

* **Shallow copy** → Copies object but references same inner objects.
* **Deep copy** → Copies entire object including nested objects.

**29. What is a singleton class?**

A class that allows only **one instance**. Implemented by controlling object creation.

**30. What is method resolution order (MRO)?**

The order in which Python searches for a method in inheritance hierarchy. Checked via ClassName.mro().

**31. What is \_\_slots\_\_ in Python?**

Limits attributes of an object, prevents dynamic creation of new attributes → saves memory.

**32. Difference between abstract class and interface?**

Python doesn’t have true interfaces, but abstract classes can mimic them. Interface = all abstract methods, abstract class = mix of abstract & normal methods.

**33. Can we change attributes of an object dynamically?**

👉 Yes. Python allows adding attributes to objects at runtime.

obj.new\_attr = "Hello"

**34. What is the use of id() function in Python?**

Returns the memory address of an object.

**35. What is del in Python OOP?**

Deletes an object reference. Calls \_\_del\_\_() if defined.

**36. What is difference between @property and normal methods?**

* @property allows a method to be accessed like an attribute.  
  Example:

class Circle:

def \_\_init\_\_(self, r): self.r = r

@property

def area(self): return 3.14 \* self.r \* self.r

**37. What is the difference between procedural programming and OOP?**

* Procedural → Based on functions.
* OOP → Based on objects.

**38. What are mixins in Python OOP?**

A class designed only to provide extra methods (not a full parent). Helps in **code reuse**.

**39. What is the difference between deep inheritance and shallow inheritance?**

* **Shallow inheritance** → Few levels of hierarchy.
* **Deep inheritance** → Many levels (makes code harder to maintain).

**40. Can Python classes have private methods?**

👉 Yes, by prefixing with \_\_, but still accessible via name mangling.

**41. What is the diamond problem in OOP? How does Python solve it?**

When multiple inheritance creates ambiguity in which parent class to use.  
Python solves it with **MRO (C3 linearization).**

**42. Can we define multiple constructors in Python?**

Not directly, but can simulate using default arguments or @classmethod.

**43. What is monkey patching in Python?**

Dynamically changing or extending methods of a class at runtime.

**44. Difference between instance method, class method, and static method?**

* Instance → Needs self
* Class → Needs cls
* Static → No self or cls

**45. What is the use of \_\_call\_\_ dunder method?**

Makes an object behave like a function.

class Test:

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

print("Object called like function")

**46. Can we create a class without using class keyword?**

👉 Yes, using type() metaclass.

**47. What is multiple dispatch in Python?**

Python doesn’t support it directly, but can be implemented with libraries like functools.singledispatch.

**48. What is a data class in Python?**

Introduced in Python 3.7 (@dataclass). Used to create classes mainly for storing data without writing boilerplate code.

**49. What is the difference between mutable and immutable objects in Python OOP?**

* Mutable → Can change after creation (list, dict, custom objects).
* Immutable → Cannot change (int, str, tuple).

**50. Can a class inherit from built-in classes?**

👉 Yes. Example:

class MyList(list): pass

**Advanced OOP in Python – 30 Professional Interview Questions**

**51. What is the difference between classmethod() and factory methods?**

* **classmethod()**: General method bound to the class, used to modify class-level data.
* **Factory method**: A special @classmethod that returns class objects in a controlled/customized way.

**52. What are descriptors in Python OOP?**

Descriptors are classes that define how attributes are accessed in another class using \_\_get\_\_, \_\_set\_\_, and \_\_delete\_\_. They’re used behind @property.

**53. What is the use of \_\_dict\_\_ in Python classes?**

It stores the **namespace dictionary** of an object → all attributes & methods are stored inside it.

**54. What is the difference between @property and descriptors?**

* @property is a built-in **syntactic sugar** for descriptors.
* Descriptors are more powerful, allow reusable attribute management.

**55. Can we inherit dunder methods?**

👉 Yes. Child classes automatically inherit them unless overridden.

**56. What is the difference between shallow inheritance and composition?**

* **Shallow inheritance**: Uses parent-child hierarchy.
* **Composition**: Instead of extending, includes other classes inside. More flexible.

**57. Can we prevent inheritance in Python?**

👉 Yes. By raising exceptions in \_\_init\_subclass\_\_() method.

class Final:

def \_\_init\_subclass\_\_(cls, \*\*kwargs):

raise TypeError("Subclassing not allowed")

**58. How does Python handle memory for objects?**

* Uses **reference counting** and **garbage collector**.
* When no reference exists, object is deleted.

**59. What is a metaclass used for?**

Metaclasses define the **rules for class creation**.  
Example: Auto-logging, auto-validations.

**60. What is the difference between type() and isinstance()?**

* type(obj) → Returns the class/type of an object.
* isinstance(obj, cls) → Checks if an object is an instance of a class or subclass.

**61. Can we override \_\_new\_\_ method? Why?**

👉 Yes, to customize **object creation** (e.g., implementing Singleton pattern).

**62. What is cooperative multiple inheritance?**

When classes in a hierarchy use super() correctly so that all parent constructors/methods get called in **MRO order**.

**63. What is the diamond problem in OOP and how does Python solve it?**

Occurs when a class inherits from two classes that have a common ancestor.  
👉 Python solves using **C3 Linearization (MRO)**.

**64. What are weak references in Python?**

References that don’t increase the reference count of an object. Used to avoid memory leaks (via weakref module).

**65. Can Python classes have private constructors?**

👉 Not truly, but can simulate using \_\_new\_\_ or metaclasses.

**66. Difference between composition and aggregation?**

* **Composition** → Strong relationship (Car has Engine, Engine cannot exist without Car).
* **Aggregation** → Weak relationship (Department has Teachers, but teachers can exist independently).

**67. What is monkey patching?**

Changing a class/method at runtime. Example:

class Test: pass

def new\_method(): print("patched!")

Test.show = new\_method

**68. What are frozen dataclasses?**

Dataclasses with frozen=True behave like **immutable objects**.

**69. What is the difference between deep inheritance and mixins?**

* Deep inheritance → multiple hierarchical levels.
* Mixin → Small reusable classes to add behavior, not full hierarchy.

**70. Can Python classes implement interfaces?**

Python has no formal interfaces, but abstract base classes (abc.ABC) act like interfaces.

**71. What is the difference between hasattr(), getattr(), and setattr()?**

* hasattr(obj, "attr") → Check if object has attribute.
* getattr(obj, "attr", default) → Get attribute value.
* setattr(obj, "attr", value) → Set attribute value.

**72. What is covariance and contravariance in OOP?**

* **Covariance** → Subclass return type allowed when overriding a method.
* **Contravariance** → Subclass can accept broader parameter types.

**73. What is duck typing vs static typing?**

* **Duck typing**: Behavior matters (method exists), not type.
* **Static typing**: Explicit type-checking (like Java, C++).

**74. What is abc.ABCMeta?**

It’s the metaclass used for abstract base classes in Python.

**75. How does Python implement polymorphism without method overloading?**

👉 Using **default arguments, \*args, and duck typing**.

**76. What is the purpose of \_\_iter\_\_ and \_\_next\_\_?**

They make a class **iterable**, so it can be used in for loops.

**77. What is difference between is and == in Python OOP?**

* is → Checks identity (memory address).
* == → Checks value equality (calls \_\_eq\_\_).

**78. Can we inherit from built-in types like list or dict?**

👉 Yes. Example:

class MyList(list):

def sum(self):

return sum(self)

**79. What is difference between abstract class and mixin?**

* Abstract class → Defines a blueprint with abstract methods.
* Mixin → Adds small pieces of reusable functionality.

**80. What is the difference between overriding and hiding methods?**

* **Overriding** → Subclass redefines parent method.
* **Hiding** → Redefining class/static method with the same name in subclass, but doesn’t override instance method.