Python Functions & OOPs Interview Questions

1. What is a function in Python?

Answer: A function is a reusable block of code designed to perform a specific task. It improves code structure, readability, and modularity. In Python, functions are first-class objects, which means they can be passed around and used like any other value.

2. Why do we use functions?

Answer: Functions help break a program into smaller components, reducing duplication and making code easier to understand. They support reusability and simplify debugging by isolating logic into manageable units.

3. What is the difference between a function and a method?

Answer: A function is independent, while a method belongs to a class or object. Methods automatically receive `self`, giving them access to instance data. Functions rely only on explicitly passed arguments.

4. How do you define and call a function in Python?

Answer: You define a function using `def`, provide parameters, and then call it using its name. Example: `def add(a,b): return a+b` and `add(2,3)`. Functions can return values or simply perform actions.

5. What is the difference between arguments and parameters?

Answer: Parameters are placeholders defined in the function header, while arguments are actual values passed during the call. Understanding the difference is important for using defaults, keywords, and *args.

6. What are default arguments in Python?

Answer: Default arguments allow specifying fallback values so callers can omit them. Example: `def add(a,b=0): return a+b`. They simplify function usage but must be used carefully to avoid mutable default pitfalls.

7. What are keyword arguments?

Answer: Keyword arguments are passed using parameter names, improving clarity and removing order dependency. They make APIs more readable and help avoid mutable default pitfalls.

8. What are *args and *kwargs used for?

Answer: `*args` captures extra positional arguments as a tuple, while `**kwargs` captures extra named arguments as a dictionary. They make functions flexible and are commonly used in decorators or wrappers.

9. What is the purpose of the return statement?

Answer: The `return` statement sends a value back to the caller and stops function execution. It enables chaining functions and using outputs elsewhere. Without `return`, Python returns None by default.

10. Can a function return multiple values?

Answer: Yes, Python can return multiple values separated by commas, which are packaged into a tuple. This is useful for returning multiple computed results in a clean and readable way.

11. What is a docstring?

Answer: A docstring is a string written inside a function to describe what it does, its parameters, and return values. It improves readability and helps tools like `help()` display documentation. Docstrings follow triple quotes and are a key part of writing clean, maintainable Python code.

12. What is recursion?

Answer: Recursion is a technique where a function calls itself to solve smaller parts of a problem. It is commonly used in algorithms like factorial, tree traversal, and divide-and-conquer problems. Base conditions prevent infinite recursion. It helps simplify complex iterative logic.

13. What are lambda functions?

Answer: Lambda functions are small anonymous functions created using the `lambda` keyword. They are mainly used for short operations like sorting, mapping, or filtering. Lambdas contain only one expression, making them compact and lightweight for simple tasks.

14. What is the difference between lambda and normal functions?

Answer: Lambda functions are anonymous and contain only one expression, while normal functions have names and can contain multiple statements. Lambdas are best for short inline tasks, whereas regular functions are used for larger, detailed logic. Internally, both are function objects.

15. What is function annotation?

Answer: Function annotations allow you to add metadata to parameters and return values using a `:` or `->`. They help document expected types but are not enforced by Python. Annotations improve code readability and support type-checking tools like mypy.

16. What is a higher-order function?

Answer: A higher-order function either accepts another function as an argument or returns a function. Examples include `map`, `filter`, and decorators. They help implement modular, functional-style programming and improve code reuse.

17. What is a callback function?

Answer: A callback function is a function passed as an argument to another function, executed later when an event or condition is met. It is commonly used in asynchronous programming, GUIs, and event-driven architectures. Callbacks help decouple logic.

18. What do map(), filter(), and reduce() do?

Answer: `map()` applies a function to each item in an iterable, `filter()` keeps items that satisfy a condition, and `reduce()` combines items into a single result. They help perform transformations, filtering, and cumulative operations efficiently.

19. What is a generator function?

Answer: A generator function uses `yield` to produce one value at a time without storing the entire sequence in memory. Generators are memory-efficient and ideal for working with large datasets or streams. They pause execution between yields.

20. What is the difference between yield and return?

Answer: `return` ends a function and sends back a value immediately, while `yield` pauses the function and resumes later. `yield` turns a function into a generator, enabling lazy evaluation. `return` outputs once, whereas `yield` can output multiple times.

21. What is memoization?

Answer: Memoization is an optimization technique where results of expensive function calls are cached and reused when the same inputs occur again. It helps speed up recursive algorithms like Fibonacci. Python provides built-in memoization support using `functools.lru_cache`.

22. Does Python support function overloading?

Answer: Python does not support true compile-time function overloading like Java or C++. Defining a function again with the same name overrides the previous one. Overloading behavior can be achieved using default parameters, *args, or condition checks inside the function.

23. What is function overriding?

Answer: Function overriding occurs when a subclass defines a method with the same name as one in its parent class, providing a new implementation. It supports polymorphism—allowing different behaviors for the same method call depending on object type.

24. What is a closure in Python?

Answer: A closure occurs when a nested function remembers values from its enclosing scope even after the outer function has finished executing. Closures are widely used in decorators, callback functions, and factory patterns. They help retain state without using classes.

25. What is a decorator?

Answer: A decorator is a function that modifies or enhances another function's behavior without changing its actual code. It wraps one function inside another. Decorators are widely used for logging, authentication, timing, caching, and enforcing rules.

26. How do you write a custom decorator?

Answer: To write a decorator, you create a function that accepts another function, defines an inner wrapper, and returns it. Example:

```
def decor(fn):
    def wrapper():
        return fn()
    return wrapper
 Decorators are applied using the @ symbol.
```

27. What is partial function application?

Answer: Partial function application involves fixing some arguments of a function and creating a new function with fewer parameters. Python's `functools.partial` is commonly used for configuring functions in advance, especially when passing them as callbacks.

28. What is currying?

Answer: Currying transforms a multi-argument function into a chain of single-argument functions. It is commonly used in functional programming. While Python doesn't do it automatically, it can be implemented manually or using higher-order function techniques.

29. What is the LEGB rule?

Answer: LEGB stands for Local, Enclosing, Global, Built-in—the order Python uses to resolve variable names. Understanding this order helps avoid naming conflicts and debugging issues related to variable shadowing.

30. What is a namespace?

Answer: A namespace is a mapping between names and objects. Each scope (global, local, built-in) has its own namespace. Namespaces help avoid collisions between identifiers in different parts of the program.

31. What is the global keyword?

Answer: `global` is used to modify a global variable inside a function. Without it, assignments create a new local variable. It allows functions to update module-level values, but excessive use reduces code clarity.

32. What is the nonlocal keyword?

Answer: `nonlocal` is used in nested functions to modify variables from the enclosing (outer) function's scope. It helps closures retain and update state without using global variables.

33. What are first-class functions?

Answer: First-class functions can be stored, passed, and returned like any other variable. Python treats functions as objects, enabling powerful functional programming patterns such as decorators and callbacks.

34. Can functions be stored in data structures?

Answer: Yes, functions can be stored in lists, tuples, dictionaries, or any data structure. This is useful in command dispatch systems, menus, and strategy patterns where behavior is selected dynamically.

35. What is a pure function?

Answer: A pure function always returns the same output for the same input and has no side effects. They improve testability, predictability, and parallel execution. They are heavily used in functional programming.

36. What is tail recursion?

Answer: Tail recursion is when a function calls itself as its final action. It improves efficiency in languages that optimize tail calls. Python, however, does not support tail recursion optimization.

37. Why Python does not support tail-call optimization?

Answer: Python avoids tail-call optimization to maintain readable stack traces for debugging. Removing stack frames would make debugging recursive functions harder and less intuitive.

38. How are functions stored in memory?

Answer: Functions are objects stored in the heap. Their names reference these objects in memory. They include metadata such as code objects, defaults, annotations, and closure references.

39. What are anonymous functions?

Answer: Anonymous functions (lambda functions) are small unnamed functions created with `lambda`. They are used for concise operations, especially in map, filter, reduce, or sorting operations.

40. What is scope in Python?

Answer: Scope defines the region where a variable is accessible. Python resolves names using LEGB: Local, Enclosing, Global, Built-in. Understanding scope prevents naming conflicts.

41. What is the difference between local and global scope?

Answer: Local scope refers to variables inside a function, while global scope refers to variables at the module level. A function can read globals but must use `global` to modify them.

42. What are positional-only parameters?

Answer: Positional-only parameters (denoted by `/`) must be passed by position and not by name. They improve API clarity and prevent accidental keyword usage.

43. What are keyword-only parameters?

Answer: Keyword-only parameters (denoted by `*`) must be passed using their names. They enforce clarity and avoid confusion when functions accept many optional arguments.

44. What is a sentinel value in function parameters?

Answer: A sentinel value is a unique object used to detect missing or special arguments without relying on None. Implemented using `object()` to avoid accidental clashes.

45. What is function introspection?

Answer: Function introspection means examining a function's attributes at runtime, such as `__name__`, `__doc__`, `__defaults__`, or `__annotations__`. It is used in debugging and decorators.

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

Answer: OOP is a programming paradigm based on objects and classes. It models real-world entities through attributes (data) and methods (behavior), promoting modularity and reusability.

47. What are the main principles of OOP?

Answer: The four pillars of OOP are Encapsulation, Abstraction, Inheritance, and Polymorphism. They help organize code, reduce duplication, and make systems extensible.

48. What is a class in Python?

Answer: A class is a blueprint for creating objects. It defines attributes and behaviors that objects of that class will share. Classes encapsulate data and functions into one structure.

49. What is an object?

Answer: An object is an instance of a class that contains its own data in instance variables. Each object can operate independently using methods defined in the class.

50. What is init()?

Answer: `__init__()` is a constructor method called automatically when an object is created. It initializes instance variables and prepares the object for use.

51. What is self in Python?

Answer: `self` refers to the current instance of the class and is automatically passed to instance methods. It allows access to instance variables and other methods. It's much like `this` in other languages.

52. What are instance variables?

Answer: Instance variables are attributes unique to each object. They are created inside the `__init__()` method and belong to individual instances, allowing each object to maintain its own state.

53. What are class variables?

Answer: Class variables are shared across all objects of a class. They are defined inside the class but outside methods. They are useful for storing common data like counters or configuration.

54. What is inheritance?

Answer: Inheritance allows a class to acquire attributes and methods from another class. It helps code reuse and enables polymorphism. Parent-child relationships make applications more structured.

55. What is single, multiple, and multilevel inheritance?

Answer: Single inheritance has one parent class, multiple inheritance has several parent classes, and multilevel inheritance forms a chain of parent-child-grandchild classes. Python supports all three.

56. What is method overriding?

Answer: Method overriding occurs when a subclass provides a new version of a method inherited from a parent class. It enables polymorphism—different objects behave differently using the same method.

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

Answer: Python does not support traditional overloading. Redefining a function replaces the previous one. Overloading-like behavior is achieved using default parameters, *args, or manual checks.

58. What is polymorphism?

Answer: Polymorphism allows the same method name to behave differently depending on the object calling it. It enables flexibility in code and is a core OOP principle seen in overriding and duck typing.

59. What is encapsulation?

Answer: Encapsulation restricts direct access to data by using methods to control interactions. Python uses naming conventions like `_` and `__` to signal private or protected attributes.

60. What is abstraction?

Answer: Abstraction hides unnecessary implementation details and exposes only essential functionality. In Python, abstraction is implemented through abstract classes and interfaces using `abc` module.

61. What is super() used for?

Answer: `super()` is used to call the parent class's methods, especially inside `__init__()` to initialize inherited attributes. It supports cooperative multiple inheritance and avoids manual parent calls.

62. Difference between class method and static method?

Answer: Class methods use `cls` and can modify class-level data, while static methods receive no automatic argument and behave like normal functions inside a class. Static methods suit utility tasks.

63. What are magic methods in Python?

Answer: Magic methods start and end with double underscores (e.g., `__add__`, `__len__`). They define object behavior for operators, printing, indexing, and built-in functions.

64. What is operator overloading?

Answer: Operator overloading allows custom classes to modify the behavior of operators like +, -, <, or ==. This makes objects behave more like built-in types, improving usability.

65. What is composition vs inheritance?

Answer: Composition is a HAS-A relationship where one class contains objects of another. Inheritance is an IS-A relationship. Composition is preferred when you want flexible, loosely coupled objects.

66. What is an abstract class?

Answer: An abstract class defines methods but cannot be instantiated. Using the `abc` module, it forces subclasses to implement required methods, enforcing a contract.

67. What is an interface? Does Python support it?

Answer: Python does not have formal interfaces but achieves them using Abstract Base Classes (ABCs). Interfaces define a contract of methods that must be implemented by subclasses.

68. What is MRO (Method Resolution Order)?

Answer: MRO defines the order Python follows to search for methods in multiple inheritance. Python uses the C3 linearization algorithm. MRO can be seen using ClassName.**mro**.

69. What is the diamond problem in OOP?

Answer: The diamond problem occurs when multiple inheritance causes ambiguity about which parent method to call. Python solves this using MRO and C3 linearization.

70. What is **str**() vs **repr**()?

Answer: str() returns a readable user-friendly string, used by print(). **repr**() returns an unambiguous developer-friendly string for debugging and recreating objects.

71. What is a metaclass?

Answer: A metaclass controls how classes are created. Just like classes create objects, metaclasses create classes. Python uses `type` as the default metaclass. They allow customizing class behavior and enforcing rules, often used in frameworks.

72. What is **new**()?

Answer: new() is a static method responsible for creating a new instance of a class. It is called before **init**() and returns the new object. It is usually overridden when customizing object creation or using immutables like tuples.

73. Difference between **new**() and **init**()?

Answer: new() handles object creation, while **init**() handles object initialization. **new**() returns the instance to be used, and **init**() sets up its attributes. **new** is used rarely, mostly when subclassing immutable types.

74. What are descriptors?

Answer: Descriptors are classes that define how attribute access works using **get**, **set**, and **delete**. They power properties, methods, and many internal Python features. They allow customizing attribute management.

75. What is **slots** used for?

Answer: slots restricts which attributes objects can have, preventing dynamic attribute creation. It reduces memory usage by eliminating the per-instance dictionary. Useful in performance-critical applications.

76. What is dynamic attribute assignment?

Answer: Dynamic attribute assignment means adding new attributes to an object at runtime. Since objects in Python are flexible, you can do things like `obj.new_attr = value`. This enables powerful customization and metaprogramming.

77. How does Python perform garbage collection?

Answer: Python uses reference counting as the primary mechanism and a cyclic garbage collector as a backup. Unreferenced objects are immediately reclaimed. Cyclic GC handles objects that reference each other.

78. What is reference counting?

Answer: Reference counting increments and decrements counters whenever variables reference or release an object. When the count reaches zero, the memory is freed immediately. This makes memory cleanup predictable.

79. What is a cyclic reference?

Answer: A cyclic reference occurs when two or more objects reference each other, preventing their reference count from reaching zero. Python's cyclic garbage collector detects and resolves these cycles automatically.

80. What is duck typing?

Answer: Duck typing means Python cares more about an object's behavior than its type. If an object "quacks like a duck," it can be used as one. This enables polymorphism without strict type hierarchies.

81. What is monkey patching?

Answer: Monkey patching is modifying classes or modules at runtime. It is useful for testing or temporary fixes but can make code harder to maintain. Python allows monkey patching because everything is dynamic.

82. Can you modify a class at runtime?

Answer: Yes, Python allows modifying classes and objects at runtime since they are mutable. You can add attributes, replace methods, or even change inheritance dynamically.

83. What is name mangling?

Answer: Name mangling automatically renames variables starting with __ to _ClassName__var to prevent accidental access. It is used to indicate private attributes in classes.

84. Are private variables truly private in Python?

Answer: No. Private variables can still be accessed using name-mangled versions like `obj._ClassName__var`. Python relies on conventions rather than strict access control.

85. What is aggregation?

Answer: Aggregation is a HAS-A relationship where one class contains another class's object, but both can exist independently. Example: A school has students.

86. What is composition?

Answer: Composition is a stronger HAS-A relationship where the contained object cannot exist without the container. Example: A car has an engine. Destroying the car destroys the engine.

87. What is dependency injection?

Answer: Dependency injection is supplying an object's dependencies from outside instead of creating them internally. It reduces coupling and improves testability.

88. Why is everything in Python an object?

Answer: Everything in Python is an object because the language is built with a unified object model. Functions, classes, modules, and even types themselves are objects, giving Python great flexibility.

89. Can a class inherit from multiple classes?

Answer: Yes, Python supports multiple inheritance. A class can inherit from multiple classes, and Python resolves method order using MRO.

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

Answer: `isinstance()` checks an object's type considering inheritance, while `type()` checks for an exact match and ignores inheritance hierarchies.