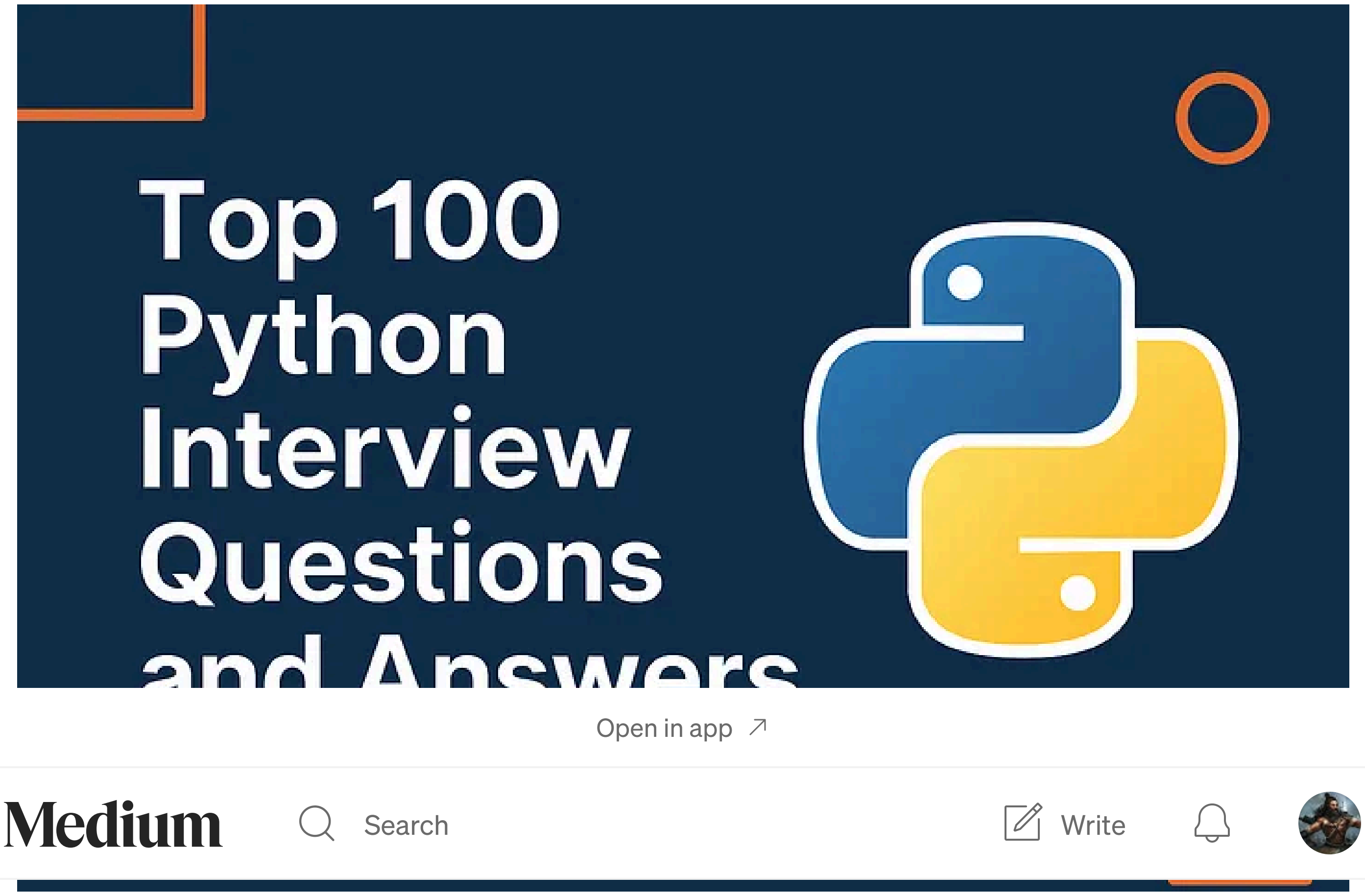
Top 100 Python Interview Questions and Answers

here we discuss questions regarding all aspects that are mostly asked in Python Interviews

Shirsh Shukla Follow 42 min read · Apr 5, 2025

-- 2



Python is one of the most popular programming languages in the world today. It is known for its simple and readable syntax, making it a great choice for beginners as well as experienced developers. Python is used in many areas of technology, including web development, data science, machine learning, automation, artificial intelligence, game development, and more.

Because of its wide usage, Python is a common topic in job interviews for software developers, data analysts, machine learning engineers, and other tech roles. Whether you're a fresher preparing for your first job, or a professional looking to switch careers, understanding the most common Python interview questions can help you feel more confident and perform better.

In this guide, we have gathered the **top 100 Python interview questions and answers**, starting from the basics and moving to more advanced topics. Each answer is written in simple words and explained clearly with examples so that anyone can understand — even if you're just starting your Python journey.

Let's dive into the questions and get ready to ace your Python interview!

1. What is Python?

Python is a popular, high-level programming language that is easy to read and write. It was created by Guido van Rossum and released in 1991. One of the main reasons why Python is so widely used is because of its simple syntax that looks like English. This makes it a great choice for beginners. Python is also versatile — you can use it for web development, data science, artificial intelligence, machine learning, automation, scripting, and more. It is free to use and open-source, which means anyone can download and use it. Big companies like Google, Instagram, and Netflix use Python in their systems.

2. What are the key features of Python?

Python has many features that make it a popular choice among developers. First, it is easy to learn and use. The syntax is simple, which helps beginners write code quickly. Second, it is an interpreted language, meaning Python executes code line by line, which makes debugging easier. Python is cross-platform, so it works on Windows, Mac, and Linux. It also supports object-oriented and functional programming. Another great feature is its huge standard library — it includes many tools and modules to do different tasks without writing everything from scratch. Python also has a large community, so getting help and learning resources is easy.

3. What are variables in Python?

Variables in Python are used to store information that you want to use later. Think of a variable as a box where you can keep something, like a number or a name. You don't need to mention the data type when creating a variable — Python figures it out. For example:

```
name = "Alice"
age = 25
```

Here, `name` stores a string and `age` stores an integer. You can also change the value of a variable anytime. Variables make it easy to write flexible programs where data can change. You just use the variable name wherever you need that value in your code.

4. What are data types in Python?

Data types tell Python what kind of value a variable holds. For example, `int` is for whole numbers, `float` is for decimal numbers, and `str` is for text. Python also has `bool` for True or False values, `list` for a collection of items, `tuple` for an unchangeable group of items, `dict` for key-value pairs, and `set` for unique items. These types help Python decide what kind of operations can be done with the data. You don't have to define the type—Python does it automatically. Knowing data types is important to avoid errors and to use functions correctly in your code.

5. What is a list in Python?

A list in Python is a collection of items that are ordered and changeable. Lists are written with square brackets like this:

```
fruits = ["apple", "banana", "cherry"]
```

You can add, remove, or change items in a list using functions like `append()` , `remove()` , or by using indexes like `fruits[1] = "orange"` . Lists can contain different data types—numbers, strings, even other lists. They are useful when you want to store multiple items in one variable. You can also loop through a list using a `for` loop. Lists are one of the most used data structures in Python because they are simple and powerful.

6. What is a tuple in Python?

A tuple is similar to a list, but the key difference is that you cannot change the items in a tuple once it is created. Tuples are **immutable**, which means unchangeable. They are written with parentheses like this:

```
colors = ("red", "green", "blue")
```

You can access items in a tuple using an index like `colors[0]`. Tuples are useful when you want to make sure that the data does not change. They are also slightly faster than lists. Tuples can be used as keys in dictionaries, but lists cannot because they are mutable. So, if you want to store fixed data, tuples are a good choice.

7. What is a dictionary in Python?

A dictionary in Python stores data in key-value pairs. Each key is linked to a value, like a word and its meaning in a real dictionary. Dictionaries are written using curly braces:

```
person = {"name": "Alice", "age": 30}
```

You can access the value by using the key: `person["name"]` returns `"Alice"`. You can also add new key-value pairs or update existing ones. Dictionaries are very useful when you need to store related information, like user details or configuration settings. Keys must be unique and immutable, like strings or numbers. Dictionaries are fast and great for looking up data by name or key.

8. What is a set in Python?

A set in Python is a collection of unique items. It is unordered, which means the items do not have a fixed position. Sets are written using curly braces:

```
numbers = {1, 2, 3, 2}
```

The set automatically removes duplicates, so the output will be `{1, 2, 3}`. Sets are useful when you want to remove duplicates or check for common values between groups using operations like union, intersection, and difference. Since sets are unordered, you cannot access items using indexes. Sets are also faster than lists when checking if an item exists. They are simple but powerful for handling unique data.

9. What is the difference between list and tuple in Python?

Both lists and tuples can store multiple items, but they have some key differences. Lists are **mutable**, which means you can change, add, or remove items after the list is created. Tuples are **immutable**, which means you cannot change them after they are made. Lists are written with square brackets `[ ]`, while tuples use parentheses `( )`. Because of immutability, tuples are faster and can be used as keys in dictionaries, while lists cannot. Use a list when your data might change. Use a tuple when your data should stay the same. Both are useful in different situations.

10. How do you write comments in Python?

Comments in Python are used to explain what your code is doing. Python ignores comments — they are just for humans reading the code. To write a single-line comment, start the line with `#`:

```
# This is a single-line comment
```

You can also write multi-line comments using triple quotes:

```
'''
This is a
multi-line comment
'''
```

Good comments help others (and yourself) understand your code, especially when it's long or complex. They make your code more readable and easier to maintain. Writing clear comments is considered a good programming habit in any language.

11. How is Python interpreted?

Python is an interpreted language, which means the code is not compiled before running. Instead, it is executed line-by-line by the Python interpreter. When you write Python code and run it, the interpreter reads each line, translates it into machine language, and then executes it immediately. This is different from compiled languages like C or Java, where the code is turned into a complete executable file before it runs. Being interpreted makes it easier to test and debug Python code. If there's an error, Python will stop at that line and show an error message. This feature is great for beginners and for rapid development.

12. What is indentation in Python and why is it important?

Indentation in Python refers to the spaces or tabs at the beginning of a line. Unlike many other programming languages, Python uses indentation to define blocks of code. For example, in loops, conditionals, and functions, the indented lines belong to the same code block. If indentation is not done correctly, Python will raise an error and stop running. This makes Python code more readable and clean. Here is a simple example:

```
if 5 > 2:
    print("Five is greater than two")
```

In the above code, the print statement is indented, which shows that it belongs to the `if` block. Proper indentation is very important in Python programming.

13. What are Python functions?

A function in Python is a block of reusable code that performs a specific task. You can define your own functions using the `def` keyword, or you can use built-in functions like `print()` , `len()` , and `type()` . Functions help you avoid repeating code and make your programs more organized. Here's an example of a simple function:

```
def greet(name):
    print("Hello, " + name)
```

You can call this function like `greet("Alice")`. Functions can take parameters and can also return values using the `return` keyword. Using functions makes your code shorter, cleaner, and easier to manage.

14. What is the difference between a function and a method in Python?

In Python, a **function** is a block of code that performs a task and is defined using the `def` keyword. It can be used on its own, outside of any class. For example:

```
def add(a, b):
    return a + b
```

A **method**, on the other hand, is a function that is associated with an object. Methods are defined inside classes and are called using dot notation. For example:

```
name = "Alice"
print(name.upper())  # upper() is a method
```

In simple terms, all methods are functions, but not all functions are methods. Methods always belong to an object or class, while functions do not.

15. What are arguments and parameters in Python functions?

Parameters are the names you define in a function when you write it. **Arguments** are the actual values you pass to the function when calling it. For example:

```
def greet(name):  # 'name' is a parameter
    print("Hello, " + name)
```

```
greet("Alice")  # "Alice" is the argument
```

In this case, `name` is a parameter, and `"Alice"` is the argument passed to the function. Python also supports default arguments, keyword arguments, and variable-length arguments. Understanding how parameters and arguments work is important for writing flexible and reusable functions.

16. What is the use of the return statement in Python?

The `return` statement in Python is used in a function to send a value back to the place where the function was called. It ends the function and passes the result to the caller. Here's an example:

```
def add(a, b):
    return a + b
```

```
result = add(5, 3)
print(result)  # Output: 8
```

In this case, the function `add` returns the sum of `a` and `b` , which is then stored in the variable `result` . If there is no `return` statement, the function will return `None` by default. Using `return` makes functions more useful because they can provide output to be used later.

17. What is the difference between `del` and `remove()` in Python?

In Python, both `del` and `remove()` are used to delete elements, but they operate differently and are used in different contexts.

- The `del` statement is a language construct used to delete an item at a specific index from a list or to delete entire variables or slices. It works with all types of objects, including lists, dictionaries, and variables. For example:

```
nums = [1, 2, 3, 4] del nums[1]  # removes the item at index 1 (value 2)
```

- The `remove()` method, on the other hand, is a list method that removes the **first occurrence** of a specific value from the list. It raises a `ValueError` if the item is not found:

```
nums = [1, 2, 3, 2]
nums.remove(2)  # removes the first 2
```

In summary, use `del` when you know the index or want to delete a variable. Use `remove()` when you want to delete a known value from a list.

18. What is the difference between for and while loops in Python?

The `for` loop and `while` loop are both used to repeat actions, but they are used in different situations. A `for` loop is best when you know in advance how many times you want to repeat something. It works well with lists, strings, and ranges:

```
for i in range(5):
    print(i)
```

A `while` loop is better when you don't know how many times you'll repeat and want to continue until a certain condition is false:

```
while i < 5:
    print(i)
    i += 1
```

So, use `for` when looping through known items, and use `while` when you need to loop based on a condition.

19. What is a conditional statement in Python?

Conditional statements are used to run certain blocks of code only when specific conditions are met. Python uses `if`, `elif`, and `else` for this:

```
age = 18
if age >= 18:
    print("You are an adult")
else:
    print("You are a minor")
```

You can also use `elif` (short for "else if") to check multiple conditions:

```
if score >= 90:
    print("A grade")
elif score >= 75:
    print("B grade")
else:
    print("C grade")
```

Conditional statements help your programs make decisions and behave differently based on input or data.

20. What is the use of the `break`, `continue`, and `pass` statements in Python?

These three statements control how loops behave:

- `break`: Stops the loop entirely and exits:

```
for i in range(5):
    if i == 3:
        break
    print(i)  # Prints 0, 1, 2
```

- `continue`: Skips the current loop cycle and moves to the next one:

```
for i in range(5):
    if i == 3:
        continue
    print(i)  # Skips 3
```

- `pass`: Does nothing. It's used as a placeholder where code is needed later:

```
for i in range(5):
    pass  # To be implemented later
```

These are useful for controlling loops more precisely based on your program's needs.

21. What are Python lists and how do you use them?

A list in Python is a collection of items that can hold different types of values like numbers, strings, or even other lists. Lists are **ordered** and **changeable**, meaning you can update, add, or remove items. You define a list using square brackets:

```
fruits = ["apple", "banana", "cherry"]
```

You can access list items by index, like `fruits[0]` which gives `"apple"`. You can also change values, like `fruits[1] = "orange"`. Python lists have many useful functions like `append()` to add an item, `remove()` to delete an item, and `sort()` to sort the list. Lists are one of the most used data types in Python.

22. What is the difference between a list and a tuple in Python?

Both lists and tuples are used to store multiple items, but the **main difference** is that lists are **mutable** (changeable), while tuples are **immutable** (cannot be changed). You create a list with square brackets `[]`, and a tuple with parentheses `()`:

```
my_list = [1, 2, 3]
my_tuple = (1, 2, 3)
```

You can change `my_list[0] = 10`, but you cannot change `my_tuple[0]`. Tuples are faster and take up less memory than lists. Use tuples when your data should not change, such as coordinates or fixed settings. Lists are better when you need to update, sort, or modify the data.

23. What are Python dictionaries and how are they used?

A dictionary in Python is a collection of **key-value pairs**. Each key is unique and maps to a value. You create a dictionary using curly braces `{}`:

```
person = {"name": "Alice", "age": 25}
```

You can access values using keys, like `person["name"]` which gives `"Alice"`. You can also add or update values, like `person["age"] = 30`. Dictionaries are useful when you want to store and retrieve data using names or identifiers instead of positions. Some helpful functions include `keys()`, `values()`, and `items()`. They are powerful for storing structured data, like JSON responses or configuration settings.

24. What are Python sets and what are they used for?

A set is a collection of **unique** items. It is **unordered**, so the items do not have a fixed position and cannot be accessed by index. Sets are defined using curly braces `{}`:

```
my_set = {1, 2, 3}
```

If you try to add a duplicate, it will be ignored. Sets are useful for checking membership and removing duplicates. You can use `add()` to insert elements and `remove()` to delete them. Python also supports set operations like union (`|`), intersection (`&`), and difference (`-`). Sets are great when you need fast lookups or want to ensure no duplicates exist.

25. How do you create a class in Python?

A class in Python is a blueprint for creating objects. It defines the structure and behavior (methods and variables) of an object. You create a class using the `class` keyword:

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

```
    def greet(self):
        print("Hello, my name is " + self.name)
```

The `__init__` method is the constructor and runs when a new object is created. You can create an object like `p1 = Person("Alice")` and call its method using `p1.greet()`. Classes help in object-oriented programming and allow you to create reusable code.

26. What is an object in Python?

An object in Python is an instance of a class. When you create a class, you're just defining the structure. But when you create an object using that class, you get a working version with real values. For example:

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

```
dog1 = Dog("Buddy")
```

Here, `dog1` is an object of the `Dog` class. It has its own copy of data and can use class methods. In Python, almost everything is an object—strings, lists, functions, and even classes. Objects make code modular, reusable, and organized.

27. What is inheritance in Python?

Inheritance in Python allows one class (called a **child** or **subclass**) to get features from another class (called a **parent** or **base class**). It helps in reusing code and building relationships between classes. Here's a basic example:

```
class Animal:
    def speak(self):
        print("Animal speaks")
```

```
class Dog(Animal):
    def bark(self):
        print("Dog barks")

d = Dog()
d.speak()
d.bark()
```

In this example, the `Dog` class inherits from `Animal`, so it can use the `speak()` method. Inheritance supports code reuse and helps organize code better when working with related classes.

28. What is polymorphism in Python?

Polymorphism means "many forms". In Python, polymorphism allows different classes to have methods with the same name, but different behavior. For example, if two classes have a method named `speak()`, you can call `speak()` on any object, and it will behave according to its class:

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

```
class Cat:
    def speak(self):
        return "Meow"

animals = [Dog(), Cat()]
for animal in animals:
    print(animal.speak())
```

Each object knows how to perform its version of the method. Polymorphism makes code flexible and helps when writing functions that can work with multiple types of objects.

29. What is encapsulation in Python?

Encapsulation is a concept in object-oriented programming that hides the internal details of a class and protects data from outside access. In Python, we use **private** variables (with a single or double underscore `_` or `__`) to indicate that they should not be accessed directly:

```
class Person:
    def __init__(self, name):
        self.__name = name  # private variable
```

```
    def get_name(self):
        return self.__name
```

In this example, `__name` is private. We access it using the `get_name()` method. Encapsulation helps keep your data safe, and only allows access through defined methods, making your code more secure and easier to maintain.

30. What is abstraction in Python?

Abstraction means showing only the essential features and hiding the unnecessary details. It helps reduce complexity and allows you to focus on what an object does, not how it does it. In Python, abstraction is often implemented using **abstract classes** and **methods** from the `abc` module:

```
from abc import ABC, abstractmethod
```

```
class Animal(ABC):
    @abstractmethod
    def make_sound(self):
        pass

class Dog(Animal):
    def make_sound(self):
        print("Bark")
```

Here, `Animal` is an abstract class, and `make_sound()` must be implemented in any child class. Abstraction helps in designing clean interfaces and focusing on high-level functionality.

31. What is the difference between `*args` and `**kwargs` in Python?

- `*args` allows a function to accept **any number of positional arguments**, packed as a tuple.
- `**kwargs` allows a function to accept **any number of keyword arguments**, packed as a dictionary.

Example:

```
def demo(*args, **kwargs):
    print(args)
    print(kwargs)

demo(1, 2, a=3, b=4)
```

These are useful for creating flexible functions, wrappers, and decorators.

32. What is the `if __name__ == "__main__"` statement used for?

The `if __name__ == "__main__"` statement is used to control the execution of code in Python scripts. When a Python file is run directly, the special built-in variable `__name__` is set to `"__main__"`. However, when that file is imported as a module into another script, `__name__` is set to the module's name instead.

This allows developers to write code that acts differently depending on whether it's run directly or imported. It's commonly used to encapsulate the script's entry point:

```
def main():
    print("Running as a script")

if __name__ == "__main__":
    main()
```

This is particularly useful in larger applications and during unit testing, as it allows for better organization and reuse of code.

33. What is a `defaultdict` in Python?

A `defaultdict` is a subclass of the built-in `dict` class that provides a default value for non-existent keys. This prevents `KeyError` exceptions and is particularly useful when counting items or grouping data.

It requires a factory function to specify the default value:

```
from collections import defaultdict
dd = defaultdict(int)
```

```
dd["apple"] += 1  # no error even though "apple" didn't exist
```

This is much cleaner than checking for key existence manually using `get()` or `if` statements. Common factories include `int`, `list`, and `set`.

34. What is list comprehension in Python?

List comprehension is a concise way to create lists in Python. Instead of using loops, you can use a single line of code. It improves readability and performance. Here's a basic example:

```
squares = [x*x for x in range(5)]
```

This creates a list `[0, 1, 4, 9, 16]`. You can also add conditions:

```
even = [x for x in range(10) if x % 2 == 0]
```

List comprehensions are useful when you want to transform or filter data quickly. They make your code shorter, cleaner, and easier to understand than using a full `for` loop.

35. What is a higher-order function in Python?

A higher-order function is any function that either accepts another function as an argument, or returns a function as its result. Python supports higher-order functions natively, which makes it a flexible language for functional-style programming.

Examples of built-in higher-order functions in Python include `map()`, `filter()`, and `sorted()`.

Example:

```
def apply_twice(func, value):
    return func(func(value))

def square(x):
    return x * x
print(apply_twice(square, 2))  # Output: 16
```

Higher-order functions promote reusability and abstraction, allowing more expressive and concise code.

36. How does Python handle memory management?

Python manages memory automatically using a system called **garbage collection**. This system keeps track of all objects and frees up memory that is no longer being used. Python also uses reference counting — each object keeps a count of how many references point to it. When that count reaches zero, the memory is released. Python's memory management is done by the **Python memory manager**, and it includes private heap space where all objects and data structures are stored. As a developer, you don't usually need to manage memory directly, but understanding how it works can help write better, more efficient code.

37. What is multiple inheritance in Python?

Multiple inheritance is when a class inherits from more than one parent class. Python fully supports this, which allows a class to combine functionality from multiple sources.

Example:

```
class A:
    def greet(self):
        print("Hello from A")
class B:
    def greet(self):
        print("Hello from B")
class C(A, B):
    pass
c = C()
c.greet()  # Uses A's method due to MRO
```

While powerful, multiple inheritance can make the class hierarchy hard to manage, so it should be used with care. The `super()` function and MRO help mitigate the complexity.

38. What is a package in Python?

A **package** in Python is a collection of modules organized in directories. It allows you to group related modules together. A package is a folder that contains an `__init__.py` file, which tells Python that the folder should be treated as a package. Example structure:

```
mypackage/
│
├── __init__.py
├── module1.py
└── module2.py
```

You can import modules from the package using dot notation:

```
from mypackage import module1
```

Packages help in organizing large applications and reusing code across projects. Python also has many third-party packages that can be installed using tools like `pip`.

39. What is the purpose of `__init__.py` in Python packages?

The `__init__.py` file in Python marks a directory as a package so that its modules can be imported. Without this file, Python won't recognize the folder as a package in older versions (though in modern Python, it's optional). This file can be empty or contain initialization code for the package. For example, you might import key modules or define variables inside it:

```
# __init__.py
from .module1 import function1
```

With this, users can simply do `from package import function1` instead of importing the whole module. It helps organize imports and controls how packages behave during import.

40. How do you handle exceptions in Python?

In Python, you handle errors and exceptions using `try`, `except`, and optionally `finally`. This allows your program to continue running even if something goes wrong. Here's an example:

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except ZeroDivisionError:
    print("Cannot divide by zero!")
except ValueError:
    print("Please enter a valid number.")
finally:
    print("This always runs.")
```

The `try` block runs the risky code. If there's an error, Python checks for a matching `except` block. The `finally` block always runs, whether an error occurred or not. Exception handling makes your programs more robust and user-friendly.

41. What is the difference between `break`, `continue`, and `pass` in Python?

In Python, `break`, `continue`, and `pass` are control flow statements, but each one serves a different purpose.

- `break`: It is used to exit a loop completely, even if the loop condition is still true. Once `break` is encountered, the loop stops running.

```
for i in range(5):
    if i == 3:
        break
    print(i)
```

- `continue`: It skips the current iteration and moves to the next one without stopping the loop.

```
for i in range(5):
    if i == 3:
        continue
    print(i)
```

- `pass` : It does nothing and is used as a placeholder where code is required syntactically but no action is needed.

```
for i in range(5):
    if i == 3:
        pass
    print(i)
```

These are useful for controlling how loops and conditionals behave during execution.

42. What are Python decorators?

A **decorator** in Python is a function that takes another function as input and adds extra functionality to it, without changing its original structure. It is often used to modify the behavior of a function or method dynamically.

```
def decorator(func):
    def wrapper():
        print("Before function call")
        func()
        print("After function call")
    return wrapper
```

```
@decorator
def greet():
    print("Hello!")

greet()
```

Here, `@decorator` wraps the `greet()` function and adds extra code before and after it. Decorators are used often in logging, authentication, timing, and access control. They help you write cleaner, reusable, and more readable code.

43. What is a `Counter` in Python?

`Counter` is a class from the `collections` module that helps count occurrences of elements in an iterable. It returns a dictionary-like object where elements are stored as keys and counts as values.

Example:

```
from collections import Counter
c = Counter("banana")
print(c)  # Counter({'a': 3, 'n': 2, 'b': 1})
```

It supports most dictionary operations and provides extra methods like `.most_common()` and `.elements()` for retrieving data in specific formats. It is extremely useful for statistics, frequency analysis, and text processing.

44. What are closures in Python?

Closures are functions that remember the values of variables from their enclosing lexical scope even after that scope has finished executing. In other words, a closure allows a function to access variables from an outer function that has already returned.

Here's a simple example:

```
def outer(msg):
    def inner():
        print(msg)
    return inner
```

```
greet = outer("Hello")
greet()  # prints "Hello"
```

In the above code, the function `inner()` forms a closure—it remembers the variable `msg` from its enclosing function `outer()` . Closures are useful for building function factories, decorators, and keeping state in a clean and elegant way.

45. What are *args and kwargs in Python?

In Python, `*args` and `**kwargs` are used in function definitions to allow the function to accept a variable number of arguments.

- `*args` collects extra **positional** arguments into a tuple.
- `**kwargs` collects extra **keyword** arguments into a dictionary.

```
def example(*args, **kwargs):
    print(args)
    print(kwargs)
```

```
example(1, 2, 3, name="Alice", age=25)
```

This would print:

```
(1, 2, 3)
{'name': 'Alice', 'age': 25}
```

These features make your functions flexible and reusable. You can call them with different numbers of parameters without changing the function definition.

46. What is the LEGB rule in Python?

LEGB stands for **Local → Enclosing → Global → Built-in**, and it defines the order in which Python searches for variables:

1. **Local:** Names inside the current function.
2. **Enclosing:** Names in outer (but not global) functions if nested.
3. **Global:** Names defined at the top-level of a script or module.
4. **Built-in:** Python's predefined names like `len()` or `str`.

Example:

```
x = "global"
def outer():
    x = "enclosing"
    def inner():
        x = "local"
        print(x)
    inner()
outer()  # prints "local"
```

Understanding LEGB is crucial for working with nested functions, closures, and scoping bugs.

47. How can you handle file operations in Python?

Python provides easy ways to work with files using built-in functions like `open()`, `read()`, `write()`, and `close()`. Here's a simple example of reading a file:

```
with open("example.txt", "r") as file:
    content = file.read()
    print(content)
```

The `with` statement automatically closes the file. You can also write to a file:

```
with open("example.txt", "w") as file:
    file.write("Hello, Python!")
```

Use modes like `"r"` for reading, `"w"` for writing, and `"a"` for appending. File operations are useful for data storage, configuration, and logging in applications.

48. What are Python's built-in data types?

Python has several built-in data types that are categorized into different groups:

- **Numeric types:** `int` , `float` , `complex`
- **Sequence types:** `list` , `tuple` , `range`
- **Text type:** `str`
- **Set types:** `set` , `frozenset`
- **Mapping type:** `dict`
- **Boolean type:** `bool`
- **Binary types:** `bytes` , `bytearray` , `memoryview`

Each type serves different purposes. For example, use `int` for whole numbers, `str` for text, `list` for ordered groups, and `dict` for key-value pairs. Understanding data types helps you store and manipulate data correctly in your programs.

49. What is the difference between mutable and immutable types in Python?

Mutable types can be changed after creation. Examples include `list`, `dict`, and `set`. You can add, remove, or change elements in these types.

Immutable types cannot be changed once created. Examples include `int`, `float`, `str`, and `tuple`. If you try to modify them, Python creates a new object instead.

```
x = "hello"
x = x + " world"  # Creates a new string
```

Knowing which types are mutable and which are not helps you avoid bugs, especially when passing variables into functions. It also affects performance and how memory is managed.

50. How do you define and use a function in Python?

To define a function in Python, you use the `def` keyword followed by the function name and parentheses. You can pass arguments into the function and return values using the `return` statement.

```
def greet(name):
    return f"Hello, {name}!"
```

```
message = greet("Alice")
print(message)
```

Functions make your code reusable and organized. Instead of repeating the same logic, you put it in a function and call it whenever needed. You can also have default arguments, variable-length arguments, and even functions inside functions.

51. What is a Python metaclass?

In Python, a metaclass is a class of a class — it defines how classes behave. While classes define how objects behave, metaclasses define how classes themselves are created.

By default, all classes in Python are instances of `type`, the default metaclass. However, you can define custom metaclasses by inheriting from `type`, and use them to automatically modify class attributes or behavior at the time of class creation.

Use cases include:

- Enforcing coding conventions (e.g., attribute naming rules)
- Automatically registering classes

- Singleton pattern implementations

Although powerful, metaclasses are an advanced feature and should be used with care to avoid unnecessary complexity.

52. What is duck punching (monkey patching) in Python?

Monkey patching (or duck punching) is the practice of changing or extending the behavior of libraries, classes, or modules at runtime.

Example:

```
import math
math.sqrt = lambda x: "No square roots allowed"
print(math.sqrt(9))  # Outputs: No square roots allowed
```

This can be useful for testing or hotfixes but is generally discouraged in production code because it can lead to unpredictable behavior and maintenance challenges.

53. What is a lambda function in Python?

A **lambda function** is a small anonymous function in Python. It's used when you need a simple function for a short period and don't want to formally define it using `def`. Lambda functions can take any number of arguments but only contain one expression.

Basic syntax:

```
lambda arguments: expression
```

Example:

```
add = lambda x, y: x + y
print(add(3, 5))  # Output: 8
```

Lambdas are useful in places where a quick function is needed, like with `map()`, `filter()`, and `sorted()`.

```
nums = [5, 2, 9]
sorted_nums = sorted(nums, key=lambda x: -x)
```

Although powerful, for complex operations, using regular functions is more readable.

54. What is method resolution order (MRO) in Python?

MRO is the order in which Python looks for methods in a class hierarchy. It determines which method gets called when multiple inheritance is involved. Python uses the C3 linearization algorithm to compute this order.

You can view a class's MRO using:

```
print(ClassName.__mro__)
```

```
class A: pass
class B(A): pass
class C(A): pass
class D(B, C): pass

print(D.__mro__)
```

This is especially useful in understanding complex inheritance and avoiding bugs due to unexpected method overrides.

55. What is exception handling in Python?

Exception handling is used in Python to catch and respond to errors that occur during program execution. Instead of crashing the program, you can handle the error gracefully using `try`, `except`, `else`, and `finally` blocks.

Here's how it works:

```
try:
    result = 10 / 0
except ZeroDivisionError:
    print("You can't divide by zero!")
else:
    print("No errors occurred.")
finally:
    print("This will always run.")
```

- `try` block contains code that might raise an error.
- `except` block handles the error.
- `else` runs if no error occurs.
- `finally` always runs, useful for cleanup.

Using exceptions helps in building robust applications that can recover from unexpected situations.

56. What are Python's magic methods?

Magic methods in Python are special methods with double underscores at the beginning and end of their names. They're also known as **dunder methods**. Python uses these methods to perform operator overloading and other behaviors internally.

Some common magic methods are:

- `__init__` : Constructor, called when an object is created.
- `__str__` : Returns a string representation of the object.

- `__len__` : Returns the length using `len()` .
- `__add__` : Defines behavior for `+` .

Example:

```
class Book:
    def __init__(self, title):
        self.title = title
```

```
    def __str__(self):
        return f"Book: {self.title}"

book = Book("Python 101")
print(book)  # Output: Book: Python 101
```

Magic methods make classes act like built-in types, improving readability and flexibility.

57. What is the difference between `is` and `==` in Python?

In Python:

- `==` checks if **values** of two variables are equal.
- `is` checks if two variables point to the **same object in memory**.

Example:

```
a = [1, 2, 3]
b = [1, 2, 3]
```

```
print(a == b)  # True, because contents are same
print(a is b)  # False, because they are two different objects
```

`is` is often used when comparing objects like `None`:

```
if my_var is None:
    print("Value is None")
```

Use `==` for equality of content and `is` when you care about identity, such as comparing singletons or cached objects.

58. What is the difference between a tuple and a list in Python?

Both tuples and lists are used to store multiple items, but:

- **List** is mutable: you can add, remove, or change items.
- **Tuple** is immutable: once created, it cannot be changed.

```
my_list = [1, 2, 3]
my_tuple = (1, 2, 3)
```

Lists are defined with square brackets `[]`, and tuples with parentheses `()`.

Lists are commonly used when data needs to change during runtime, while tuples are ideal for fixed data or when you want to ensure the data stays unchanged. Also, because tuples are immutable, they can be used as dictionary keys, unlike lists.

59. What is duck typing in Python?

Duck typing is a concept from dynamic typing. In Python, it means that the type or class of an object is less important than the methods or operations it supports. The idea is: *"If it walks like a duck and quacks like a duck, it's a duck."*

Example:

```
class Duck:
    def quack(self):
        print("Quack!")
```

```
class Person:
    def quack(self):
        print("I can quack too!")

def make_quack(thing):
    thing.quack()

make_quack(Duck())
make_quack(Person())
```

Even though `Duck` and `Person` are different classes, both can be passed because they have a `quack()` method. This flexibility makes Python code more reusable and simple.

60. How is memory managed in Python?

Python uses a combination of **reference counting** and a **garbage collector** to manage memory.

- Each object has a reference count: when the count reaches zero, it is deleted.
- Python also uses a garbage collector to handle **circular references**, where two objects refer to each other but are no longer used.

You don’t usually need to manage memory manually. Python handles it for you. But you can still check or influence it using the `gc` module:

```
import gc
gc.collect()
```

Also, memory is allocated in private heaps managed by the Python interpreter. Efficient memory handling is part of Python’s design to help developers focus on logic instead of low-level resource management.

61. What is the purpose of the `gc` module in Python?

The `gc` (garbage collection) module provides access to Python's automatic memory management. It allows you to manually trigger garbage collection, monitor objects that aren't being collected, and control collection thresholds.

For example:

```
import gc
gc.collect()  # Manually triggers garbage collection
```

It's particularly useful when dealing with circular references — where two objects refer to each other, preventing their reference count from dropping to zero. The `gc` module helps identify and clean these unreachable objects.

62. What are Python iterators?

An **iterator** in Python is an object that allows you to loop over its elements, one at a time. It must implement the `__iter__()` and `__next__()` methods. You can get an iterator from any iterable (like lists, tuples, sets) using the `iter()` function. Then, you use `next()` to get the next item.

Example:

```
my_list = [10, 20, 30]
it = iter(my_list)
print(next(it))  # 10
print(next(it))  # 20
```

When there are no more items, `next()` raises a `StopIteration` error. Iterators are memory-efficient because they don't store the whole sequence in memory, which is helpful for large datasets or file reading.

63. What is the difference between `deepcopy()` and `copy()`?

In Python, `copy()` and `deepcopy()` both come from the `copy` module and are used to duplicate objects, but they behave differently:

- `copy.copy()` creates a **shallow copy** of an object. It copies the outer object, but not the nested objects inside it. So changes to inner objects affect both copies.
- `copy.deepcopy()` creates a **completely independent clone**, including all nested objects. Changes to one object do not affect the other.

Example:

```
import copy
```

```
original = [[1, 2], [3, 4]]
shallow = copy.copy(original)
deep = copy.deepcopy(original)
```

```
original[0][0] = 99
print(shallow[0][0])  # 99 (affected)
print(deep[0][0])     # 1 (not affected)
```

Use `deepcopy()` when you need a completely separate object structure.

64. What is slicing in Python?

Slicing in Python lets you extract a portion of a list, string, or tuple using a range of indexes. The basic syntax is `object[start:stop:step]`.

Example:

```
my_list = [0, 1, 2, 3, 4, 5]
print(my_list[1:4])      # [1, 2, 3]
print(my_list[::2])      # [0, 2, 4]
print(my_list[::-1])     # [5, 4, 3, 2, 1, 0]
```

- `start` is the index to begin from (inclusive),
- `stop` is where to end (exclusive)
- `step` tells how many items to skip.

Slicing is powerful for reversing sequences, picking even-indexed items, or extracting parts of text. It works with strings too:

```
text = "Python"
print(text[1:4])  # "yth"
```

65. What is the difference between `append()` and `extend()` in lists?

Both `append()` and `extend()` are used to add elements to a list, but they do it differently:

- `append()` adds a **single element** to the end of the list, even if it's another list.
- `extend()` takes an **iterable** and adds **each element** from it to the list.

Example:

```
a = [1, 2]
a.append([3, 4])
print(a)  # [1, 2, [3, 4]]
```

```
b = [1, 2]
b.extend([3, 4])
print(b)  # [1, 2, 3, 4]
```

So, `append()` adds the whole object, while `extend()` breaks it apart and adds each item individually. Use `append()` when you want to keep an item together; use `extend()` when you want to expand the list.

66. What is the purpose of `enumerate()` in Python?

The `enumerate()` function is used when you need both the **index and value** while looping over an iterable. It adds a counter to the iterable and returns it as an enumerate object, which you can convert into a list or use in a loop.

Example:

```
fruits = ['apple', 'banana', 'cherry']
for index, fruit in enumerate(fruits):
    print(index, fruit)
```

Output:

```
0 apple
1 banana
2 cherry
```

This is much cleaner than using `range(len(fruits))` . You can also start the index at a custom number by using `enumerate(fruits, start=1)` . It's helpful when you're processing lists, keeping track of positions, or labeling items in a report or UI.

67. What is the difference between `filter()`, `map()`, and `reduce()`?

All three are functional programming tools in Python:

- `map()` applies a function to **every item** in an iterable and returns a new iterable.
- `filter()` applies a function that returns `True` or `False`, and only keeps items where the function returns `True`.
- `reduce()` repeatedly applies a function to **accumulate** a result (from `functools` module).

Example:

```
from functools import reduce
```

```
nums = [1, 2, 3, 4]

print(list(map(lambda x: x * 2, nums)))      # [2, 4, 6, 8]
print(list(filter(lambda x: x % 2 == 0, nums)))  # [2, 4]
print(reduce(lambda x, y: x + y, nums))      # 10
```

`map()` transforms data, `filter()` selects data, and `reduce()` combines data. They help write clean, short code for data processing.

68. What is the use of `zip()` in Python?

The `zip()` function is used to combine two or more iterables into a single iterable of tuples, where the i-th tuple contains the i-th element from each iterable. It stops at the shortest input length.

Example:

```
names = ['Alice', 'Bob']
scores = [85, 90]
zipped = zip(names, scores)
print(list(zipped))  # [('Alice', 85), ('Bob', 90)]
```

It's great for looping over multiple lists at once:

```
for name, score in zip(names, scores):
    print(f"{name} scored {score}")
```

If the input lists are of unequal length, the extra items are ignored. You can also unzip using `zip(*zipped_data)` .

69. What is `*args` and `**kwargs` in Python?

In Python:

- `*args` lets you pass a **variable number of positional arguments** to a function.

- `**kwargs` lets you pass a **variable number of keyword arguments** (as a dictionary).

Example:

```
def demo(*args, **kwargs):
    print("Args:", args)
    print("Kwargs:", kwargs)
```

```
demo(1, 2, 3, a=4, b=5)
```

Output:

```
Args: (1, 2, 3)
Kwargs: {'a': 4, 'b': 5}
```

These are useful when writing flexible functions that can accept any number of inputs. It helps when wrapping other functions or when you don't know in advance how many parameters might be passed.

70. What is recursion in Python?

Recursion is when a function calls itself to solve a smaller part of a problem until it reaches a base case. It's commonly used in problems like calculating factorials, Fibonacci numbers, and tree traversals.

Example:

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)
```

```
print(factorial(5))  # 120
```

Every recursive function must have a base case to prevent infinite recursion. Python limits recursion depth by default (about 1000 calls). You can check it using `sys.getrecursionlimit()`.

Recursion makes some problems easier to solve, though it may use more memory than loops.

71. What is a Python module?

A **Python module** is a file containing Python code, usually saved with a `.py` extension. Modules help organize and reuse code across multiple programs. You can define functions, variables, and classes inside a module and then import them into other files using the `import` statement.

For example, if you create a file `math_utils.py` with a function:

```
def add(a, b):
    return a + b
```

You can use it in another file like this:

```
import math_utils
print(math_utils.add(2, 3))
```

Python has many built-in modules like `math`, `random`, and `datetime`. You can also create your own custom modules to keep your code clean, organized, and maintainable, especially in large projects.

72. What are weak references in Python?

Weak references allow Python to refer to an object without increasing its reference count. This means the object can still be garbage-collected, which is useful when managing caches or graphs where circular references can occur.

Python provides `weakref` module:

```
import weakref

class MyClass:
    pass
obj = MyClass()
```

```
r = weakref.ref(obj)
print(r())  # Returns the object
del obj
print(r())  # Returns None (object is collected)
```

This mechanism helps avoid memory leaks in large or long-running applications.

73. What is the `super()` function in Python?

The `super()` function returns a proxy object that delegates method calls to a parent or sibling class. It's commonly used in class constructors (`__init__`) to ensure proper initialization in inheritance chains.

Example:

```
class Parent:
    def __init__(self):
        print("Parent initialized")

class Child(Parent):
    def __init__(self):
        super().__init__()
        print("Child initialized")
c = Child()
```

Using `super()` helps support maintainable, extensible code and is critical in multiple inheritance where calling the correct parent method matters.

74. What is the purpose of `__init__.py`?

The `__init__.py` file is used to mark a directory as a **Python package**. Without it, Python doesn't recognize the folder as part of the package system. Even if it's empty, it's still needed (especially in older Python versions).

For example, in this structure:

```
my_package/
    __init__.py
    module1.py
```

You can now do:

```
from my_package import module1
```

Besides identifying packages, `__init__.py` can also run initialization code or expose specific classes/functions when the package is imported. For instance, you can import functions from other modules inside `__init__.py`, making them accessible directly from the package. It helps organize your project structure for modular and maintainable code.

75. What is `collections.deque` and why is it better than a list for some use cases?

`collections.deque` (double-ended queue) is a part of Python's standard `collections` module. It is a specialized list-like container optimized for fast appends and pops from both ends.

While Python lists are efficient for append and pop operations at the **end**, they are slow when inserting or deleting items at the **beginning** because all elements must be shifted.

`deque` solves this by using a doubly linked list internally, allowing O(1) operations on both ends:

```
from collections import deque
dq = deque([1, 2, 3])
dq.appendleft(0)  # fast insert at the beginning
dq.pop()          # fast removal from end
```

It is ideal for queue and stack implementations, sliding windows, and breadth-first searches.

76. What are `namedtuple` s and why are they useful?

A `namedtuple` is a factory function in the `collections` module that returns a subclass of `tuple` with named fields. This makes your code more readable and self-documenting, while still being as memory-efficient as a regular tuple.

Example:

```
from collections import namedtuple
Point = namedtuple("Point", ["x", "y"])
p = Point(1, 2)
print(p.x, p.y)  # Access by name instead of index
```

Use `namedtuple` when you need immutable data structures with readable fields—ideal for data modeling, configuration, and return values.

77. What are Python comprehensions?

Python **comprehensions** are a concise way to create lists, sets, or dictionaries using a single line of code. They are often used as a more readable alternative to loops.

Example of list comprehension:

```
squares = [x*x for x in range(5)]
```

This creates `[0, 1, 4, 9, 16]`. Similarly, you can use:

- **Set comprehension:** `{x*x for x in range(5)}`
- **Dict comprehension:** `{x: x*x for x in range(5)}`

You can also add conditions:

```
even_squares = [x*x for x in range(10) if x % 2 == 0]
```

Comprehensions make the code cleaner and easier to read compared to using traditional loops, especially when the operation is simple and directly maps from input to output.

78. What is the difference between `@staticmethod` and `@classmethod` in Python?

Both `@staticmethod` and `@classmethod` are decorators in Python, but they serve different purposes. A `@staticmethod` is a function within a class that does not access instance (`self`) or class (`cls`) variables. It is used when the logic relates to the class but doesn't need its state. A `@classmethod` , on the other hand, takes the class itself as the first parameter (`cls`) and can modify class-level data. It's useful for alternative constructors or utility functions that operate on class variables. Use `@staticmethod` for general utility functions, and `@classmethod` when you need access to class properties.

79. What are Python annotations?

Python annotations are a way to add **type hints** to function parameters and return types. Introduced in PEP 484, they help improve code readability, allow static type checkers like `mypy` to analyze your code, and assist with IDE autocompletion.

Annotations do not affect runtime behavior by default; they are just metadata. For example:

```
def greet(name: str) -> str:
    return f"Hello, {name}"
```

Here, `name` is expected to be a string, and the function returns a string. You can also access annotations using the `__annotations__` attribute.

While annotations are optional, they are increasingly used in production code for documentation, validation, and tooling support.

80. What are type hints and why are they useful in Python?

Type hints (also called type annotations) allow developers to specify the expected data types of function parameters and return values. Introduced in Python 3.5 via PEP 484, they don't change how code runs but help with readability, autocompletion, and static type checking.

Example:

```
def add(a: int, b: int) -> int:
    return a + b
```

Tools like `mypy` can then analyze your code for type consistency without running it. Type hints are valuable in large codebases and collaborative projects.

81. What is a namespace in Python?

A **namespace** in Python refers to a space where names are mapped to objects. Think of it as a dictionary where the keys are variable names and the values are the objects those names refer to. Python uses namespaces to keep track of all the names in your program, like variable names, function names, class names, etc.

There are four types of namespaces in Python:

- **Built-in**: Contains built-in functions like `print()` and `len()`.
- **Global**: Contains names defined at the top-level of a script or module.
- **Enclosing:** Relevant for nested functions (outer function scope).
- **Local**: Inside a function or block.

For example:

```
x = 10  # Global namespace
```

```
def func():
    y = 5  # Local namespace
```

Namespaces prevent naming conflicts and help Python know which variable you're referring to in different parts of the program. You can access namespaces using functions like `globals()` and `locals()`.

82. What is the difference between global and local variables?

A **local variable** is defined inside a function and can only be used within that function. A **global variable**, on the other hand, is defined outside any function and can be accessed from anywhere in the code, including inside functions.

Example:

```
x = 5  # Global
```

```
def my_func():
    x = 10  # Local
    print(x)

my_func()  # Prints 10
print(x)   # Prints 5
```

If you want to change the global variable inside a function, you must use the `global` keyword:

```
def my_func():
    global x
    x = 20
```

Local variables help keep functions independent, while global variables can be accessed across functions but may lead to unexpected behavior if not managed properly.

83. What is a Python set and how is it different from a list?

A **set** in Python is an unordered collection of unique elements. It is defined using curly braces `{}` or the `set()` function. Unlike lists, sets do not allow duplicate values, and they are not indexed, meaning you cannot access items using indexes like `my_set[0]`.

Example:

```
my_list = [1, 2, 2, 3]
my_set = set(my_list)  # {1, 2, 3}
```

Key differences:

- **Uniqueness:** Sets automatically remove duplicates.
- **Order:** Lists maintain order; sets do not.
- **Mutability:** Both are mutable, but sets only contain immutable elements.
- **Operations:** Sets support mathematical operations like union, intersection, and difference.

Use sets when you need to store unique values or perform set-based operations efficiently.

84. How does exception handling work in Python?

Exception handling in Python is done using `try`, `except`, `finally`, and `else` blocks. It helps your program handle unexpected errors without crashing. You wrap the code that might throw an error in a `try` block, then use `except` to handle specific or general exceptions.

Example:

```
try:
    x = 1 / 0
except ZeroDivisionError:
    print("Cannot divide by zero.")
finally:
    print("This will always run.")
```

You can also use `else` to run code if no exception occurs:

```
try:
    x = 5
except:
    print("Error")
else:
    print("No error")
```

This system helps in writing robust and error-tolerant applications. Always catch specific exceptions instead of using a broad `except:` .

85. What is the difference between `repr()` and `str()` in Python?

Both `repr()` and `str()` are used to get string representations of objects, but for different purposes:

- `str()` is for **user-friendly** display (e.g., `print()` output).
- `repr()` is for **developer-oriented** output that ideally could be used to recreate the object.

Example:

```
s = "Hello"
print(str(s))   # Output: Hello
print(repr(s))  # Output: 'Hello'
```

You can override `__str__()` and `__repr__()` in custom classes for better control over their behavior.86. What is `None` in Python?

`None` is a special constant in Python that represents the **absence of a value** or a **null value.** It is often used to indicate that a variable doesn't have any meaningful data yet. It's an object of its own datatype — `NoneType`.

Example:

```
x = None
if x is None:
    print("x has no value")
```

Functions that don't explicitly return anything will return `None` by default:

```
def greet():
    print("Hello")
```

```
print(greet())  # Outputs "Hello" and then "None"
```

`None` is not the same as `0`, `False`, or an empty string. It's used in comparisons, default arguments, and when checking if a variable was assigned any value or not.

87. What is the use of the `id()` function in Python?

The `id()` function in Python returns the **unique identifier** of an object, which is its memory address in CPython (the standard Python implementation). This is useful for checking whether two variables refer to the same object in memory.

Example:

```
a = [1, 2, 3]
b = a
print(id(a), id(b))  # Same id
```

Even if two objects have the same value, they may not have the same `id`:

```
x = [1, 2]
y = [1, 2]
print(id(x) == id(y))  # False
```

You can combine `id()` with the `is` keyword to check identity. This is especially useful when dealing with mutable and immutable types and understanding object references.

88. What is a recursive lambda function?

A recursive lambda is a lambda function that calls itself to solve problems. However, since lambda functions in Python are anonymous and can't directly refer to themselves, we must assign them to a variable.

Example:

```
factorial = lambda n: 1 if n == 0 else n * factorial(n - 1)
print(factorial(5))  # Output: 120
```

This is a simple, elegant way to define recursion in functional style, though it's typically more readable using a `def` function for more complex logic.

89. What is a ternary operator in Python?

The **ternary operator** in Python is a way to write simple `if-else` statements in one line. It's also called a **conditional expression**.

Syntax:

```
x = value_if_true if condition else value_if_false
```

Example:

```
age = 18
status = "Adult" if age >= 18 else "Minor"
```

It makes code more concise and readable for small decisions. But avoid using it for complex logic, as it can reduce readability. It's commonly used when assigning values based on a quick condition.

90. What are Python assertions?

Assertions are a debugging tool used to test if a condition is true. If the condition is false, the program will raise an `AssertionError`. They help catch bugs early by checking if your assumptions in the code hold true.

Example:

```
x = 5
assert x > 0, "x must be positive"
```

If `x` is less than or equal to 0, the assertion will fail. You can provide a custom error message after the comma.

Assertions are mostly used in development and testing phases. In production, they can be disabled with the `-O` (optimize) switch. They are not a replacement for proper error handling but are very useful in spotting bugs quickly.

91. What is the use of `zip()` in Python?

The `zip()` function in Python is used to combine two or more iterables (like lists or tuples) into a single iterable of tuples. It pairs the elements from each iterable based on their position (index). If the iterables are of different lengths, `zip()` stops at the shortest one.

Example:

```
names = ["Alice", "Bob", "Charlie"]
scores = [85, 90, 78]
zipped = zip(names, scores)
print(list(zipped))  # [('Alice', 85), ('Bob', 90), ('Charlie', 78)]
```

You can use `zip()` for many tasks like combining data, iterating over multiple sequences at once, or transposing rows and columns in matrices. It's efficient and easy to use, often seen in data processing, file merging, or where multiple iterables need to be processed in parallel.

92. What is the purpose of `dir()` in Python?

The `dir()` function is a built-in utility that returns a list of all attributes and methods (including inherited ones) available for a given object. It's widely used for introspection, helping developers understand what functionality is available on an object without reading the source code or documentation.

For example:

```
print(dir([]))  # Shows all methods and attributes for a list
```

When called with no arguments, `dir()` returns the list of names in the current local scope.

This tool is especially useful in REPL environments or debugging sessions, helping you explore objects dynamically and understand the Python object model better.

93. What is the `map()` function in Python?

The `map()` function applies a given function to all items in an iterable and returns a map object (which is an iterator). It's a clean way to apply transformations to a list or tuple without writing a loop.

Syntax:

```
map(function, iterable)
```

Example:

```
numbers = [1, 2, 3, 4]
squared = map(lambda x: x**2, numbers)
print(list(squared))  # [1, 4, 9, 16]
```

`map()` is often used when you need to apply the same function to every element. It's efficient and can be combined with other functional programming tools like `filter()` or `reduce()`. It keeps code short and readable.

94. What is the `filter()` function in Python?

The `filter()` function is used to filter elements from an iterable based on a condition. It returns an iterator containing only the elements for which the function returns `True`.

Syntax:

```
filter(function, iterable)
```

Example:

```
numbers = [1, 2, 3, 4, 5]
even = filter(lambda x: x % 2 == 0, numbers)
print(list(even))  # [2, 4]
```

It is useful when you want to keep elements that meet certain criteria and discard the rest. `filter()` makes your code more expressive and avoids manual loops with conditional checks.

95. What is the `reduce()` function in Python?

The `reduce()` function from the `functools` module applies a function to the items of a sequence and reduces it to a single value. It processes the sequence pairwise.

Syntax:

```
from functools import reduce
reduce(function, iterable)
```

Example:

```python
from functools import reduce
numbers = [1, 2, 3, 4]
result = reduce(lambda x, y: x + y, numbers)
print(result)  # 10
```

You can use `reduce()` for operations like summing, multiplying, or combining values. It's powerful, but sometimes less readable than a loop or `sum()`. Use it when you need to process a list into a single result step by step.

96. What is a `frozenset` in Python?

A `frozenset` is the immutable version of a Python set. It cannot be modified after creation, which means you cannot add or remove elements. This immutability makes it hashable, allowing it to be used as a key in dictionaries or as elements in other sets.

Example:

```
fs = frozenset([1, 2, 3])
```

It supports all set operations like union, intersection, and difference. Use `frozenset` when you need to ensure the contents of a set do not change, especially when dealing with caching, memoization, or set-based keys.

97. What is the difference between `yield` and `return` in Python?

Both `yield` and `return` are used in functions, but they work very differently. `return` ends a function and sends back a value. Once `return` is called, the function ends. On the other hand, `yield` pauses the function and sends a value but keeps the function state alive for the next call.

With `yield`, the function becomes a generator. You can use `next()` to get the next value from it.

Example:

```
def gen():
    yield 1
    yield 2
```

```
g = gen()
print(next(g))  # 1
print(next(g))  # 2
```

`yield` is memory-efficient and great for iterating over large or infinite data.

98. What is the Global Interpreter Lock (GIL) in Python?

The Global Interpreter Lock (GIL) is a mechanism in the CPython interpreter that prevents multiple native threads from executing Python bytecodes at the same time. It exists to protect access to Python objects, ensuring thread safety. Because of the GIL, multi-threaded Python programs may not achieve true parallelism on multi-core CPUs for CPU-bound tasks. However, for I/O-bound operations like file handling or network requests, threading can still be beneficial. If you need real parallelism in CPU-heavy operations, consider using multiprocessing or external tools like NumPy or C extensions.

99. How do you handle memory management in Python?

Python handles memory management automatically using a technique called **garbage collection.** The Python interpreter keeps track of objects and their references using a reference counting system. When an object's reference count drops to zero, it is deleted automatically.

In addition to reference counting, Python has a **cyclic garbage collector** to clean up objects involved in reference cycles. You can also manage memory manually using the `gc` module, but usually, it's not needed.

Example:

```
import gc
gc.collect()  # Triggers garbage collection
```

Good practices like avoiding unnecessary global variables and closing files and connections help manage memory better in Python.

100. What is the `with` statement in Python and why is it used?

The `with` statement in Python is used for **resource management**, such as working with files or network connections. It ensures that resources are properly closed after they are used, even if an error occurs during their use.

Example:

```
with open("file.txt", "r") as file:
    data = file.read()
```

Here, `file` is automatically closed when the block ends. This is better than manually calling `file.close()` because it handles exceptions safely.

The `with` statement works with context managers, which define `__enter__()` and `__exit__()` methods. It makes your code cleaner, safer, and easier to maintain.

I’ve collected all this information from various reliable sources, articles, and websites to bring you a comprehensive list of Python interview questions and answers.

While I’ve tried my best to ensure everything is accurate and helpful, there’s always a chance that something might need correction or improvement. If you find anything incorrect or think something can be better explained, please don’t hesitate to let us know in the comments section.

Your feedback will help make this resource more accurate and useful for everyone preparing for their Python interviews. Let’s grow and learn together! 💻🐍✨

If you got something wrong? Mention it in the comments. I would love to improve. your support means a lot to me! If you enjoy the content, I'd be grateful if you could consider subscribing to my YouTube channel as well.

I am Shirsh Shukla, a creative Developer, and a Technology lover. You can find me on LinkedIn or maybe follow me on Twitter or just walk over my portfolio for more details. And of course, you can follow me on GitHub as well.

Have a nice day!🙂

Pay SHIRSH SHUKLA using PayPal.Me

Go to paypal.me/shirsh94 and type in the amount. Since it's PayPal, it's easy and secure. Don't have a PayPal account...

www.paypal.com

Pay SHIRSH SHUKLA Using UPI

To pay via UPI, link your bank account to an app like Google Pay, PhonePe, or Paytm. Then, pay to Shirsh Shukla by entering his UPI ID or scanning his QR code, input the amount, and authenticate with your UPI PIN. The transaction will be processed instantly, and you'll receive a confirmation notification.

UPI ID: shirsh.dollop@okhdfcbank

Scan to pay with any UPI app

https://drive.google.com/file/d/1hdC-E7Kf97NM3YzWKvpm5olb89kNrlcs/view

Python · Python Programming · Interview · Interview Questions · Interview Preparation

Written by Shirsh Shukla

1.92K followers · 5 following

Follow

SDE at Reliance Jio | Mobile Application Developer | Speaker | Technical Writer | community member at Stack Overflow | Organizer @FlutterIndore

Responses (2)

Hlgsagar

What are your thoughts?

See all responses