

Python Coding Interview Questions With Answers - 2025

1. What is Python?

Python is an interpreted, high-level, general-purpose programming language. It emphasizes code readability with its use of significant indentation. Python supports multiple programming paradigms, including structured (particularly procedural), object-oriented, and functional programming.

2. How is Python interpreted?

Python code is executed line by line at runtime. Python internally converts the source code into an intermediate form called bytecode, which is then executed by the Python virtual machine (PVM).

3. What are Python's key features?

- Easy to learn and use
 - Interpreted language
 - Dynamically typed
 - Extensive libraries
 - Object-oriented
 - Portable
-

4. What is PEP 8?

PEP 8 is the Python Enhancement Proposal that provides guidelines and best practices for writing Python code. It covers various aspects such as naming conventions, code layout, and indentation.

5. How do you manage memory in Python?

Python uses automatic memory management and a garbage collector to handle memory. The garbage collector recycles memory when objects are no longer in use.

6. What are Python's data types?

- 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`
-

7. What is the difference between a list and a tuple?

- List: Mutable, can be changed after creation.
 - Tuple: Immutable, cannot be changed after creation.
-

8. How do you handle exceptions in Python?

Using the `try-except` block:

```
try:
    # code that may raise an exception
except SomeException as e:
    # code to handle the exception
```

9. Converting an Integer into Decimals

```
import decimal
integer = 10
print(decimal.Decimal(integer))
print(type(decimal.Decimal(integer)))
```

```
> 10
> <class 'decimal.Decimal'>
```

10. Converting a String of Integers into Decimals

```
import decimal
string = '12345'
print(decimal.Decimal(string))
print(type(decimal.Decimal(string)))
```

```
> 12345
> <class 'decimal.Decimal'>
```

11. Reversing a String using an Extended Slicing Technique

```
string = "Python Programming"
print(string[::-1])
```

```
> gnimmargorP nohtyP
```

12. Counting Vowels in a Given Word

```
vowel = ['a', 'e', 'i', 'o', 'u']
word = "programming"
count = 0
for character in word:
    if character in vowel:
        count += 1
print(count)
```

```
> 3
```

13. Counting Consonants in a Given Word

```
vowel = ['a', 'e', 'i', 'o', 'u']
word = "programming"
count = 0
for character in word:
    if character not in vowel:
        count += 1
print(count)

> 8
```

14. Counting the Number of Occurrences of a Character in a String

```
word = "python"
character = "p"
count = 0
for letter in word:
    if letter == character:
        count += 1
print(count)

> 1
```

15. Writing Fibonacci Series

```
fib = [0,1]
# Range starts from 0 by default
for i in range(5):
    fib.append(fib[-1] + fib[-2])

# Converting the list of integers to string
print(', '.join(str(e) for e in fib))

> 0, 1, 1, 2, 3, 5, 8
```

16. Finding the Maximum Number in a List

```
numberList = [15, 85, 35, 89, 125]
```

```
maxNum = numberList[0]
```

```
for num in numberList:
```

```
    if maxNum < num:
```

```
        maxNum = num
```

```
print(maxNum)
```

```
> 125
```

17. Finding the Minimum Number in a List

```
numberList = [15, 85, 35, 89, 125, 2]
```

```
minNum = numberList[0]
```

```
for num in numberList:
```

```
    if minNum > num:
```

```
        minNum = num
```

```
print(minNum)
```

```
> 2
```

18. Finding the Middle Element in a List

```
numList = [1, 2, 3, 4, 5]
```

```
midElement = int((len(numList)/2))
```

```
print(numList[midElement])
```

```
> 3
```

19. Converting a List into a String

```
lst = ["P", "Y", "T", "H", "O", "N"]
string = ''.join(lst)

print(string)
print(type(string))

> PYTHON
> <class 'str'>
```

20. Adding Two List Elements Together

```
lst1 = [1, 2, 3]
lst2 = [4, 5, 6]

res_lst = []
for i in range(0, len(lst1)):
    res_lst.append(lst1[i] + lst2[i])
print(res_lst)

> [5, 7, 9]
```

21. Comparing Two Strings for Anagrams

```
str1 = "Listen"
str2 = "Silent"

str1 = list(str1.upper())
str2 = list(str2.upper())
str1.sort(), str2.sort()

if(str1 == str2):
    print("True")
else:
    print("False")

> True
```

22. Checking for Palindrome Using Extended Slicing Technique

```
str1 = "Kayak".lower()
str2 = "kayak".lower()

if(str1 == str2[::-1]):
    print("True")
else:
    print("False")

> True
```

23. Counting the White Spaces in a String

```
string = "P r ogramm in g "
print(string.count(' '))

> 5
```

24. Counting Digits, Letters, and Spaces in a String

```
# Importing Regular Expressions Library
import re

name = 'Python is 1'

digitCount = re.sub("[^0-9]", "", name)
letterCount = re.sub("[^a-zA-Z]", "", name)
spaceCount = re.findall("[ \n]", name)

print(len(digitCount))
print(len(letterCount))
print(len(spaceCount))

> 1
```

```
> 8
> 2
```

25. Counting Special Characters in a String

```
# Importing Regular Expressions Library
import re
spChar = "!@#$$%^&*()"

count = re.sub('[\w]+', '', spChar)
print(len(count))

> 10
```

26. Removing All Whitespace in a String

```
import re

string = "C O D E"
spaces = re.compile(r'\s+')
result = re.sub(spaces, '', string)
print(result)

> CODE
```

27. Building a Pyramid in Python

```
floors = 3
h = 2*floors-1
for i in range(1, 2*floors, 2):
    print('{:^{}}'.format('*'*i, h))

> *
  ***
*****
```

28. Randomizing the Items of a List in Python

```
from random import shuffle

lst = ['Python', 'is', 'Easy']
shuffle(lst)
print(lst)

> ['Easy', 'is', 'Python']
```

29. Find the Largest Element in a List

```
def find_largest_element(lst):
    return max(lst)

# Example usage:
print(find_largest_element([1, 2, 3, 4, 5]))

> 5
```

30. Remove Duplicates from a List

```
def remove_duplicates(lst):
    return list(set(lst))

# Example usage:
print(remove_duplicates([1, 2, 2, 3, 4, 4, 5]))

> [1, 2, 3, 4, 5]
```

31. Factorial of a Number

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

```
# Example usage:  
print(factorial(5))
```

```
> 120
```

32. Merge Two Sorted Lists

```
def merge_sorted_lists(lst1, lst2):  
    return sorted(lst1 + lst2)
```

```
# Example usage:  
print(merge_sorted_lists([1, 3, 5], [2, 4, 6]))
```

```
> [1, 2, 3, 4, 5, 6]
```

33. Find the First Non-Repeating Character

```
def first_non_repeating_character(s):  
    for i in s:  
        if s.count(i) == 1:  
            return i  
    return None
```

```
# Example usage:  
print(first_non_repeating_character("swiss"))
```

```
> w
```

Advanced Level

In this section, we dive into more complex Python topics, aimed at those looking to deepen their expertise. You'll tackle:

- Advanced Data Structures: Work with sets, tuples, and more complex data types.
- Recursion and Iteration: Solve problems using advanced looping and recursive techniques.
- Algorithms: Explore sorting, searching, and optimization algorithms in depth.
- Modules and Libraries: Leverage powerful Python libraries for various applications.
- Error Handling and Debugging: Learn to write robust code with effective error handling and debugging techniques.
- Web Scraping and APIs: Gain experience in extracting data from websites and interacting with APIs.
- Advanced Object-Oriented Programming (OOP): Master OOP concepts like inheritance, polymorphism, and design patterns.

This level will challenge you to think critically and apply your knowledge to complex problems, preparing you for high-level Python applications and interviews.

34. What are Python metaclasses?

Metaclasses are classes of classes that define how classes behave. A class is an instance of a metaclass. They allow customization of class creation.

35. Explain the difference between `is` and `==`.

`is` : Checks if two references point to the same object.

`==` : Checks if the values of two objects are equal.

36. How does Python's memory management work?

Python uses reference counting and garbage collection. Objects with a reference count of zero are automatically cleaned up by the garbage collector.

37. What is the purpose of Python's `with` statement?

The `with` statement simplifies exception handling by encapsulating common preparation and cleanup tasks in so-called context managers.

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

38. What are Python's `@staticmethod` and `@classmethod`?

`@staticmethod` : Defines a method that does not operate on an instance or class; no access to self or cls.

`@classmethod` : Defines a method that operates on the class itself; it receives the class as an implicit first argument (cls).

39. How do you implement a singleton pattern in Python?

Using a metaclass

```
class Singleton(type):
    _instances = {}
    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            cls._instances[cls] = super(Singleton, cls).__call__(*args, *
            return cls._instances[cls]

class MyClass(metaclass=Singleton):
    pass
```

40. Explain Python's garbage collection mechanism

Python uses a garbage collection mechanism based on reference counting and a cyclic garbage collector to detect and collect cycles (groups of objects that reference each other but are not accessible from any other object).

41. What are Python's magic methods?

Magic methods (or dunder methods) are special methods with double underscores at the beginning and end. They enable the customization of behavior for standard operations

- `__init__` : Constructor
 - `__str__` : String representation
 - `__add__` : Addition operator
-

42. How do you handle multi-threading in Python?

Using the `threading` module:

```
import threading

def print_numbers():
    for i in range(10):
        print(i)

thread = threading.Thread(target=print_numbers)
thread.start()
thread.join()
```

43. What are Python's coroutine functions?

Coroutines are a type of function that can pause and resume their execution. They are defined with `async def` and use `await` to yield control back to the event loop.

```
import asyncio

async def say_hello():
    await asyncio.sleep(1)
    print("Hello")

asyncio.run(say_hello())
```

44. Explain Python's Global Interpreter Lock (GIL). How does it affect multithreading?

The Global Interpreter Lock (GIL) is a mutex that protects access to Python objects, preventing multiple native threads from executing Python bytecodes simultaneously in CPython. This lock simplifies memory management and ensures thread safety, but it limits the performance of multi-threaded Python programs by allowing only one thread to execute at a time. As a result, Python multithreading is more suitable for I/O-bound tasks than CPU-bound tasks. Multiprocessing or other Python implementations, like Jython or IronPython, may be preferred for CPU-bound tasks.

45. What are metaclasses in Python, and how are they used?

A metaclass in Python is a class of a class that defines how a class behaves. Classes themselves are instances of metaclasses. You can customize class creation by defining a metaclass, such as modifying class properties, adding methods, or implementing design patterns. A common use case for metaclasses is enforcing coding standards or design patterns, such as singleton, or auto-registering classes.

46. Can you explain the difference between `deepcopy` and `copy` in Python?

The `copy` module in Python provides two methods: `copy()` and `deepcopy()`.

- `copy.copy()` creates a shallow copy of an object. It copies the object's structure but not the elements themselves, meaning it only copies references for mutable objects.
 - `copy.deepcopy()` creates a deep copy of the object, including recursively copying all objects contained within the original object. Changes made to the deep-copied object do not affect the original object.
-

47. Describe Python's `__slots__` and its benefits.

`__slots__` is a special attribute in Python that allows you to explicitly declare data members (slots) and prevent the creation of `__dict__`, thereby reducing memory overhead. By using `__slots__`, you can limit the attributes of a class to a fixed set of fields and reduce the per-instance memory consumption. This is particularly beneficial when creating a large number of instances of a class.

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

- `is` checks for identity, meaning it returns `True` if two references point to the same object in memory.
- `==` checks for equality, meaning it returns `True` if the values of the objects are equal, even if they are different objects in memory.

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

```
print(a is b)  # True, because both a and b refer to the same object
print(a is c)  # False, because a and c refer to different objects
print(a == c)  # True, because a and c have the same values
```

49. Find the Longest Consecutive Sequence in an Unsorted List

Problem: Given an unsorted list of integers, find the length of the longest consecutive sequence.

Example:

```
Input: [100, 4, 200, 1, 3, 2]
Output: 4  # (Sequence: [1, 2, 3, 4])
```

Solution:

```
def longest_consecutive(nums):
    num_set = set(nums)  # Convert list to set for O(1) lookups
    longest = 0
```

```

for num in num_set:
    if num - 1 not in num_set: # Start of a new sequence
        length = 1
        while num + length in num_set:
            length += 1
        longest = max(longest, length)

return longest

# Example usage:
print(longest_consecutive([100, 4, 200, 1, 3, 2])) # Output: 4

```

Complexity: $O(N)$, where N is the number of elements.

Most Frequently Asked Questions by Big Tech Companies

As we gear up for technical interviews, it's essential to be prepared for the toughest Python questions. If you're preparing for high-level technical interviews, you'll want to master these complex Python problems. Here's a list of coding questions that's been asked by top-tier companies!

1: Find the Longest Substring Without Repeating Characters

Given a string, find the length of the longest substring without repeating characters.

Answer

```

def length_of_longest_substring(s: str) -> int:
    char_set = set()
    left = 0
    max_len = 0

    for right in range(len(s)):
        while s[right] in char_set:
            char_set.remove(s[left])

```



```

        left += 1
        char_set.add(s[right])
        max_len = max(max_len, right - left + 1)

    return max_len

# Example usage:
input_str = "abcabcbb"
print(length_of_longest_substring(input_str)) # Output: 3

```

Explanation

This approach uses a sliding window and a set to keep track of characters. The left pointer moves when a duplicate character is found, ensuring all characters in the window are unique.

Asked by: Facebook

2. Find the Kth Smallest Element in an Unsorted Array

Given an unsorted array, find the Kth smallest element in the array.

Answer

```

import heapq

def kth_smallest(nums, k):
    return heapq.nsmallest(k, nums)[-1]

# Example usage:
nums = [3, 2, 1, 5, 6, 4]
k = 2
print(kth_smallest(nums, k)) # Output: 2

```

Explanation

Using Python's `heapq.nsmallest()` function efficiently finds the Kth smallest element by retrieving the first K elements from the list and then returning the last element of that list.

Asked by: Apple

3. Merge Intervals

Given a collection of intervals, merge all overlapping intervals.

Answer:

```
def merge_intervals(intervals):
    intervals.sort(key=lambda x: x[0])
    merged = []

    for interval in intervals:
        if not merged or merged[-1][1] < interval[0]:
            merged.append(interval)
        else:
            merged[-1][1] = max(merged[-1][1], interval[1])

    return merged

# Example usage:
intervals = [[1,3],[2,6],[8,10],[15,18]]
print(merge_intervals(intervals)) # Output: [[1, 6], [8, 10], [15, 18]]
```

Explanation

First, we sort intervals by the starting point. Then, we iterate through and merge intervals if they overlap.

Asked by: Microsoft

4. Count Distinct Substrings in a String

Given a string, find the number of distinct substrings in the string.

Answer

```
def count_distinct_substrings(s: str) -> int:
    n = len(s)
    substrings = set()

    for i in range(n):
        for j in range(i+1, n+1):
```

```
        substrings.add(s[i:j])

    return len(substrings)

# Example usage:
input_str = "abc"
print(count_distinct_substrings(input_str)) # Output: 6
```

Explanation

This solution generates all possible substrings and stores them in a set, ensuring uniqueness.

Asked by: Netflix