100 Python interview questions and answers

Basic Python Interview Questions

1. **What is Python?**
    - Python is an interpreted, high-level, dynamically typed programming language known for its simplicity and readability.
2. **What are Python's key features?**
    - Open-source, dynamically typed, interpreted, easy-to-learn, extensive library support, object-oriented.
3. **What is PEP 8?**
    - PEP 8 is the Python Enhancement Proposal that provides guidelines for writing clean, readable Python code.
4. **What is Python used for?**
    - Web development, data analysis, AI/ML, automation, scripting, game development.
5. **How is Python different from other programming languages?**
    - It has simple syntax, is dynamically typed, and has automatic memory management.
6. **What are Python's built-in data types?**
    - `int`, `float`, `str`, `list`, `tuple`, `dict`, `set`, `bool`.
7. **What is the difference between a list and a tuple?**
    - Lists are mutable, while tuples are immutable.
8. **What are Python's built-in data structures?**
    - Lists, tuples, sets, dictionaries.
9. **What is the difference between `is` and `==`?**
    - `is` checks identity (memory location), `==` checks value equality.
10. **What is a Python dictionary?**
    - A dictionary is an unordered collection of key-value pairs.

Intermediate Python Interview Questions

11. **What are Python functions?**
    - A function is a reusable block of code that performs a specific task.
12. **What is the difference between `return` and `yield`?**
    - `return` stops function execution; `yield` allows the function to continue execution (used in generators).
13. **What is a lambda function?**
    - An anonymous function defined using `lambda` keyword.
14. **What is a decorator in Python?**
    - A decorator modifies the behavior of a function or class method.
15. **What are Python modules and packages?**
    - A module is a single `.py` file; a package is a collection of modules.
16. **What is the difference between `deepcopy()` and `copy()`?**
    - `copy()` creates a shallow copy; `deepcopy()` creates an independent copy of nested objects.
17. **What is the difference between `*args` and `**kwargs`?**
    - `*args` allows passing variable-length positional arguments; `**kwargs` allows passing variable-length keyword arguments.
18. **How does Python handle memory management?**
    - Python uses automatic garbage collection and reference counting.
19. **What is Python's garbage collection?**
    - It automatically deallocates unused memory.
20. **What are Python's scope rules?**
    - LEGB (Local, Enclosing, Global, Built-in).

Advanced Python Interview Questions

21. **What is multithreading in Python?**
    - Running multiple threads in parallel within a single process.
22. **What is the Global Interpreter Lock (GIL)?**
    - A mutex that allows only one thread to execute Python bytecode at a time.
23. **What is multiprocessing in Python?**
    - Running multiple processes in parallel to bypass GIL.

24. **What are metaclasses in Python?**
    - A metaclass controls how classes are created.
25. **What is duck typing in Python?**
    - A concept where type checking is done at runtime based on behavior.
26. **What are Python magic methods?**
    - Special methods like `__init__`, `__str__`, `__len__`, `__getitem__`.
27. **What is the difference between `classmethod`, `staticmethod`, and instance methods?**
    - `classmethod` modifies class state, `staticmethod` has no access to instance or class state, instance methods work on object instances.
28. **What is the `with` statement used for?**
    - It ensures proper resource management (e.g., file handling).
29. **How can you improve Python performance?**
    - Use optimized libraries, generators, caching, and multiprocessing.
30. **What are Python comprehensions?**
    - List, dictionary, and set comprehensions for concise looping.

OOP in Python

31. **What are Python classes and objects?**
    - Classes are blueprints; objects are instances.
32. **What is inheritance in Python?**
    - A mechanism where one class derives properties from another.
33. **What is multiple inheritance?**
    - A class inheriting from multiple parent classes.
34. **What is encapsulation in Python?**
    - Restricting access to certain details of an object.
35. **What is polymorphism?**
    - Using the same function name for different types of objects.
36. **What is method overriding?**
    - Redefining a method in a subclass.

37. **What is method overloading?**
    - Python does not support true method overloading but achieves it using default arguments.
38. **What is the difference between `super()` and direct parent class calls?**
    - `super()` calls the parent method in a dynamic way.
39. **What is an abstract class in Python?**
    - A class that cannot be instantiated and has abstract methods.
40. **What are mixins in Python?**
    - A way to add reusable functionality to classes without inheritance.

Python Libraries & Frameworks

41. **What is NumPy used for?**
    - Numerical computing.
42. **What is Pandas used for?**
    - Data analysis and manipulation.
43. **What is Flask/Django?**
    - Flask is a micro web framework; Django is a full-stack web framework.
44. **What is TensorFlow?**
    - A deep learning framework.
45. **What is Matplotlib?**
    - A plotting library.
46. **What is Scikit-learn?**
    - A machine learning library.
47. **What is BeautifulSoup?**
    - A web scraping library.
48. **What is SQLAlchemy?**
    - A database ORM.
49. **What is Selenium?**
    - A web automation library.
50. **What is PyTorch?**
    - A deep learning framework.

Python Coding Questions

51. **Reverse a string in Python.**

```
def reverse_string(s):
    return s[::-1]
```

52. **Find the factorial of a number.**

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

53. **Check if a number is prime.**

```
def is_prime(n):
    if n < 2: return False
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0: return False
    return True
```


54. **Find the largest element in a list.**

```
max(lst)
```

55. **Check if a string is a palindrome.**

```
def is_palindrome(s):
    return s == s[::-1]
```

Here are **questions 55-100** to complete the list of 100 Python interview questions.

More Python Coding Questions

55. Write a Python program to find the Fibonacci sequence up to n numbers.

```
def fibonacci(n):

    a, b = 0, 1

    for _ in range(n):
```

```
        print(a, end=" ")
        a, b = b, a + b
```

56. Find the sum of digits of a number.

```
def sum_of_digits(n):
    return sum(int(digit) for digit in str(n))
```

57. Find the greatest common divisor (GCD) of two numbers.

```
import math
def gcd(a, b):
    return math.gcd(a, b)
```

58. Find the least common multiple (LCM) of two numbers.

```
def lcm(a, b):
    return (a * b) // math.gcd(a, b)
```


59. Count occurrences of a character in a string.

```
def char_count(s, char):
    return s.count(char)
```

60. Find the most frequent element in a list.

```
from collections import Counter
def most_frequent(lst):
    return Counter(lst).most_common(1)[0][0]
```

61. Merge two sorted lists into one sorted list.

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

62. Remove duplicates from a list.

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

63. Find the second largest number in a list.

```
def second_largest(lst):
    return sorted(set(lst))[-2]
```

64. Check if two strings are anagrams.

```
def is_anagram(s1, s2):
    return sorted(s1) == sorted(s2)
```


65. Find the missing number in an array from 1 to n.

```
def missing_number(lst, n):
    return sum(range(1, n + 1)) - sum(lst)
```

66. Find the intersection of two lists.

```
def list_intersection(lst1, lst2):
    return list(set(lst1) & set(lst2))
```

67. Find the union of two lists.

```
def list_union(lst1, lst2):
    return list(set(lst1) | set(lst2))
```

68. Find all pairs in a list that sum up to a given number.

```
def find_pairs(lst, target):
    return [(x, y) for x in lst for y in lst if x + y == target]
```

69. Find duplicate elements in a list.

```
def find_duplicates(lst):
    return list(set([x for x in lst if lst.count(x) > 1]))
```

70. Convert a string to title case.

```
def to_title_case(s):
    return s.title()
```


71. Reverse words in a sentence.

```
def reverse_words(sentence):
    return " ".join(sentence.split()[::-1])
```

72. Implement a stack using a list.

```
class Stack:
    def __init__(self):
        self.stack = []

    def push(self, item):
        self.stack.append(item)

    def pop(self):
```

```
        return self.stack.pop() if self.stack else None
```

73. Implement a queue using a list.

```
class Queue:
    def __init__(self):
        self.queue = []

    def enqueue(self, item):
        self.queue.append(item)

    def dequeue(self):
        return self.queue.pop(0) if self.queue else None
```


74. Find the longest word in a string.

```
def longest_word(s):
    return max(s.split(), key=len)
```

75. Check if a number is an Armstrong number.

```
def is_armstrong(n):
    return n == sum(int(digit)**len(str(n)) for digit in str(n))
```

76. Find the first non-repeating character in a string.

```
def first_non_repeating(s):
    for char in s:
        if s.count(char) == 1:
            return char
```

```
    return None
```

77. Find all permutations of a string.

```
from itertools import permutations

def string_permutations(s):
    return ["".join(p) for p in permutations(s)]
```

78. Generate all subsets of a list.

```
from itertools import combinations

def subsets(lst):
    return [list(combinations(lst, i)) for i in range(len(lst) + 1)]
```

79. Find the longest common prefix among a list of strings.

```
import os

def longest_common_prefix(lst):
    return os.path.commonprefix(lst)
```

80. Check if a list is sorted.

```
def is_sorted(lst):
    return lst == sorted(lst)
```

More Advanced Python Questions

81. What is the difference between `@staticmethod` and `@classmethod`?

- `@staticmethod` doesn't access instance or class attributes, whereas `@classmethod` takes `cls` as a parameter and can modify class-level attributes.

82. What is the purpose of the `self` keyword in Python?

- It represents the instance of a class and allows access to instance attributes.

83. How does exception handling work in Python?

```
try:
    x = 10 / 0
except ZeroDivisionError as e:
    print(f"Error: {e}")
finally:
    print("Execution complete")
```

84. What are Python iterators and generators?

- Iterators are objects that implement `__iter__()` and `__next__()` methods.
- Generators use `yield` and remember their state.

85. Explain the difference between shallow copy and deep copy.

- Shallow copy copies only references; deep copy duplicates all elements.

86. What is monkey patching in Python?

- Modifying a module or class at runtime.

87. What is the difference between `open()` modes `r`, `w`, `a`, and `rb`?

- `r`: Read, `w`: Write (overwrites), `a`: Append, `rb`: Read binary.

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

- `==` checks value equality, `is` checks object identity.

89. How to implement memoization in Python?

```
from functools import lru_cache
@lru_cache(maxsize=None)
def fib(n):
    return n if n < 2 else fib(n - 1) + fib(n - 2)
```

90. How to implement a linked list in Python?

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None
```

91. What is method resolution order (MRO)?

- It defines the order in which methods are inherited.

92. Explain heapq in Python.

- It provides a heap queue (priority queue) implementation.

93. How to implement a binary search algorithm?

```
def binary_search(lst, target):
    low, high = 0, len(lst) - 1
    while low <= high:
        mid = (low + high) // 2
        if lst[mid] == target:
            return mid
        elif lst[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
```

Here are **questions 95-100** with their answers.

---

95. What is Python type hinting, and how does it work?

Python type hinting allows developers to specify expected data types for function arguments and return values. It improves code readability and helps catch type-related errors. However, type hints are not enforced at runtime.

Example:

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

Here, `a` and `b` are expected to be integers, and the function returns an integer.

---

96. What are context managers in Python, and how does the `with` statement work?

Context managers allow resource management using the `with` statement, ensuring proper cleanup after use.

Example:

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

- The `with` statement ensures the file is closed automatically after execution.
- The `open()` function implements a context manager using `__enter__()` and `__exit__()` methods.

Custom Context Manager:

```
class MyContext:
    def __enter__(self):
        print("Entering context")
        return self

    def __exit__(self, exc_type, exc_value, traceback):
        print("Exiting context")

with MyContext() as ctx:
```

```
print("Inside context")
```

---

97. How does `collections.defaultdict` work in Python?

`defaultdict` is a subclass of Python's `dict` that provides a default value for missing keys, avoiding `KeyError`.

Example:

```
from collections import defaultdict

# Default factory function returns 0 if key is not found
numbers = defaultdict(int)
numbers["a"] += 1
print(numbers["a"])  # Output: 1
print(numbers["b"])  # Output: 0 (instead of KeyError)

# Default factory for lists
words = defaultdict(list)
words["fruits"].append("apple")
print(words)  # Output: {'fruits': ['apple']}
```


- Without `defaultdict`, accessing a non-existent key in a regular dictionary raises a `KeyError`.

---

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

- `copy.copy()` creates a **shallow copy**, meaning it copies references instead of creating new objects.
- `copy.deepcopy()` creates a **deep copy**, meaning it recursively copies all objects and creates independent duplicates.

Example:

```
import copy

original = [[1, 2], [3, 4]]
shallow_copy = copy.copy(original)
deep_copy = copy.deepcopy(original)

shallow_copy[0][0] = 99
print(original)      # Output: [[99, 2], [3, 4]] (shallow copy affected original)
print(deep_copy)     # Output: [[1, 2], [3, 4]] (deep copy remains unchanged)
```

- **Shallow Copy**: Changes in nested objects affect the original.
- **Deep Copy**: Creates a fully independent copy.

---

99. How does Python handle memory management (reference counting & garbage collection)?

Python uses **automatic memory management**, which includes:

1. **Reference Counting**: Each object has a reference count. When the count drops to zero, the object is deleted.
2. **Garbage Collection (GC)**: Python's garbage collector removes circular references (e.g., objects referring to each other).

Reference Counting Example:

```
import sys

x = [1, 2, 3]
print(sys.getrefcount(x))  # Output: Reference count (varies)
y = x
print(sys.getrefcount(x))  # Reference count increases
del x
print(sys.getrefcount(y))  # Still exists because of 'y'
```


Garbage Collection Example:

```
import gc

class A:
    def __init__(self):
        print("Object created")
    def __del__(self):
        print("Object deleted")

obj = A()
del obj  # Reference count reaches 0, so __del__() is called

gc.collect()  # Forces garbage collection
```

- **Python runs GC periodically** to free up memory occupied by objects in cycles.
- You can manually trigger GC using `gc.collect()`.

---

100. How to optimize memory usage in Python?

1. **Use Generators Instead of Lists**
    - Instead of storing large datasets in lists, use generators (`yield`) to generate values on demand.

```
def large_numbers():
    for i in range(1000000):
        yield i
gen = large_numbers()  # Consumes less memory than list(range(1000000))
```

2.
3. **Use `__slots__` in Classes**
    - Reduces memory overhead by restricting attributes.

```
class MyClass:
    __slots__ = ['name', 'age']  # Saves memory by preventing dictionary allocation
```

4.

Use `del` to Free Unused Variables

```
 x = [1, 2, 3]
del x  # Removes reference
```

5.
6. **Avoid Unnecessary Object Creation**
    - Reuse objects where possible.

Enable Garbage Collection When Needed

```
 import gc
gc.collect()  # Forces memory cleanup
```