Goldman Sachs

Python Developer Interview Questions

0-3 YOE

22LPA

Basic Python Concepts:

1. Key Differences Between Python 2 and Python 3

Python 2 and Python 3 are two major versions of Python, but Python 2 is no longer maintained. Some key differences include:

Feature	Python 2	Python 3
Print Statement	print "Hello" (without parentheses)	print("Hello") (with parentheses)
Integer Division	5/2 returns 2 (integer division by default)	5/2 returns 2.5 (true division)
Unicode Support	Strings are ASCII by default	Strings are Unicode by default (str type)
xrange()	Used xrange() for memory-efficient looping	range() now behaves like xrange()

Feature	Python 2	Python 3
Input Function	raw_input() for strings, input() for integers	input() takes all input as strings
Libraries	Some libraries support Python 2 only	All new development is for Python 3

Example:

```
# Python 2 (Not supported anymore)
print "Hello, World"  # No parentheses


# Python 3
print("Hello, World")  # Requires parentheses
```

---

2. What Are Python Decorators and How Do They Work?

A **decorator** is a function that modifies the behavior of another function without changing its structure. It is often used for logging, authentication, and timing functions.

Example: Logging with a Decorator

```
def decorator_function(original_function):
    def wrapper_function():
        print(f"Executing {original_function.__name__}")
        return original_function()
    return wrapper_function

@decorator_function  # This is how decorators are applied
def say_hello():
    print("Hello!")
```

say_hello()

Output:

Executing say_hello

Hello!

Here, decorator_function wraps say_hello() and adds extra functionality.

---

3. Difference Between a List and a Tuple

Feature	List (list)	Tuple (tuple)
Mutability	Mutable (can be changed)	Immutable (cannot be changed)
Performance	Slower due to dynamic resizing	Faster due to fixed size
Syntax	Defined using []	Defined using ()
Use Case	When data needs modification	When data should remain constant

Example:

```
# List (Mutable)
my_list = [1, 2, 3]
my_list[0] = 10  # Allowed
print(my_list)  # [10, 2, 3]


# Tuple (Immutable)
my_tuple = (1, 2, 3)
# my_tuple[0] = 10  # Error: TypeError: 'tuple' object does not support item assignment
```

Lists are preferred when frequent modifications are needed, whereas tuples are used for fixed data.

---

4. What is a Lambda Function?

A **lambda function** is an anonymous function in Python that can have multiple arguments but only one expression.

Syntax:

lambda arguments: expression

Example:

```
# Regular Function
def square(x):
    return x * x


print(square(5))  # Output: 25


# Lambda Equivalent
square_lambda = lambda x: x * x
print(square_lambda(5))  # Output: 25
```

Lambda functions are often used in map(), filter(), and sorted() operations.

Example with sorted():

```
data = [(1, "Alice"), (3, "Bob"), (2, "Charlie")]
sorted_data = sorted(data, key=lambda x: x[0])
print(sorted_data)  # [(1, "Alice"), (2, "Charlie"), (3, "Bob")]
```

---

5. What is a Python Generator and How is it Different from a Normal Function?

A **generator** is a special type of iterator that produces values **lazily**, meaning it generates values on the fly instead of storing them in memory. It is defined using the yield keyword.

Key Differences Between Generators and Normal Functions

Feature	Normal Function	Generator Function
Memory Usage	Stores all values in memory	Generates values one by one (efficient)
Execution	Executes completely when called	Saves state and resumes from yield
Return Statement	Uses return	Uses yield

Example: Normal Function vs. Generator

```
# Normal function (stores all values)
def square_list(n):
    result = []
    for i in range(n):
        result.append(i * i)
    return result

print(square_list(5))  # [0, 1, 4, 9, 16]

# Generator function (yields one value at a time)
def square_generator(n):
    for i in range(n):
        yield i * i

gen = square_generator(5)
print(next(gen))  # Output: 0
print(next(gen))  # Output: 1
print(next(gen))  # Output: 4
```

Using yield allows generators to be **memory efficient** when dealing with large datasets.

---

6. How Does Python Handle Memory Management?

Python uses **automatic memory management** and relies on a **garbage collector** to free up unused memory. The key components of Python's memory management system are:

Key Aspects of Python Memory Management

1. **Reference Counting:**
    - Every object in Python has a reference count (i.e., the number of variables pointing to it).
    - When the reference count drops to zero, Python automatically deletes the object.

Example:

```
import sys


a = [1, 2, 3]

print(sys.getrefcount(a))  # Output: 2 (one reference from 'a' and another from function call)

b = a  # Another reference to the same object

print(sys.getrefcount(a))  # Output: 3


del a  # Deletes one reference

print(sys.getrefcount(b))  # Output: 2
```

2. **Garbage Collection:**
    - Python uses a garbage collector to clean up cyclic references (e.g., objects referring to each other).
    - The gc module can be used to manually trigger garbage collection.

Example:

```
import gc
```

gc.collect() # Force garbage collection

3. **Memory Pools (Private Heap):**
    - Python does not release memory back to the OS immediately; it maintains a private heap for efficient allocation.
    - The **PyObject_Malloc()** function handles memory allocation internally.

Optimizing Memory Usage:

- Use **generators** instead of lists to handle large datasets efficiently.
- Use **slots** in classes to reduce memory usage.

---

7. What Is the Difference Between deepcopy() and copy() in Python?

The copy module in Python provides two ways to duplicate objects:

Feature	copy.copy() (Shallow Copy)	copy.deepcopy() (Deep Copy)
Definition	Creates a new object but references nested objects	Creates a completely independent copy
Nested Objects	Changes in nested objects affect both copies	Changes in nested objects do not affect the original
Performance	Faster	Slower (due to recursion)

Example:

```
import copy

# Original list with nested list
original_list = [[1, 2, 3], [4, 5, 6]]


# Shallow Copy
shallow_copy = copy.copy(original_list)
```

```
# Deep Copy
deep_copy = copy.deepcopy(original_list)

# Modify the nested list
original_list[0][0] = 99

print(shallow_copy)  # [[99, 2, 3], [4, 5, 6]]  (affected)
print(deep_copy)  # [[1, 2, 3], [4, 5, 6]]  (unchanged)
```

When to Use?

- Use **shallow copy** when the object has immutable elements or when deep copy isn't necessary.
- Use **deep copy** when working with nested data structures that require full duplication.

---

8. What Is a Python Module and Package?

Module:

- A **module** is a single Python file (.py) that contains functions, classes, and variables.
- It allows code reuse by importing it into other scripts.

Example of a Module (math_utils.py):

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

Importing the Module:

```
python
CopyEdit
import math_utils
```

```
print(math_utils.add(3, 5))  # Output: 8
```

Package:

- A **package** is a collection of modules organized in directories containing an __init__.py file.
- It helps in structuring large codebases.

Example of a Package:

markdown

CopyEdit

```
my_package/
├── __init__.py
├── module1.py
└── module2.py
```

Using a Package:

```
from my_package import module1
```

Key Differences Between Modules and Packages

Feature	Module	Package
Structure	A single .py file	A directory with multiple modules
Purpose	Organizes related functions/classes	Groups related modules
Importing	import module_name	from package_name import module_name

---

9. What Is the Purpose of the with Statement in Python?

The with statement is used for **resource management** (e.g., file handling, database connections) and ensures that resources are automatically closed after execution.

Why Use the with Statement?

- **Ensures Proper Cleanup:** Resources are automatically released (e.g., closing a file).

- **Prevents Memory Leaks:** Avoids forgetting to release resources.
- **Reduces Code Complexity:** No need for try-except-finally blocks.

Example: File Handling Without with

```
file = open("example.txt", "w")
try:
    file.write("Hello, World!")
finally:
    file.close()  # Must be manually closed
```

Example: File Handling With with

```
with open("example.txt", "w") as file:
    file.write("Hello, World!")  # File closes automatically
```

Here, the file is **automatically closed** once the with block exits.

Data Structures and Algorithms:

10. What Is a Dictionary in Python, and How Is It Different from a List?

A **dictionary** in Python is an **unordered collection** of key-value pairs, where keys are unique and immutable (e.g., strings, numbers, tuples). It allows for **fast lookups** and modifications.

Dictionary vs. List

Feature	Dictionary (dict)	List (list)
Structure	Key-value pairs ({key: value})	Ordered sequence of elements ([item1, item2])
Access Time	Fast (O(1) for key lookup using a hash table)	Slower (O(n) for searching an element)

Feature	Dictionary (dict)	List (list)
Order (Python 3.7+)	Maintains insertion order	Maintains order
Key/Index Type	Keys must be unique and immutable	Indices are integers starting from 0
Use Case	Fast lookups, structured data	Ordered collections, sequences

Example of a Dictionary

```
employee = {
   "name": "Alice",
   "age": 30,
   "role": "Data Analyst"
}


print(employee["name"])  # Output: Alice
```

Example of a List

```
fruits = ["apple", "banana", "cherry"]
print(fruits[1])  # Output: banana
```

When to Use?

- Use a **dictionary** when you need quick lookups based on unique identifiers (keys).
- Use a **list** when order matters and indexed access is required.

---

11. How Would You Reverse a String in Python?

There are multiple ways to reverse a string in Python:

1. Using String Slicing (Fastest and Easiest)

```
s = "Goldman"
reversed_s = s[::-1]
```

```
print(reversed_s)  # Output: "namdloG"
```

2. Using reversed() and join()

```
s = "Sachs"
reversed_s = "".join(reversed(s))
print(reversed_s)  # Output: "shcaS"
```

3. Using a Loop

```
s = "Python"
reversed_s = ""
for char in s:
    reversed_s = char + reversed_s  # Adding characters in reverse order
print(reversed_s)  # Output: "nohtyP"
```

4. Using Recursion

```
def reverse_string(s):
    if len(s) == 0:
        return s
    return s[-1] + reverse_string(s[:-1])

print(reverse_string("Finance"))  # Output: "ecnaniF"
```

Best Approach?

- **String slicing ([::-1])** is the most efficient method because it is optimized at the C level.
- **reversed() with join()** is also efficient but slightly less readable.
- **Loops and recursion** are less efficient and mainly used for demonstration purposes.

---

12. Explain the Difference Between is and == in Python.

Python provides two comparison operators:

Operator	Purpose	Example
== (Equality)	Compares **values**	x == y checks if values of x and y are the same
is (Identity)	Compares **memory locations**	x is y checks if x and y refer to the same object in memory

Example 1: Using == (Value Comparison)

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

b = [1, 2, 3]


print(a == b)  # True (because values are the same)

print(a is b)  # False (because they are different objects in memory)
```

Example 2: Using is (Identity Comparison)

```
x = "hello"

y = "hello"


print(x == y)  # True (values are the same)

print(x is y)  # True (Python optimizes immutable objects like strings and caches them)
```

Example 3: Identity in Mutability

```
list1 = [10, 20, 30]

list2 = list1  # Both point to the same object


print(list1 is list2)  # True

list1.append(40)

print(list2)  # Output: [10, 20, 30, 40] (because both refer to the same object)
```

Key Takeaways

- **Use ==** when comparing values.
- **Use is** when checking whether two variables reference the same object.

---

13. How Would You Remove Duplicates from a List in Python?

There are multiple ways to remove duplicates:

1. Using set() (Fastest but Unordered)

```
nums = [4, 2, 7, 4, 2, 8]
unique_nums = list(set(nums))
print(unique_nums)  # Output: [2, 4, 7, 8] (order may change)
```

- ⚠ **Limitation:** Order is not preserved.

2. Using dict.fromkeys() (Preserves Order)

```
nums = [4, 2, 7, 4, 2, 8]
unique_nums = list(dict.fromkeys(nums))
print(unique_nums)  # Output: [4, 2, 7, 8]
```

- ✅ **Advantage:** Maintains order (Python 3.7+).

3. Using a Loop (Manual Method)

```
nums = [4, 2, 7, 4, 2, 8]
unique_nums = []
for num in nums:
    if num not in unique_nums:
        unique_nums.append(num)
print(unique_nums)  # Output: [4, 2, 7, 8]
```

- ⚠ **Limitation:** Slower ($O(n^2)$ complexity).

4. Using List Comprehension + set()

```
nums = [4, 2, 7, 4, 2, 8]
```

```
seen = set()
unique_nums = [num for num in nums if num not in seen and not seen.add(num)]
print(unique_nums)  # Output: [4, 2, 7, 8]
```

- ✅ **Advantage:** Preserves order and is efficient.

Best Approach?

- Use **set()** for speed when order doesn't matter.
- Use **dict.fromkeys()** or **list comprehension** for order preservation.

---

14. How Do You Implement a Stack or Queue in Python?

A **stack** follows the **Last-In-First-Out (LIFO)** principle, while a **queue** follows the **First-In-First-Out (FIFO)** principle. Both can be implemented in Python using **lists** or the **collections.deque** module.

Stack Implementation (LIFO)

Using a Python list:

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

    def push(self, item):
        self.stack.append(item)  # Add to the top

    def pop(self):
        if not self.is_empty():
            return self.stack.pop()  # Remove from the top
        return "Stack is empty"
```

```
    def peek(self):
        if not self.is_empty():
            return self.stack[-1]  # Get the top element without removing it
        return "Stack is empty"

    def is_empty(self):
        return len(self.stack) == 0

# Usage
s = Stack()
s.push(10)
s.push(20)
print(s.pop())  # Output: 20
print(s.peek())  # Output: 10
```

- **Push:** O(1)
- **Pop:** O(1)

Queue Implementation (FIFO)

Using collections.deque (more efficient than a list for pop operations):

```
from collections import deque

class Queue:
    def __init__(self):
        self.queue = deque()

    def enqueue(self, item):
        self.queue.append(item)  # Add to the rear
```

```
    def dequeue(self):
        if not self.is_empty():
            return self.queue.popleft()  # Remove from the front
        return "Queue is empty"

    def is_empty(self):
        return len(self.queue) == 0

# Usage
q = Queue()
q.enqueue(10)
q.enqueue(20)
print(q.dequeue())  # Output: 10
```

- **Enqueue:** O(1)
- **Dequeue:** O(1)

📌 **Key Takeaway:**

- **Stack:** Use a **list** or deque for LIFO operations.
- **Queue:** Use deque or queue.Queue for FIFO operations (more efficient than list pop(0)).

---

15. How Would You Find the Most Common Element in a List?

There are multiple ways to find the most common element.

1. Using collections.Counter (Recommended)

```
from collections import Counter
```

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

counter = Counter(nums)

most_common = counter.most_common(1)[0][0]  # Returns most frequent element

print(most_common)  # Output: 3
```

- Counter(nums).most_common(1) returns a list of tuples [(3, 3)], so we extract the first element.

2. Using max() with count() (Inefficient for Large Lists)

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

most_common = max(set(nums), key=nums.count)

print(most_common)  # Output: 3
```

- **Time Complexity:** $O(n^2)$ because count() iterates over the list multiple times.

3. Using a Dictionary (Manual Counting)

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

freq = {}


for num in nums:
   freq[num] = freq.get(num, 0) + 1  # Increase count for each element


most_common = max(freq, key=freq.get)  # Get the key with max value

print(most_common)  # Output: 3
```

- **Time Complexity:** O(n)

📌 **Key Takeaway:**

- Use collections.Counter (O(n)) for efficiency.
- Avoid max(set(nums), key=nums.count) ($O(n^2)$) for large lists.

---

16. What Is the Time Complexity of the Built-in sorted() Function in Python?

The built-in sorted() function in Python uses **Timsort**, a hybrid sorting algorithm combining **Merge Sort** and **Insertion Sort**.

Time Complexity of sorted()

Case	Complexity
Best Case (Nearly Sorted)	**O(n)**
Average Case	**O(n log n)**
Worst Case	**O(n log n)**

Example Usage

```
arr = [4, 2, 7, 1, 9]

sorted_arr = sorted(arr)  # Returns a new sorted list

print(sorted_arr)  # Output: [1, 2, 4, 7, 9]
```

Why O(n log n) Complexity?

- **Merge Sort** contributes O(n log n).
- **Insertion Sort** improves performance for nearly sorted data (O(n) in the best case).

Custom Sorting

Sorting by absolute values:

```
arr = [-3, -1, 2, -7]

sorted_arr = sorted(arr, key=abs)

print(sorted_arr)  # Output: [-1, 2, -3, -7]
```

📌 **Key Takeaway:**

- sorted() is highly optimized and stable (preserves order of equal elements).
- Always runs in **O(n log n)** in the worst case.

---

17. How Would You Implement a Binary Search Algorithm in Python?

Binary Search is an efficient search algorithm that works on a **sorted list**. It follows a **divide-and-conquer** approach.

Binary Search Algorithm (Iterative)

```
def binary_search(arr, target):
    left, right = 0, len(arr) - 1

    while left <= right:
        mid = (left + right) // 2  # Find the middle index

        if arr[mid] == target:
            return mid  # Target found
        elif arr[mid] < target:
            left = mid + 1  # Search in the right half
        else:
            right = mid - 1  # Search in the left half

    return -1  # Target not found

# Example
arr = [1, 3, 5, 7, 9, 11]
target = 7
print(binary_search(arr, target))  # Output: 3 (Index of 7)
```

Binary Search Algorithm (Recursive)

```
def binary_search_recursive(arr, left, right, target):
```

```
    if left > right:
        return -1  # Base case: target not found

    mid = (left + right) // 2

    if arr[mid] == target:
        return mid
    elif arr[mid] < target:
        return binary_search_recursive(arr, mid + 1, right, target)
    else:
        return binary_search_recursive(arr, left, mid - 1, target)

# Example
arr = [1, 3, 5, 7, 9, 11]
target = 5
print(binary_search_recursive(arr, 0, len(arr) - 1, target))  # Output: 2
```

Time Complexity

Case	Complexity
Best Case (Middle Element)	**O(1)**
Average Case	**O(log n)**
Worst Case	**O(log n)**

📌 **Key Takeaway:**

- **Binary Search works only on sorted lists**.
- **O(log n)** makes it much faster than linear search (O(n)).

---

18. What Is the Purpose of the collections Module in Python?

The collections module provides specialized **container datatypes** beyond Python's built-in types (list, tuple, dict, set). These are optimized for performance and memory efficiency.

Key Data Structures in collections

1. **Counter** – Counts elements in an iterable.
2. **defaultdict** – A dictionary with default values.
3. **OrderedDict** – A dictionary that maintains the order of keys.
4. **deque** – A double-ended queue for fast append/pop.
5. **namedtuple** – A tuple with named fields.

Example 1: Counter – Counting Elements

```
from collections import Counter

nums = [1, 2, 2, 3, 3, 3, 4]
counter = Counter(nums)
print(counter)  # Output: Counter({3: 3, 2: 2, 1: 1, 4: 1})
print(counter.most_common(1))  # Output: [(3, 3)] (Most frequent element)
```

- **Use Case:** Count word occurrences in a text file.

Example 2: defaultdict – Avoiding Key Errors

```
from collections import defaultdict

word_count = defaultdict(int)  # Default value is 0
words = ["apple", "banana", "apple"]

for word in words:
```

```
word_count[word] += 1  # No need to check if key exists

print(word_count)  # Output: {'apple': 2, 'banana': 1}
```

- **Use Case:** Grouping data without handling missing keys.

Example 3: OrderedDict – Preserving Insertion Order

```
from collections import OrderedDict

ordered_dict = OrderedDict()
ordered_dict["a"] = 1
ordered_dict["b"] = 2
ordered_dict["c"] = 3

print(ordered_dict)  # Output: OrderedDict([('a', 1), ('b', 2), ('c', 3)])
```

- **Use Case:** Cache implementations where order matters.

Example 4: deque – Efficient Append and Pop

```
from collections import deque

dq = deque([1, 2, 3])
dq.append(4)  # Append to the right
dq.appendleft(0)  # Append to the left

print(dq)  # Output: deque([0, 1, 2, 3, 4])
print(dq.pop())  # Output: 4 (Removes from the right)
print(dq.popleft())  # Output: 0 (Removes from the left)
```

- **Use Case:** Implementing queues/stacks efficiently.

Example 5: namedtuple – Creating Readable Tuples

```
from collections import namedtuple


Person = namedtuple("Person", ["name", "age"])

p = Person("Alice", 30)


print(p.name)  # Output: Alice

print(p.age)  # Output: 30
```

- **Use Case:** Storing structured data like database rows.

📌 **Key Takeaway:**
The collections module provides **optimized data structures** for **counting, ordering, queuing, and structuring data**.

---

19. What Are List Comprehensions in Python? Can You Give an Example?

A **list comprehension** is a concise way to create lists using a **single-line expression**.

Basic Syntax:

```
[expression for item in iterable if condition]
```

- **expression** → The value to be stored in the new list.
- **iterable** → The sequence to iterate over.
- **condition (optional)** → A filter for elements.

Example 1: Creating a List of Squares

```
squares = [x**2 for x in range(5)]
```

```
print(squares)  # Output: [0, 1, 4, 9, 16]
```

Equivalent to:

```
squares = []
for x in range(5):
    squares.append(x**2)
```

Example 2: Filtering Even Numbers

```
nums = [1, 2, 3, 4, 5, 6]
evens = [x for x in nums if x % 2 == 0]
print(evens)  # Output: [2, 4, 6]
```

- **Condition (if x % 2 == 0) filters elements.**

Example 3: Nested List Comprehension

```
matrix = [[j for j in range(3)] for i in range(3)]
print(matrix)
# Output: [[0, 1, 2], [0, 1, 2], [0, 1, 2]]
```

- **Creates a 3×3 matrix.**

Example 4: Using if-else in List Comprehension

```
nums = [1, 2, 3, 4, 5]
result = ["Even" if x % 2 == 0 else "Odd" for x in nums]
print(result)  # Output: ['Odd', 'Even', 'Odd', 'Even', 'Odd']
```

- **Inline if-else condition.**

Example 5: Flattening a Nested List

```
nested_list = [[1, 2], [3, 4], [5, 6]]
```

```
flat_list = [num for sublist in nested_list for num in sublist]

print(flat_list)  # Output: [1, 2, 3, 4, 5, 6]
```

- **Extracts elements from sublists into a single list**.

📌 **Key Takeaway:**
List comprehensions are **faster and more readable** than traditional loops. 🚀

Conclusion

- **collections module** provides **efficient** data structures for counting, ordering, queuing, and structuring data.
- **List comprehensions** simplify list creation and filtering, making code more **Pythonic**.

Object-Oriented Programming (OOP):

20. Explain the Concept of Inheritance in Python.

Inheritance is an **OOP (Object-Oriented Programming) concept** where a class **inherits** properties and behaviors (methods) from another class. This promotes **code reusability** and **modularity**.

Types of Inheritance in Python

1. **Single Inheritance** – One class inherits from another.
2. **Multiple Inheritance** – A class inherits from multiple classes.
3. **Multilevel Inheritance** – A class inherits from a derived class.
4. **Hierarchical Inheritance** – Multiple classes inherit from one base class.
5. **Hybrid Inheritance** – A mix of the above.

Example 1: Single Inheritance

```
class Animal:
    def speak(self):
        return "I make sounds"
```

```
class Dog(Animal):  # Dog inherits from Animal
    def speak(self):
        return "Bark"

dog = Dog()
print(dog.speak())  # Output: Bark
```

- The Dog class **overrides** the speak() method from Animal.

Example 2: Multiple Inheritance

```
class Engine:
    def start(self):
        return "Engine started"

class Wheels:
    def roll(self):
        return "Wheels rolling"

class Car(Engine, Wheels):
    pass

my_car = Car()
print(my_car.start())  # Output: Engine started
print(my_car.roll())   # Output: Wheels rolling
```

- Car class **inherits methods** from both Engine and Wheels.

Example 3: Multilevel Inheritance

```
class Animal:
    def breathe(self):
        return "Breathing"


class Mammal(Animal):
    def has_fur(self):
        return "I have fur"


class Dog(Mammal):
    def bark(self):
        return "Bark"


dog = Dog()
print(dog.breathe())  # Output: Breathing
print(dog.has_fur())  # Output: I have fur
print(dog.bark())     # Output: Bark
```

- Dog inherits from Mammal, which inherits from Animal.

📌 **Key Takeaway:**
Inheritance **eliminates code duplication** and **follows the DRY principle (Don't Repeat Yourself).** 🚀

---

21. What Is the Purpose of the __init__ Method in Python?

The __init__ method is a **constructor** in Python that **initializes an object** when a class is instantiated. It allows passing values **at the time of object creation**.

Example: Using __init__ for Initialization

```
class Person:
    def __init__(self, name, age):
        self.name = name  # Instance variable
        self.age = age

    def greet(self):
        return f"Hi, I'm {self.name} and I'm {self.age} years old."

p1 = Person("Alice", 25)
print(p1.greet())  # Output: Hi, I'm Alice and I'm 25 years old.
```

- **__init__ assigns values** to name and age at object creation.
- Without __init__, attributes must be assigned **manually**.

Example: Default Arguments in __init__

```
class Car:
    def __init__(self, brand="Toyota"):
        self.brand = brand

car1 = Car("BMW")
car2 = Car()  # Uses default value

print(car1.brand)  # Output: BMW
print(car2.brand)  # Output: Toyota
```

- If no value is provided, the **default** brand="Toyota" is used.

📌 **Key Takeaway:**
__init__ **automates object initialization**, improving **efficiency** and **code clarity**. 🚗⚙️

---

22. What Are Class Variables and Instance Variables in Python?

Python classes have **two types of variables**:

1. **Instance Variables** – Defined inside __init__, **unique to each object**.
2. **Class Variables** – Defined inside the class but **shared among all objects**.

Example: Difference Between Instance and Class Variables

```
class Employee:
    company = "Google"  # Class variable

    def __init__(self, name, salary):
        self.name = name  # Instance variable
        self.salary = salary

emp1 = Employee("Alice", 70000)
emp2 = Employee("Bob", 80000)

print(emp1.company)  # Output: Google
print(emp2.company)  # Output: Google

Employee.company = "Microsoft"  # Changing class variable
print(emp1.company)  # Output: Microsoft
print(emp2.company)  # Output: Microsoft
```

```
emp1.salary = 75000  # Changing instance variable (only for emp1)

print(emp1.salary)  # Output: 75000

print(emp2.salary)  # Output: 80000
```

- company is a **class variable** → **shared across all instances**.
- name and salary are **instance variables** → **unique for each object**.

Key Differences

Feature	Class Variable	Instance Variable
Definition	Inside the class	Inside __init__
Shared?	Yes (same across all instances)	No (unique per instance)
Accessed via	ClassName.variable	self.variable
Modification	Changes reflect for all objects	Changes affect only that object

📌 **Key Takeaway:**

- **Class variables** store **global properties** of a class.
- **Instance variables** store **unique attributes** per object.

Conclusion

- **Inheritance** allows a class to derive properties from another class.
- **__init__** initializes objects automatically at creation.
- **Class variables** are **shared** across instances, while **instance variables** are **unique** per object.

23. Explain the Concept of Polymorphism in Python

Polymorphism means **"many forms"** and refers to the ability of different objects to be treated as instances of the same class through a shared interface. In Python, polymorphism allows methods in different classes to share the **same name** but behave **differently** based on the object calling them.

Example 1: Polymorphism in Methods

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

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

# Function that demonstrates polymorphism
def make_sound(animal):
    print(animal.speak())

dog = Dog()
cat = Cat()

make_sound(dog)  # Output: Bark
make_sound(cat)  # Output: Meow
```

📌 **Here, speak() behaves differently depending on whether the object is a Dog or a Cat.**

Example 2: Polymorphism in Built-in Functions

```
print(len("Hello"))     # Output: 5 (String length)
print(len([1, 2, 3, 4])) # Output: 4 (List length)
print(len({"a": 1, "b": 2})) # Output: 2 (Dictionary length)
```

📌 **Python's len() function behaves differently depending on the data type.**

Example 3: Polymorphism in Class Inheritance (Method Overriding)

```
class Animal:
    def make_sound(self):
        return "Some sound"


class Dog(Animal):
    def make_sound(self):  # Overriding method
        return "Bark"


dog = Dog()
print(dog.make_sound())  # Output: Bark
```

📌 **The make_sound() method in Dog overrides the one in Animal.**

Key Takeaway

Polymorphism allows **code flexibility** by allowing **different classes** to be treated as a **single entity**, reducing redundancy and improving maintainability. 🚀

---

24. What is the Difference Between @staticmethod and @classmethod?

Both @staticmethod and @classmethod **modify the behavior of methods**, but they serve different purposes.

1 .@staticmethod

- Does **not** require self or cls.
- It belongs to the class but **does not access or modify class attributes**.
- Used for utility/helper functions.

Example: @staticmethod

```
class MathOperations:
    @staticmethod
    def add(x, y):
        return x + y  # No need to access class attributes

print(MathOperations.add(3, 5))  # Output: 8
```

📌 **Here, add() is just a utility function that does not interact with the class.**

2.@classmethod

- Takes cls as the first argument.
- Can **modify class-level attributes**.
- Used when a method needs to work on the class itself, not instances.

Example: @classmethod

```
class Employee:
    company = "Google"

    @classmethod
    def change_company(cls, new_name):
        cls.company = new_name  # Modifies class variable

Employee.change_company("Microsoft")
print(Employee.company)  # Output: Microsoft
```

📌 **Here, change_company() modifies the class attribute company.**

🔍 **Key Differences:**

Feature	@staticmethod	@classmethod
Requires self or cls?	❌ No	✅ Yes (cls)
Accesses instance variables?	❌ No	❌ No
Modifies class variables?	❌ No	✅ Yes
Called on	Class or instance	Class or instance
Use case	Utility functions	Class-wide modifications

Key Takeaway

- **Use @staticmethod** for **independent** utility functions.
- **Use @classmethod** when **modifying class-level attributes**.

---

25. What is Multiple Inheritance in Python, and How Does It Work?

Multiple Inheritance allows a class to inherit **from more than one parent class**.

📌 **Python supports multiple inheritance, unlike some languages (e.g., Java), which only support single inheritance.**

Example 1: Basic Multiple Inheritance

```
class Engine:
    def start(self):
        return "Engine started"

class Wheels:
    def roll(self):
        return "Wheels rolling"
```

```
class Car(Engine, Wheels):  # Multiple inheritance
    pass

my_car = Car()
print(my_car.start())  # Output: Engine started
print(my_car.roll())   # Output: Wheels rolling
```

📌 **Here, Car inherits from both Engine and Wheels, gaining access to both methods.**

Example 2: Method Resolution Order (MRO) in Multiple Inheritance

If multiple parent classes have a **method with the same name**, **Python follows MRO (Method Resolution Order)** to determine which method to execute.

```
class A:
    def show(self):
        return "A"

class B(A):
    def show(self):
        return "B"

class C(A):
    def show(self):
        return "C"

class D(B, C):  # Multiple inheritance
    pass
```

```
obj = D()

print(obj.show())  # Output: B
```

📌 **Python follows the order defined in the class declaration (D(B, C)). It checks B first before C.**

🔍 How to Check MRO?

```
print(D.mro())

# Output: [<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>, <class 'object'>]
```

📌 **This shows the order in which Python searches for methods when calling obj.show().**

Key Takeaway

- **Multiple inheritance** allows a class to inherit from multiple classes.
- **MRO (Method Resolution Order)** decides which method is executed when there's a conflict.
- Python follows **C3 linearization (MRO algorithm)** to determine the method call order.

Final Summary

Concept	Key Takeaways
Polymorphism	Allows the same method to work differently across different objects.
@staticmethod vs @classmethod	@staticmethod is for utility functions, @classmethod modifies class attributes.
Multiple Inheritance	Allows inheriting from multiple classes, follows MRO for method resolution.

Advanced Python:

26. What Are Python's Built-in Data Types?

Python provides several built-in data types categorized as **numeric, sequence, set, mapping, and boolean types**.

- **Numeric Types**
  - **int** → Integer numbers
  - **float** → Floating-point numbers
  - **complex** → Complex numbers

Example:

```
a = 10      # int
b = 3.14    # float
c = 2 + 3j  # complex
```

- **Sequence Types**
  - **list** → Ordered, mutable collection
  - **tuple** → Ordered, immutable collection
  - **range** → Represents a sequence of numbers

Example:

```
lst = [1, 2, 3]  # List
tup = (4, 5, 6)  # Tuple
rng = range(1, 5) # 1, 2, 3, 4
```

- **Set Types**
  - **set** → Unordered collection of unique elements
  - **frozenset** → Immutable set

Example:

```
s = {1, 2, 3, 3}  # {1, 2, 3}

fs = frozenset([4, 5, 6])  # Immutable
```

🔹 **Mapping Type**

- **dict** → Key-value pairs

Example:

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

🔹 **Boolean Type**

- **bool** → True or False

Example:

```
x = True

y = False
```

🔹 **Binary Data Types**

- **bytes**, **bytearray**, **memoryview**

Example:

```
b = b"hello"  # Bytes

ba = bytearray(5)  # Mutable byte sequence
```

📌 **Key Takeaway:** Python has diverse data types for different needs, from numbers and sequences to mappings and binary data.

---

27. What is the Global Interpreter Lock (GIL) in Python, and How Does It Affect Multi-threading?

The **Global Interpreter Lock (GIL)** is a mutex that **allows only one thread to execute Python bytecode at a time** in CPython.

🔹 **Why Does GIL Exist?**

- Python's memory management uses **reference counting**.
- **GIL prevents race conditions** when multiple threads modify the same object.

- Ensures **thread safety**.

🔹 **Impact of GIL on Multi-threading**

- **CPU-bound tasks** (e.g., computations) **are slow** in multi-threading due to GIL.
- **I/O-bound tasks** (e.g., file I/O, network calls) **are not affected** much.

Example: GIL Effect on Multi-threading

```
import threading


def count():
  for _ in range(1000000):
    pass  # CPU-intensive task


t1 = threading.Thread(target=count)
t2 = threading.Thread(target=count)


t1.start()
t2.start()


t1.join()
t2.join()
```

📌 **Even with two threads, execution time does not improve due to GIL.**

🔹 **Workarounds for GIL**

✅ **Use multiprocessing** (creates separate processes):

```
from multiprocessing import Process


p1 = Process(target=count)
p2 = Process(target=count)
```

```
p1.start()
p2.start()

p1.join()
p2.join()
```

✅ **Use Jython or PyPy** (alternative Python implementations without GIL).
✅ **Use async programming** for I/O-bound tasks.

📌 **Key Takeaway:** GIL limits multi-threading for CPU-bound tasks but not for I/O-bound tasks. Use multiprocessing for parallelism. 🚀

---

28. What is the Difference Between map() and filter() in Python?

Both **map()** and **filter()** are higher-order functions used for processing iterables.

🔹 **map() Function**

- **Applies a function to each element** in an iterable.
- Returns a **map object (which can be converted to a list, tuple, etc.)**.

Example:

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

📌 **Used for transforming each element.**

🔹 **filter() Function**

- **Filters elements** based on a function that returns True or False.
- Returns a **filter object**.

Example:

```
nums = [1, 2, 3, 4, 5, 6]
evens = filter(lambda x: x % 2 == 0, nums)
print(list(evens))  # Output: [2, 4, 6]
```

📌 **Used for selecting elements based on a condition.**

🔍 **Key Differences:**

Feature	map()	filter()
Purpose	Transforms elements	Filters elements
Returns	New iterable with modified values	New iterable with filtered values
Function Output	Function returns any value	Function must return True/False

📌 **Key Takeaway:** Use **map() for transformation**, and **filter() for selection.** 🚀

---

29. Explain How to Handle Exceptions in Python with try/except/finally.

Python uses try/except/finally to handle runtime errors (exceptions).

🔹 **Basic Structure**

```
try:
    # Code that may raise an exception
    x = 10 / 0
except ZeroDivisionError as e:
    print(f"Error: {e}")
finally:
    print("This will always execute.")
```

📌 **The finally block runs regardless of whether an exception occurs.**

🔹 **Handling Multiple Exceptions**

```
try:
  x = int("abc")  # ValueError
except (ZeroDivisionError, ValueError) as e:
  print(f"Handled error: {e}")
```

🔹 **Using else with try/except**

```
try:
  x = 10 / 2  # No error
except ZeroDivisionError:
  print("Cannot divide by zero")
else:
  print("No errors occurred!")  # Runs if no exception occurs
```

📌 **Key Takeaway:** Always handle exceptions properly to prevent crashes. Use finally for cleanup actions.

---

30. How Would You Implement Concurrency in Python?

Python provides three main ways to achieve concurrency:

🔹 **1.Multi-threading (for I/O-bound tasks)**

```
import threading

def print_numbers():
  for i in range(5):
    print(i)

t1 = threading.Thread(target=print_numbers)
```

```
t1.start()
t1.join()
```

📌 **Best for I/O-bound tasks like file reading, API calls.** 🚀

🔹 **2.Multi-processing (for CPU-bound tasks)**

```
from multiprocessing import Process

def compute():
    print(sum(range(1000000)))

p = Process(target=compute)
p.start()
p.join()
```

📌 **Best for CPU-bound tasks (parallel processing).**

🔹 **3.Asynchronous Programming (asyncio)**

```
import asyncio

async def task():
    print("Task started")
    await asyncio.sleep(2)  # Simulate I/O
    print("Task finished")

asyncio.run(task())
```

📌 **Best for handling thousands of concurrent tasks efficiently (e.g., API requests, DB queries).**

Final Summary

Concept	Key Takeaways
Python Data Types	Includes numeric, sequence, set, mapping, and boolean types.
GIL	Allows only one thread at a time; use multiprocessing to bypass it.
map() vs filter()	map() transforms elements; filter() selects elements.
Exception Handling	Use try/except/finally to prevent crashes.
Concurrency	Use threading for I/O-bound, multiprocessing for CPU-bound, and async for high-performance tasks.