**PYTHON INTERVIEW QUESTIONS**

1. What are Python’s key features? Why is it called an interpreted language?

Python is a high-level, object-oriented programming language known for its **simplicity and readability**.  
Key features include:

* **Interpreted**: Executes code line-by-line using an interpreter, making debugging easier.
* **Dynamically typed**: No need to declare variable types.
* **Extensive libraries**: Rich standard and third-party libraries.
* **Cross-platform**: Runs on Windows, Linux, macOS.
* **Supports multiple paradigms**: Procedural, object-oriented, and functional.
* **Open-source**: Freely available and community-driven.

**2. Difference between Python 2 and Python 3**

* Python 2 (legacy, discontinued in 2020) vs Python 3 (current, supported).
* Major differences:
  + **Print**: print "Hello" (Py2) vs print("Hello") (Py3).
  + **Division**: 5/2 = 2 (Py2 integer division) vs 5/2 = 2.5 (Py3 true division).
  + **Unicode**: Strings are ASCII by default in Py2, Unicode by default in Py3.
  + **Community Support**: Only Python 3 is maintained.

**3.Explain indentation in Python. What happens if indentation is incorrect?**

* Python uses **indentation (spaces/tabs)** to define code blocks (instead of {} like Java/C).
* Example:

if True:

print("Yes")

* Incorrect indentation → **IndentationError** or **unexpected behavior**.

**4. What are Python keywords? Can you use them as variable names?**

* **Keywords** are reserved words that have predefined meaning (e.g., if, for, class, def, return).
* No, they **cannot** be used as variable names. Example:

class = 10 # ❌ SyntaxError

**5. Difference between list, tuple, set, and dictionary**

* List → Ordered, mutable, allows duplicates.

l = [1, 2, 2, 3]

* Tuple → Ordered, immutable, allows duplicates.

t = (1, 2, 2, 3)

* Set → Unordered, unique elements only.

s = {1, 2, 3}

* Dictionary → Key-value pairs, keys unique.

d = {"a": 1, "b": 2}

**6. Difference between mutable and immutable data types**

* Mutable: Can be changed after creation.  
  Examples: list, dict, set.
* Immutable: Cannot be changed after creation.  
  Examples: tuple, string, int.

**7. Explain == vs is operator in Python**

* == → Compares values.
* is → Compares memory reference (identity).

a = [1, 2]

b = [1, 2]

print(a == b) # True (same values)

print(a is b) # False (different objects in memory)

**8. Difference between append() vs extend() in lists**

* append(x) → Adds one element to the list.
* extend(iterable) → Adds all elements of an iterable.

l = [1, 2]

l.append([3, 4]) # [1, 2, [3, 4]]

l = [1, 2]

l.extend([3, 4]) # [1, 2, 3, 4]

**9. Shallow copy vs deep copy**

* Shallow copy → Creates a new object but references inner objects.
* Deep copy → Creates new object + recursively copies inner objects.
* import copy

a = [[1, 2], [3, 4]]

shallow = copy.copy(a)

deep = copy.deepcopy(a)

a[0][0] = 99

print(shallow) # [[99, 2], [3, 4]]

print(deep) # [[1, 2], [3, 4]]

**10. How does Python handle memory management (Garbage collection)?**

* Python uses automatic memory management:
  + Reference counting: Keeps track of references to objects.
  + Garbage Collector (GC): Removes cyclic references (via gc module).
* Unreferenced objects are collected automatically, freeing memory.

**11. Functions in Python; Difference between \*args and \*\*kwargs**

A function in Python is a reusable block of code that performs a specific task. It is defined using the def keyword and can accept parameters and return values.

\*args: Accepts a variable number of **positional arguments** as a tuple.

\*\*kwargs: Accepts a variable number of **keyword arguments** as a dictionary.

**12. Explain decorators in Python with an example**

A **decorator** is a function that modifies the behavior of another function without changing its code. It’s commonly used for logging, access control, and timing.

**13.What are generators and yield?**

A **generator** is a special type of function that returns an iterator and produces values one at a time using the yield keyword. It supports **lazy evaluation**, which is memory-efficient.

**14.Difference between iterable, iterator, and generator**

- Iterable : an object that can be looped over (eg., list, tuple). Has \_\_iter\_\_() method.

- Iterator: an object with \_\_next\_\_() method that returns items one by one.

- Generator: a type of iterator created using Yield. Automatically implements \_\_iter\_\_() and \_\_next\_\_()

**15.Explain list comprehension with an example**

**List comprehension** is a concise way to create lists using a single line of code.

squares = [x\*\*2 for x in range(5)] print(squares) # Output: [0, 1, 4, 9, 16]

**16.Difference between classmethod, staticmethod, and instance method**

**Instance Method**: Operates on object (self).

**Class Method**: Operates on class (cls). Uses @classmethod.

**Static Method**: No access to class or object. Uses @staticmethod.

class Example: def instance\_method(self): pass @classmethod def class\_method(cls): pass @staticmethod def static\_method(): pass

**17.Python’s OOP concepts: inheritance, polymorphism, encapsulation**

* **Inheritance**: Enables a class to inherit attributes and methods from another class.
* **Polymorphism**: Allows different classes to define methods with the same name but different behavior.
* **Encapsulation**: Restricts access to internal data using private variables (\_var, \_\_var).

**18. What are magic/dunder methods in Python? Examples**

**Magic methods** (or **dunder methods**) are special methods with double underscores used to customize class behavior.

Examples:

* \_\_init\_\_: Constructor method
* \_\_str\_\_: String representation
* \_\_len\_\_, \_\_add\_\_, \_\_eq\_\_, etc.

class Person: def \_\_init\_\_(self, name): self.name = name def \_\_str\_\_(self): return f"My name is {self.name}"

**19. Explain Python’s Global Interpreter Lock (GIL)**

The **GIL** is a mutex in CPython that allows only one thread to execute Python bytecode at a time.  
It ensures **thread safety** during memory management using reference counting.  
While it simplifies single-threaded execution, it limits true parallelism in **CPU-bound** multi-threaded programs.  
For **I/O-bound** tasks, the GIL is released during operations, so threading can still be effective.

**20.Difference between deepcopy vs copy.copy() in Python**

Copy.copy() -> Shallow copy: Copies the object but **not nested objects**.

Copy.deepcopy() -> -> Deep copy: Copies the object and **all nested objects recursively**.

import copy a = [[1, 2], [3, 4]] shallow = copy.copy(a) deep = copy.deepcopy(a)

Modifying a[0][0] affects shallow, but not deep.

21. Explain try, except, finally, else in Python.

*In Python, exception handling is done using try, except, else, and finally blocks. The try block contains code that might raise an exception. If an exception occurs, the except block handles it. If no exception occurs, the else block runs. The finally block always executes, regardless of whether an exception occurred or not—it's typically used for cleanup tasks like closing files or releasing resources*

try:

result = 10 / 2

except ZeroDivisionError:

print("Division by zero!")

else:

print("Division successful:", result)

finally:

print("Execution complete.")

22. What are custom exceptions? How do you create one?

*Custom exceptions are user-defined error types that help make error handling more descriptive and specific to the application logic. We create them by subclassing Python’s built-in Exception class.*

class InvalidAgeError(Exception):

def \_\_init\_\_(self, message="Age must be above 18"):

super().\_\_init\_\_(message)

def validate\_age(age):

if age < 18:

raise InvalidAgeError()

This allows us to raise meaningful exceptions that are easier to debug and maintain.

23. Explain the with statement and context managers in Python.

*The with statement in Python is used to simplify resource management, like opening files or managing database connections. It ensures that resources are properly cleaned up, even if an error occurs. This is achieved using context managers, which implement \_\_enter\_\_ and \_\_exit\_\_ methods.*

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

content = file.read()

*Here, the file is automatically closed after the block, even if an exception occurs. We can also create custom context managers using classes or the contextlib module.*

24. What are lambda functions? Give an example using map, filter, and reduce.

*Lambda functions are anonymous functions defined using the lambda keyword. They're useful for short, throwaway functions, especially in functional programming contexts like map, filter, and reduce.*

from functools import reduce

nums = [1, 2, 3, 4, 5]

squared = list(map(lambda x: x\*\*2, nums)) # [1, 4, 9, 16, 25]

evens = list(filter(lambda x: x % 2 == 0, nums)) # [2, 4]

product = reduce(lambda x, y: x \* y, nums)

# 120

25. Difference between isinstance() and type().

*Both are used for type checking, but isinstance() is more flexible. It checks if an object is an instance of a class or its subclass, while type() checks for an exact match.*

class Animal: pass

class Dog(Animal): pass

d = Dog()

print(isinstance(d, Animal)) # True

print(type(d) == Animal) # False

*So, in most cases, isinstance() is preferred, especially when working with inheritance.*

26. Difference between **module** and **package** in Python

*In Python, a* ***module*** *is a single .py file that contains Python code—functions, classes, or variables. A* ***package****, on the other hand, is a directory that contains multiple modules and an \_\_init\_\_.py file to make it recognizable as a package.*

**Example:**

* math.py → module
* numpy/ → package containing multiple modules like core.py, linalg.py, etc.

*Packages help organize code hierarchically, especially in larger projects*

27. What are **Python namespaces and scope** (LEGB rule)?

*A* ***namespace*** *is a mapping between names and objects. Python uses namespaces to keep track of variable names and their values. The* ***scope*** *determines where a variable can be accessed.*

**LEGB Rule** defines the order of scope resolution:

* **L**: Local — inside the current function
* **E**: Enclosing — in outer functions (for nested functions)
* **G**: Global — at the module level
* **B**: Built-in — Python’s built-in names like len, print

def outer():

x = "enclosing"

def inner():

x = "local"

print(x)

inner()

*Here, Python resolves x using the LEGB rule.*

28. Explain **import vs from-import**

*Both are used to bring external code into the current script. import loads the entire module, while from-import allows importing specific components.*

import math

print(math.sqrt(16))

from math import sqrt

print(sqrt(16))

*Use import when you need multiple functions or want to avoid name conflicts. Use from-import for cleaner syntax when only a few items are needed.*

29. What is a **virtual environment (venv)** in Python? Why is it needed?

*"A virtual environment is an isolated Python environment that allows you to manage dependencies separately for each project. It prevents conflicts between packages used in different projects."*

**Why it's needed:**

* Keeps dependencies clean and project-specific
* Avoids version clashes
* Makes deployment and collaboration easier

Command to create: python -m venv myenv

After activation, any packages installed go into that environment, not the global Python installation.

30. Difference between **shallow copy, deep copy, assignment operator**

*Refer above*

*31.* *How is* ***JSON*** *handled in Python (json module)?*

*"Python provides the json module to work with JSON data, which is commonly used for data exchange in web applications."*

***Key functions:***

* *json.dumps() → Converts Python object to JSON string*
* *json.loads() → Converts JSON string to Python object*
* *json.dump() → Writes JSON to a file*
* *json.load() → Reads JSON from a file*

***Example:***

*import json data = {"name": "Deepthi", "age": 22} json\_str = json.dumps(data) parsed = json.loads(json\_str)*

*"This module makes it easy to serialize and deserialize data for APIs and config files."*

***FILE : 02***

***Lists, Strings & Collections***

*32.* ***Difference Between append() and extend() in Lists:***

* ***append()****: Adds a single element to the end of the list. The element can be any type, including another list, but it will be added as a single object (a nested list if the element is a list).*
* *lst = [1, 2, 3]*
* *lst.append(4) # lst becomes [1, 2, 3, 4]*
* *lst.append([5, 6]) # lst becomes [1, 2, 3, 4, [5, 6]]*
* ***extend()****: Adds each element of an iterable (like another list, tuple, or string) to the list. This effectively "flattens" the iterable and appends the individual elements to the original list.*
* *lst = [1, 2, 3]*
* *lst.extend([4, 5]) # lst becomes [1, 2, 3, 4, 5]*

33. **Difference Between Shallow Copy and Deep Copy:**

* **Shallow Copy**: Creates a new object, but does not recursively copy nested objects. It copies references to nested objects, meaning if the nested objects are modified, the changes will reflect in both the original and copied objects.
* import copy
* lst1 = [1, [2, 3]]
* lst2 = copy.copy(lst1) # Shallow copy
* lst2[1][0] = 10 # Modifies the nested list in both lst1 and lst2
* **Deep Copy**: Creates a new object and recursively copies all objects, including nested ones. Changes to the deep-copied object do not affect the original object.
* import copy
* lst1 = [1, [2, 3]]
* lst2 = copy.deepcopy(lst1) # Deep copy

lst2[1][0] = 10 # lst1 remains unaffected

34. **How Do You Remove Duplicates From a List?**

There are several ways to remove duplicates:

* **Using set()**: Converts the list to a set (which removes duplicates) and then converts it back to a list.
* lst = [1, 2, 2, 3, 4, 4]
* lst = list(set(lst)) # lst becomes [1, 2, 3, 4] (order may not be preserved)
* **Using a Loop**: Retains order and removes duplicates by checking if an element is already in the result list.
* lst = [1, 2, 2, 3, 4, 4]
* result = []
* for item in lst:
* if item not in result:
* result.append(item) # result becomes [1, 2, 3, 4]
* **Using dict.fromkeys()**: Retains order and removes duplicates efficiently.
* lst = [1, 2, 2, 3, 4, 4]

lst = list(dict.fromkeys(lst)) # lst becomes [1, 2, 3, 4]

35. **How Do Slicing Operations Work on Lists and Strings?**

* **Slicing Syntax**: list[start:stop:step]
  + start: Index from which to start the slice (inclusive).
  + stop: Index at which to end the slice (exclusive).
  + step: Defines the stride (how many steps to skip).

Examples:

* **List Slicing**:
* lst = [0, 1, 2, 3, 4, 5]
* print(lst[1:4]) # Output: [1, 2, 3]
* print(lst[:3]) # Output: [0, 1, 2]
* print(lst[::2]) # Output: [0, 2, 4]
* **String Slicing**:
* s = "Hello, World!"
* print(s[7:12]) # Output: 'World'

print(s[::-1]) # Output: '!dlroW ,olleH'

36. **How Do You Reverse a String in Python?**

There are multiple ways to reverse a string in Python:

* **Using slicing**:

s = "Hello"

reversed\_s = s[::-1] # Output: 'olleH'

* **Using reversed() function**:

s = "Hello"

reversed\_s = ''.join(reversed(s)) # Output: 'olleH'

* **Using a loop**:

s = "Hello"

reversed\_s = ""

for char in s:

reversed\_s = char + reversed\_s

# Output: 'olleH'

**Control Flow & Loops**

**37. What is the difference between break, continue, and pass?**

* **break: Exits the loop entirely, regardless of the loop's condition. It is typically used when a certain condition is met and you no longer want to continue with the loop.**

**for i in range(10):**

**if i == 5:**

**break # Exits the loop when i equals 5**

**print(i)**

**# Output: 0 1 2 3 4**

* **continue: Skips the current iteration of the loop and proceeds with the next iteration. It’s often used to skip over specific conditions.**

**for i in range(5):**

**if i == 2:**

**continue # Skips the iteration when i equals 2**

**print(i)**

**# Output: 0 1 3 4**

* **pass: A placeholder that does nothing. It is used when a statement is syntactically required, but you don't want to execute any code. It’s commonly used in empty functions, classes, or loops.**

**for i in range(5):**

**if i == 2:**

**pass # Does nothing when i equals 2**

**print(i)**

**# Output: 0 1 2 3 4**

**38.** **What is the purpose of the else clause in loops?**

The **else** clause in loops executes after the loop completes normally (i.e., it doesn’t exit due to a break). If the loop is exited prematurely with a break, the else block is skipped. It’s often used when you want to run code only if the loop was not terminated by a break.

* **Example without break**:

for i in range(5):

print(i)

else:

print("Loop completed without break.")

# Output:

# 0

# 1

# 2

# 3

# 4

# Loop completed without break.

* **Example with break**:

for i in range(5):

if i == 3:

break

print(i)

else:

print("Loop completed without break.") # This won't execute

# Output:

# 0

# 1

# 2

39. **How does enumerate() work in Python?**

The enumerate() function adds a counter to an iterable (like a list or a string) and returns an enumerate object, which generates pairs of index and value. It's useful when you need both the index and value during iteration.

lst = ['a', 'b', 'c']

for index, value in enumerate(lst):

print(index, value)

# Output:

# 0 a

# 1 b

# 2 c

You can also specify a starting index:

for index, value in enumerate(lst, start=1):

print(index, value)

# Output:

# 1 a

# 2 b

# 3 c

**40. How do you iterate through a dictionary?**

You can iterate through a dictionary in several ways:

* **Iterating through keys** (default behavior):

d = {'a': 1, 'b': 2, 'c': 3}

for key in d:

print(key)

# Output:

# a

# b

# c

* **Iterating through values** using .values():

for value in d.values():

print(value)

# Output:

# 1

# 2

# 3

* **Iterating through both keys and values** using .items():

for key, value in d.items():

print(key, value)

# Output:

# a 1

# b 2

# c 3

**41. How do you use list comprehension in Python?**

List comprehension is a concise way to create lists by applying an expression to each item in an iterable, optionally filtering items using a condition.

**Basic syntax**:

[expression for item in iterable]

* **Example without condition**:

lst = [x \*\* 2 for x in range(5)]

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

* **Example with condition** (only include even numbers):

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

print(lst) # Output: [0, 2, 4, 6, 8]

* **Using multiple loops** (nested list comprehension):

lst = [(x, y) for x in range(2) for y in range(2)]

print(lst) # Output: [(0, 0), (0, 1), (1, 0), (1, 1)]

**Functions**

42. **What are default arguments in Python functions?**

Default arguments are values provided in a function definition that are used when the caller doesn't pass an argument for that parameter. They allow functions to be called with fewer arguments than they are defined to take.

**Example**:

def greet(name, message="Hello"):

print(f"{message}, {name}!")

greet("Alice") # Uses default value for message

greet("Bob", "Hi") # Overrides default value for message

43. **What is the difference between \*args and \*\*kwargs?**

* **\*args**: Allows a function to accept any number of positional arguments. It collects extra positional arguments into a tuple.

**Example**:

def sum\_numbers(\*args):

return sum(args)

print(sum\_numbers(1, 2, 3)) # Output: 6

* + In this case, args is a tuple: (1, 2, 3).
* **\*\*kwargs**: Allows a function to accept any number of keyword arguments. It collects extra keyword arguments into a dictionary.

**Example**:

def print\_info(\*\*kwargs):

for key, value in kwargs.items():

print(f"{key}: {value}")

print\_info(name="Alice", age=25)

**Output**:

name: Alice

age: 25

* **Key Difference**:
  + \*args is for positional arguments (non-keyword).
  + \*\*kwargs is for keyword arguments (named arguments).

**44. Explain pass-by-value vs pass-by-reference in Python.**

Python uses a mechanism often described as **"pass-by-object-reference"** or **"pass-by-assignment"**. This means:

* The reference to the object is passed to the function, not the actual object itself. However, whether the object can be modified inside the function depends on the object type.
* **Mutable objects** (e.g., lists, dictionaries) can be modified inside the function because their reference is passed.

**Example** (Mutable object - list):

def modify\_list(lst):

lst.append(4)

my\_list = [1, 2, 3]

modify\_list(my\_list)

print(my\_list) # Output: [1, 2, 3, 4]

* **Immutable objects** (e.g., integers, strings, tuples) cannot be modified inside the function. If you attempt to change the value, a new object is created.

**Example** (Immutable object - integer):

def modify\_number(num):

num += 1

my\_num = 5

modify\_number(my\_num)

print(my\_num) # Output: 5 (no modification)

* + Here, the num variable inside the function points to a new integer object after the += operation, leaving the original value unchanged.

**45. Can functions be assigned to variables in Python?**

Yes, functions can be assigned to variables in Python. Functions are first-class objects, meaning they can be passed around like any other object, such as strings or integers.

**Example**:

def greet(name):

return f"Hello, {name}!"

greeting\_function = greet

print(greeting\_function("Alice")) # Output: Hello, Alice!

* In this example, greeting\_function is assigned to the greet function. When called, it behaves like the original function.

**46. What are lambda functions?**

A **lambda function** is a small anonymous function defined with the lambda keyword. It can take any number of arguments but can only have a single expression. The result of the expression is automatically returned.

**Syntax**:

lambda arguments: expression

**Example**:

add = lambda x, y: x + y

print(add(2, 3)) # Output: 5

* In this example, lambda x, y: x + y is a function that takes two arguments and returns their sum. It's equivalent to:
* def add(x, y):
* return x + y

**When to use lambda functions?**

* Lambda functions are often used where small, throwaway functions are needed, like in sorting, filtering, or mapping operations.

**Example with map()**:

numbers = [1, 2, 3]

squared = list(map(lambda x: x \*\* 2, numbers))

print(squared) # Output: [1, 4, 9]

**OOP in Python**

47. **What are classes and objects in Python?**

* **Class**: A class is a blueprint or a template for creating objects. It defines the attributes (variables) and methods (functions) that the objects created from the class will have. A class encapsulates data and behavior.

**Example**:

class Dog:

def \_\_init\_\_(self, name, age):

self.name = name # attribute

self.age = age # attribute

def bark(self): # method

print(f"{self.name} says Woof!")

* **Object**: An object is an instance of a class. It is created from the class and has its own set of data (attributes).

**Example**:

my\_dog = Dog("Rex", 5) # object of class Dog

my\_dog.bark() # Output: Rex says Woof!

* **Key Difference**:
  + **Class** is a blueprint for creating objects.
  + **Object** is an instance of a class.

**48. What are instance methods, class methods, and static methods?**

* **Instance Methods**: These are the regular methods in a class that operate on an instance (object) of the class. They take self as the first parameter, which refers to the current instance of the class.

**Example**:

class MyClass:

def instance\_method(self):

print("This is an instance method.")

* + You call it like: obj.instance\_method()
* **Class Methods**: These methods operate on the class itself, rather than on instances of the class. They take cls as the first parameter, which refers to the class, not the instance.
  + To define a class method, you use the @classmethod decorator.

**Example**:

class MyClass:

@classmethod

def class\_method(cls):

print("This is a class method.")

* + You call it like: MyClass.class\_method()
* **Static Methods**: These methods don't operate on the instance or the class. They are utility functions that don't need access to self or cls. You define a static method with the @staticmethod decorator.

**Example**:

class MyClass:

@staticmethod

def static\_method():

print("This is a static method.")

* + You call it like: MyClass.static\_method()
* **Key Differences**:
  + **Instance methods**: Take self (object reference) as the first argument.
  + **Class methods**: Take cls (class reference) as the first argument.
  + **Static methods**: Do not take self or cls, and don't rely on class or instance.

**49. What are dunder (magic) methods in Python?**

* **Dunder methods** (also known as **magic methods** or **special methods**) are methods that start and end with double underscores (\_\_). These methods allow you to define the behavior of objects for built-in operations such as addition, string representation, and comparisons.
* **Common examples**:
  + \_\_init\_\_(self, ...): Constructor for object initialization.
  + \_\_str\_\_(self): Defines the string representation of an object (used by print()).
  + \_\_repr\_\_(self): Defines the official string representation of an object (used by repr()).
  + \_\_add\_\_(self, other): Defines behavior for the + operator.
  + \_\_len\_\_(self): Defines behavior for len().

**Example**:

class Point:

def \_\_init\_\_(self, x, y):

self.x = x

self.y = y

def \_\_str\_\_(self):

return f"Point({self.x}, {self.y})"

def \_\_add\_\_(self, other):

return Point(self.x + other.x, self.y + other.y)

p1 = Point(2, 3)

p2 = Point(4, 5)

print(p1 + p2) # Output: Point(6, 8)

* **Key Purpose**: Dunder methods allow customizing how objects behave with built-in operations, making objects more intuitive to work with.

**50. What is the purpose of the \_\_init\_\_ method?**

The \_\_init\_\_ method is a **constructor** in Python. It is automatically called when a new object (instance) of a class is created. Its purpose is to initialize the object's state (attributes) with the values passed as arguments.

* It is not technically a return method, and it doesn’t return anything (None by default).
* The \_\_init\_\_ method is used to set up the initial state of the object.

**Example**:

class Dog:

def \_\_init\_\_(self, name, age):

self.name = name # Initialize name attribute

self.age = age # Initialize age attribute

dog1 = Dog("Rex", 5) # `\_\_init\_\_` is called automatically

print(dog1.name) # Output: Rex

print(dog1.age) # Output: 5

* Here, the \_\_init\_\_ method initializes the name and age attributes when a new Dog object is created.

**51. What is polymorphism in Python?**

**Polymorphism** in Python refers to the ability of different objects to respond to the same method or operation in different ways. It allows a single interface (method or function) to be used for different types of objects.

There are two types of polymorphism in Python:

* **Method Overloading**: Python does not support method overloading in the traditional sense (same method name with different signatures). However, polymorphism can still be achieved by passing different types of arguments to a single method.
* **Method Overriding**: Subclasses can override methods from the base class to provide a specific implementation.

**Example**:

class Animal:

def speak(self):

raise NotImplementedError("Subclass must implement abstract method")

class Dog(Animal):

def speak(self):

return "Woof!"

class Cat(Animal):

def speak(self):

return "Meow!"

def animal\_sound(animal):

print(animal.speak())

# Polymorphism in action

dog = Dog()

cat = Cat()

animal\_sound(dog) # Output: Woof!

animal\_sound(cat) # Output: Meow!

* In this example, the method speak() is overridden in both the Dog and Cat classes. The function animal\_sound() works polymorphically, as it can take any object that inherits from Animal and call the speak() method, regardless of the specific subclass.
* **Key Purpose of Polymorphism**: It provides flexibility and makes code more general, allowing different objects to be treated in a uniform way.

**Iterators, Generators & Comprehensions**

52. **What is the difference between an iterable and an iterator?**

* **Iterable**: Any Python object capable of returning its members one at a time. Examples: lists, tuples, strings, sets, dictionaries. These objects implement the \_\_iter\_\_() method.
* **Iterator**: An object that represents a stream of data. It remembers its current state and produces the next element when next() is called. Iterators implement both \_\_iter\_\_() and \_\_next\_\_() methods.

👉 In short: **Iterables can be looped over, Iterators actually produce the values one by one.**

**53. What is the difference between iter() and next()?**

* iter(iterable) → returns an **iterator object** from an iterable.
* next(iterator) → fetches the **next element** from an iterator. If no more items are available, it raises StopIteration.

Example:

numbers = [1, 2, 3]

it = iter(numbers) # convert iterable to iterator

print(next(it)) # 1

print(next(it)) # 2

**54. What are generators in Python?**

* Generators are **special iterators** that allow you to generate values on the fly instead of storing them in memory.
* They are written like normal functions but use the yield keyword instead of return.
* Generators are memory-efficient and useful when dealing with large datasets or streams of data.

Example:

def countdown(n):

while n > 0:

yield n

n -= 1

for i in countdown(5):

print(i)

**55. What is the purpose of the yield keyword?**

* The yield keyword is used inside a function to make it a generator.
* Unlike return, it **pauses the function execution**, saving its state, and resumes from there when the next value is requested.
* It helps in writing **lazy evaluations** and **infinite sequences** efficiently.

**56. How do dictionary comprehensions work?**

* Dictionary comprehensions provide a concise way to create dictionaries using an expression inside curly braces {}.
* General syntax:
* {key\_expr: value\_expr for item in iterable if condition}
* Example:
* nums = [1, 2, 3, 4]
* square\_dict = {x: x\*\*2 for x in nums if x % 2 == 0}
* print(square\_dict) # {2: 4, 4: 16}

Exception Handling

**57. What is the difference between Exception and BaseException?**

* BaseException is the **root of all exceptions** in Python. All built-in exceptions are derived from it.
* Exception is a **subclass of BaseException** and is the base for most user-defined and system exceptions.
* You usually catch Exception in try/except blocks, but you rarely catch BaseException (because it also includes things like KeyboardInterrupt and SystemExit, which you usually don’t want to handle).

**58. Explain try, except, finally, and else.**

* try: The block where you place the code that may raise exceptions.
* except: Handles the exception if it occurs.
* else: Runs only if no exception occurs in the try block.
* finally: Always runs, whether an exception occurred or not (commonly used for cleanup like closing files).

Example:

try:

x = 10 / 2

except ZeroDivisionError:

print("Division by zero!")

else:

print("No error, result:", x)

finally:

print("Execution finished")

**59. What are custom exceptions? How do you create them?**

* Custom exceptions are **user-defined errors** created to handle specific conditions in your program.
* They are created by subclassing the built-in Exception class.

Example:

class NegativeNumberError(Exception):

pass

def check\_number(n):

if n < 0:

raise NegativeNumberError("Negative numbers not allowed")

check\_number(-5)

**60. Can one except block handle multiple exceptions?**

* Yes. You can group multiple exceptions in a **tuple** inside a single except block.

Example:

try:

value = int("abc")

except (ValueError, TypeError):

print("Invalid conversion or type issue")

**61. What happens if there is no matching except block for an error?**

* If no matching except block is found:
  1. The exception propagates up the call stack.
  2. If it reaches the top level and is still unhandled, the program **terminates** with a traceback error message.

**Decorators & Context Managers**

62. What are decorators in Python?

* Decorators are functions that modify the behavior of other functions or classes without changing their code.
* They are applied using the @decorator\_name syntax above a function.

Example:

def decorator(func):

def wrapper():

print("Before function")

func()

print("After function")

return wrapper

@decorator

def hello():

print("Hello")

hello()

63. Can a function return another function in Python?

* Yes. In Python, functions are first-class objects, so a function can return another function.

Example:

def outer():

def inner():

return "Inner function result"

return inner

f = outer()

print(f()) # Inner function result

64. What is a higher-order function?

* A higher-order function is a function that either:
  1. Takes another function as an argument, or
  2. Returns a function as its result.
* Examples: map(), filter(), sorted() with key.

Example:

def square(x): return x\*x

print(list(map(square, [1,2,3])))

65. What is a context manager in Python?

* A context manager is an object that defines how to set up and clean up resources automatically.
* It implements the \_\_enter\_\_() and \_\_exit\_\_() methods.
* The most common example is file handling using with.

Example:

with open("file.txt", "w") as f:

f.write("Hello")

# file is automatically closed here

66. What is the difference between with and try-finally?

* Both ensure proper cleanup of resources.
* try-finally: You must explicitly close or clean up resources in the finally block.
* with: Automatically manages resources using the context manager protocol (\_\_enter\_\_, \_\_exit\_\_). It is cleaner, safer, and less error-prone.

**Modules & Imports**

**67. What is the difference between a module and a package?**

* **Module**: A single Python file (.py) that contains code (functions, classes, variables).
* **Package**: A collection of modules organized in a directory with an \_\_init\_\_.py file (can be empty in Python 3.3+).  
  👉 In short: **module = file, package = folder of modules**.

**68. What does \_\_name\_\_ == "\_\_main\_\_" mean in Python?**

* Every Python module has a built-in variable \_\_name\_\_.
* If the script is run **directly**, \_\_name\_\_ is set to "\_\_main\_\_".
* If the script is **imported as a module**, \_\_name\_\_ is set to the module’s name.
* This allows you to control which code runs when the file is imported vs executed directly.

Example:

if \_\_name\_\_ == "\_\_main\_\_":

print("Script is running directly")

**69. What is the difference between import and from-import?**

* import module → imports the **entire module**. You access functions/classes with module.name.
* from module import name → imports a **specific function/class/variable** directly.

Example:

import math

print(math.sqrt(16)) # using module name

from math import sqrt

print(sqrt(16)) # direct access

**70. How do relative imports work in Python?**

* Relative imports are used **inside packages** to import from sibling or parent modules.
* They use **dots (.)** to indicate the current or parent directory.
  + from . import module → import from the same package
  + from .. import module → import from parent package

Example (inside mypackage/subpkg/module1.py):

from . import module2 # same folder

from .. import utils # parent folder

**71. How does Python search for modules when importing?**

* Python follows a search order defined in sys.path:
  1. The **current directory** of the script.
  2. The directories in the PYTHONPATH environment variable.
  3. The **standard library directories**.
  4. The **site-packages** directory (third-party packages).

👉 You can check the search order with:

import sys

print(sys.path)

**Namespaces & Scope**

**72. What is the LEGB rule in Python?**

* LEGB defines the **scope resolution order** Python follows when looking up a variable:
  + **L (Local):** Inside the current function.
  + **E (Enclosing):** In outer (non-global) functions if nested.
  + **G (Global):** Defined at the top level of the module.
  + **B (Built-in):** Python’s built-in names (e.g., len, sum).

👉 Python searches in this order and uses the first match.

**73. What is the difference between global and local variables?**

* **Local variable:** Defined inside a function, accessible only within that function.
* **Global variable:** Defined outside all functions, accessible throughout the module (and across modules if imported).

**74. How do you use the global keyword in Python?**

* The global keyword is used **inside a function** to modify a variable declared in the **global scope**.

Example:

x = 5

def modify():

global x

x = 10

modify()

print(x) # 10

**75. How do you use the nonlocal keyword?**

* nonlocal is used in **nested functions** to modify variables from the **enclosing (non-global) scope**.
* Without it, reassigning a variable inside an inner function would create a new local variable instead of updating the outer one.

Example:

def outer():

x = "outer"

def inner():

nonlocal x

x = "modified"

inner()

print(x)

outer() # modified

**76. What happens when two variables with the same name exist in different scopes?**

* Python applies the **LEGB rule**:
  + The **innermost scope variable** takes priority.
  + Outer scope variables are shadowed (hidden) unless accessed using global or nonlocal.

Example:

x = "global"

def func():

x = "local"

print(x)

func() # local

print(x) # global

**Data Handling**

**77. How do you read and write files in Python?**

* Use the built-in open() function with a mode:
  + "r" → read
  + "w" → write (overwrites)
  + "a" → append
  + "rb"/"wb" → binary read/write

Example:

with open("file.txt", "w") as f:

f.write("Hello")

with open("file.txt", "r") as f:

data = f.read()

**78. What is the difference between text mode and binary mode in file handling?**

* **Text mode ("t")** → Default. Reads/writes strings. Handles character encoding (utf-8).
* **Binary mode ("b")** → Reads/writes raw bytes. No encoding/decoding. Used for images, audio, executables.

**79. How do you handle CSV files in Python?**

* Use the built-in csv module.
* **Reading:**

import csv

with open("data.csv", "r") as f:

reader = csv.reader(f)

for row in reader:

print(row)

* **Writing:**

with open("data.csv", "w", newline="") as f:

writer = csv.writer(f)

writer.writerow(["Name", "Age"])

writer.writerow(["Alice", 25])

**80. How do you serialize and deserialize JSON in Python?**

* Use the json module.
* **Serialization (Python → JSON):**

import json

data = {"name": "Alice", "age": 25}

json\_str = json.dumps(data)

* **Deserialization (JSON → Python):**

parsed = json.loads(json\_str)

* You can also use json.dump() and json.load() for working directly with files.

**81. How do you handle command-line arguments in Python?**

* Use the built-in sys module or the more powerful argparse module.

Example with sys.argv:

import sys

print(sys.argv) # list of command-line arguments

Example with argparse:

import argparse

parser = argparse.ArgumentParser()

parser.add\_argument("name")

args = parser.parse\_args()

print("Hello,", args.name)

**Functional Programming**

**82.What is the difference between map(), filter(), and reduce()?**

* **map(func, iterable)** → Applies a function to **each element** and returns a new iterable.
* **filter(func, iterable)** → Filters elements where the function returns **True**.
* **reduce(func, iterable)** → Repeatedly applies a function to elements, reducing them to a **single value** (in functools).

**83. What is a lambda function?**

* A **lambda function** is an **anonymous, single-expression function** defined using the lambda keyword.
* Syntax: lambda args: expression

Example:

square = lambda x: x\*x

print(square(5)) # 25

**84. Can you give an example where filter() is useful?**

* Example: Filtering even numbers from a list.

nums = [1, 2, 3, 4, 5, 6]

evens = list(filter(lambda x: x % 2 == 0, nums))

print(evens) # [2, 4, 6]

**85. How does reduce() work in Python?**

* reduce() applies a **rolling computation** on a sequence and returns a single result.
* It is part of functools.

Example:

from functools import reduce

nums = [1, 2, 3, 4]

product = reduce(lambda x, y: x \* y, nums)

print(product) # 24

**86. What is the difference between list comprehension and map()?**

* **List comprehension:** More **readable and flexible**, allows filtering and complex expressions.
* **map():** Applies only a single function to all items; returns an iterator.

Example (both square numbers):

nums = [1, 2, 3]

squares1 = [x\*x for x in nums] # list comprehension

squares2 = list(map(lambda x: x\*x, nums)) # map

👉 **List comprehension is preferred** for readability, while map() is good for functional programming style.