Python

🐍 Python Introduction

1️⃣ What is Python?

Simple definition

Python is a **high-level, interpreted, dynamically typed, general-purpose programming language**.

Interview one-liner

> Python is an interpreted, high-level language known for its readability, dynamic typing, and extensive standard library.

2️⃣ Why Python was created

Python was created to:

- Make programming **simple and readable**
- Reduce complexity of languages like C, C++
- Allow faster development

Created by **Guido van Rossum (1991)**

Philosophy: **"Readability counts"**

3️⃣ Key Features of Python

- Interpreted
- Dynamically typed
- Object-oriented
- Platform independent
- Large standard library
- Automatic memory management

⚙️ Advantages and Disadvantages of Python

✅ Advantages of Python

1️⃣ Interpreted Language

What it means

Python code is executed **line by line**.

Advantages

- Easy debugging
- No compilation step
- Faster development

Example

If an error occurs on line 10, lines 1–9 already ran.

2️⃣ Dynamically Typed

What it means

You don't declare variable types.

```
x =10
x ="hello"
```

Advantages

- Less code
- Faster coding
- Flexible programming

3️⃣ Easy to Learn & Readable

- English-like syntax
- Less boilerplate code

```
print("Hello World")
```

4️⃣ Large Standard Library

- Built-in modules for:
  - File handling
  - Math
  - Networking
  - OS operations

5️⃣ Platform Independent

- Same code runs on Windows, Linux, Mac

❌ Disadvantages of Python

1️⃣ Slower Execution Speed

❌ Because:

- Interpreted
- Dynamic typing
- Runtime checks

Python is slower than C/C++.

2️⃣ Runtime Errors (Dynamic Typing)

Errors appear **only during execution**.

```
x ="10"
y =5
print(x + y)# TypeError
```

3️⃣ High Memory Consumption

- Objects + references
- Garbage collection overhead

4️⃣ Not Ideal for Mobile & Game Engines

- Performance limitations
- Limited mobile support

⚠️ Interview Tip

Python trades **performance** for **developer productivity**.

🧠 Working of Python Interpreter 🔥

Step-by-Step Execution Flow

Example Code

```
a =10
b =20
print(a + b)
```

1️⃣ Source Code (`.py` file)

You write Python code in a `.py` file.

2️⃣ Compilation to Bytecode

Python **compiles source code into bytecode**.

- Bytecode is:
  - Low-level
  - Platform-independent
- Stored as `.pyc` file
- Saved inside `__pycache__`

📌 Happens **automatically**.

3️⃣ Python Virtual Machine (PVM)

- PVM executes bytecode
- Converts bytecode to machine instructions
- Executes line by line

📌 PVM is why Python is platform-independent.

4️⃣ Execution Output

The result is shown on:

- Terminal
- Console
- Output stream

🔁 Visual Flow (Mental Model)

```
Source Code (.py)
      ↓
Bytecode (.pyc)
      ↓
Python Virtual Machine (PVM)
      ↓
Output
```

🔥 Important Interview Clarifications

❓ Is Python compiled or interpreted?

✅ **Both**

- Compiled → to bytecode
- Interpreted → executed by PVM

📌 Best answer in interviews.

❓ Why Python is slower?

Because:

- Interpreted execution
- Dynamic type checking
- Garbage collection

❓ What is PVM?

> PVM is the runtime engine that executes Python bytecode.

🧠 Mental Model (Remember Forever)

- Python ≈ **Translator**
- Source code → translated to bytecode
- PVM → reads and executes

🧪 Common Interview Questions

1. Why Python is called interpreted?
2. Difference between compiled and interpreted languages?

3. What is bytecode?
4. Where is `.pyc` stored?
5. Why Python is platform independent?

✅ Final Summary

- Python is high-level, interpreted & dynamic
- Easy to learn, powerful, but slower
- Python uses bytecode + PVM
- Understanding interpreter = strong foundation

🐍 Types of Data Types

1️⃣ What it is

Simple definition

A **data type** defines **what kind of data an object can hold** and **what operations can be performed on it**.

Interview one-liner

> In Python, data types classify objects based on the kind of value they store and the operations they support.

📌 Important:

Python is **dynamically typed**, but **strongly typed**.

2️⃣ Why data types are used

Data types help Python:

- Allocate memory efficiently
- Decide valid operations
- Prevent invalid operations

Example:

```
"10" +"20"# valid → "1020"
10 +20# valid → 30
"10" +20# ❌ TypeError
```

3️⃣ Before this, what was used

Earlier languages:

- Manual type declaration (C, C++)
- Weak type checking (JavaScript)

Problems:

❌ Verbose

❌ Error-prone

❌ Less flexibility

Python solved this with **automatic type inference**.

4️⃣ Built-in Data Types in Python

Python has **8 main built-in data type categories**.

🔷 1. Numeric Types

Used for numbers.

Type	Example
`int`	`10`, `-5`
`float`	`3.14`, `2.5`
`complex`	`3+4j`

```
x =10
y =2.5
z =3 +4j
```

📌 Interview tip: Python integers have **no size limit**.

🔷 2. Boolean Type

Used for **True/False**.

```
a =True
b =False
```

- Subclass of `int`
- `True == 1`, `False == 0`

🔷 3. String (`str`)

Used to store text.

```
name ="Swalih"
```

Characteristics:

- Immutable
- Indexed
- Supports slicing

```
name[0]# 'S'
```

🔷 4. List

Used to store **ordered, mutable** collection.

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

✔️ Allows duplicates
✔️ Index-based
✔️ Mutable

🔷 5. Tuple

Used to store **ordered, immutable** collection.

```
t = (1,2,3)
```

✔ Faster than lists

✔ Used for fixed data

🔷 Range()

The Python **range() function** returns a sequence of numbers, in a given range. The most common use of it is to iterate sequences on a sequence of numbers using **Python** loops.

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

Designed for **readability, Abstracts looping logic, Focuses on iteration**

- Returns a **range object**
- **Does NOT store all numbers**
- Generates values **lazily**

🔷 6. Set

Used to store **unordered, unique** elements.

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

✔ No duplicates

✔ Fast lookup

📌 Cannot access by index.

🔷 7. Dictionary (`dict`)

Stores data as **key–value pairs**.

```
user = {"name":"Swalih","age":25}
```

✔ Keys must be immutable

✔ Fast lookups

🔷 8. NoneType

Represents **absence of value**.

```
x =None
```

Used for:

- Default return values
- Empty variables

5️⃣ Classification by Mutability (VERY IMPORTANT 🔥)

Mutable(not change)	Immutable(change)
list	int
dict	float
set	bool
	str
	tuple

📌 Interview favorite question.

6️⃣ Where data types are used (Real-world)

Data Type	Use Case
int	Counters
float	Measurements
str	User input
list	Product list
tuple	Coordinates
set	Unique IDs
dict	JSON / APIs

7️⃣ Advantages of Python data types

✅ Built-in support

✅ Memory efficient

✅ Rich methods

✅ Dynamic typing

8️⃣ Disadvantages

❌ Runtime errors

❌ Memory overhead

❌ Slower than low-level languages

9️⃣ Mental Model (Remember Forever 🧠)

👉 Think of **data types as different containers**:

- List → Editable notebook
- Tuple → Printed book
- Set → Filter
- Dict → Phonebook

🔟 Interview Questions & Answers

Q1: Is Python statically typed?

❌ No, dynamically typed

Q2: Difference between list and tuple?

List	Tuple
Mutable	Immutable
Slower	Faster
More memory	Less memory

Q3: Why keys in dictionary must be immutable?

Because keys must be **hashable**.

Q4: Is `bool` a data type?

Yes, subclass of `int`.

1️⃣1️⃣ Common Mistakes

❌ Using list as dict key

❌ Modifying tuple

❌ Expecting set order

❌ Confusing `None` with `0`

✅ Final Summary

- Python has **8 built-in data types**
- Objects have types, variables don't
- Mutability matters for bugs & performance
- Data types = core of Python logic

Types of Variables declaration

Python variables are classified **by scope and behavior**, not by data type.

🔷 1. Local Variables

What it is

Variables declared **inside a function**.

```
defshow():
    x =10# local variable
print(x)
```

Scope

- Accessible **only inside the function**

Interview point

> Local variables are created when a function is called and destroyed after execution.

🔷 2. Global Variables

What it is

Variables declared **outside all functions**.

```
x =20# global variable

defshow():
print(x)
```

Scope

- Accessible **inside and outside functions**

Modifying global variable inside function

```
x =10

defupdate():
global x
    x =20
```

Interview trap ⚠️

Using too many global variables is **bad practice**.

🔷 3. Instance Variables

What it is

Variables that belong to **an object (instance)**.

```
classPerson:
def__init__(self, name):
self.name = name# instance variable
```

Characteristics

- Each object has its **own copy**
- Defined using `self`

Real-world analogy

Each person has their **own name**, age, etc.

🔷 4. Class Variables (Static Variables)

What it is

Variables shared by **all objects of a class**.

```
classPerson:
    species ="Human"# class variable
```

Characteristics

- Shared memory
- Defined outside methods

Interview difference

Instance Variable	Class Variable
Object specific	Shared
Uses `self`	Uses class name

🔷 5. Nonlocal Variables

What it is

Variables used in **nested functions**.

```
defouter():
    x =10
definner():
nonlocal x
        x =20
    inner()
print(x)
```

Why needed

To modify variable from **outer function but not global**.

🔷 6. Static Variables (Python context)

Python doesn’t have true static variables like Java, but:

- **Class variables** behave like static
- Function attributes can simulate static behavior

```
defcounter():
ifnothasattr(counter,"count"):
        counter.count =0
    counter.count +=1
print(counter.count)
```

5️⃣ Where it is used (Real-world)

Variable Type	Use Case
Local	Temporary calculations
Global	Config values
Instance	User data
Class	Shared constants
Nonlocal	Closures

6️⃣ Advantages

✅ Easy memory management

✅ Readable and clean code

✅ Supports OOP concepts

✅ Dynamic typing flexibility

7️⃣ Disadvantages

❌ Global variables cause bugs

❌ Dynamic typing can cause runtime errors

❌ Harder debugging in large projects

8️⃣ Mental Model (Remember Forever 🧠)

Think of **variables as labels**, not boxes.

- The **label** points to an object
- Multiple labels can point to the same object

```
a =10
b = a
```

Both point to **same object 10**.

9️⃣ Interview Questions & Answers

Q1: Does Python have data-type-based variables?

No. Python is dynamically typed.

Q2: Difference between global and nonlocal?

Global	Nonlocal
Outside all functions	Outer function
Used with `global`	Used with `nonlocal`

Q3: Are Python variables stored in stack or heap?

- Variables → references (stack)
- Objects → heap

Q4: Can a variable change its type?

Yes.

```
x =10
x ="Hello"
```

🐍 Python Functions

1️⃣ What is a Function?

Simple definition

A **function** is a reusable block of code that performs a specific task.

Interview one-liner

> A function is a named block of reusable code that executes when called.

2️⃣ Why Functions are used

- Avoid code repetition
- Improve readability
- Easy maintenance
- Modular programming
- Testing & debugging

3️⃣ Creating a Function

Syntax

```
def function_name(parameters):
    # function body
    return value
```

Example

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

📌 `def` → function keyword

📌 Function body must be **indented**

4️⃣ Calling a Function

```
greet()
```

📌 Function code executes **only when called**.

5️⃣ Function with Parameters & Arguments

```
def greet(name):
    print(f"Hello {name}")

greet("Swalih")
```

📌 **Parameter** → variable in function definition

📌 **Argument** → value passed during function call

6️⃣ Types of Arguments in Python

🔷 1. Positional Arguments

Arguments are passed in **order**.

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

add(2, 3)
```

🔷 2. Keyword Arguments

Arguments passed using **parameter names**.

```
def greet(name, age):
    print(name, age)

greet(age=25, name="Swalih")
```

📌 Order doesn't matter.

🔷 3. Default Arguments

Parameters with default values.

```
def greet(name="Guest"):
    print(name)

greet()
greet("Swalih")
```

🔷 4. Arbitrary Arguments (`args`) ⭐

What it is

Allows a function to accept **any number of positional arguments**.

Example

```
def add(*args):
    print(args)

add(1, 2, 3, 4)
```

📌 `args` is a **tuple**.

Real-world Example

```
def total(*prices):
    return sum(prices)

total(100, 200, 300)
```

🔷 5. Arbitrary Keyword Arguments (`*kwargs`) ⭐⭐

What it is

Allows a function to accept **any number of keyword arguments**.

Example

```
def profile(**kwargs):
    print(kwargs)

profile(name="Swalih", age=25, city="Calicut")
```

📌 `kwargs` is a **dictionary**.

7️⃣ Mixing Arguments (INTERVIEW FAVORITE 🔥)

Correct Order

```
def func(positional, *args, default=10, **kwargs):
    pass
```

Order Rule

1. Positional
2. `args`
3. Default
4. `*kwargs`

❌ Wrong order causes error.

8️⃣ Return Statement

```
def square(x):
    return x * x
```

📌 `return` ends function execution.

9️⃣ Common Mistakes

❌ Forgetting `return`

❌ Confusing args and kwargs

❌ Using mutable default arguments

⚠️ Mutable default argument problem:

```
def func(x=[]):
    x.append(1)
    return x
```

🔟 Mental Model (Remember Forever 🧠)

- Function = **Machine**
- Input = Arguments
- Process = Function body
- Output = Return value

1️⃣1️⃣ Interview Questions & Answers

Q1: Difference between `args` and `*kwargs`?

`*args`	`**kwargs`
Tuple	Dictionary
Positional	Keyword
Order based	Key-value

Q2: Are functions objects in Python?

✅ Yes (first-class functions).

Q3: Can a function return multiple values?

✅ Yes (returns a tuple).

```
def calc(a, b):
    return a+b, a-b
```

✅ Final Summary

- Functions make code reusable
- Python supports flexible arguments
- `args` → many positional arguments
- `*kwargs` → many keyword arguments
- Essential for clean code & interviews

🚀 Next Topics (Pick One)

- **Lambda functions**
- **Recursive functions**
- **Scope & LEGB rule**
- **Decorators (advanced)**

🐍 Recursion

1️⃣ What is Recursion?

Simple definition

Recursion is a technique where a **function calls itself** to solve a problem.

Interview one-liner

> Recursion is a process in which a function calls itself until a base condition is met.

2️⃣ Why Recursion is used

Recursion is used when:

- A problem can be broken into **smaller sub-problems**
- The structure is **self-similar**

Examples:

- Factorial
- Tree traversal
- File systems
- Divide & conquer algorithms

3️⃣ Two Mandatory Parts of Recursion (VERY IMPORTANT 🔥)

Every recursive function must have:

1️⃣ Base Case

Stops the recursion.

2️⃣ Recursive Case

Function calls itself.

❌ Without base case → infinite recursion.

4️⃣ Basic Recursion Example

Example: Print numbers

```
def print_numbers(n):
    if n == 0:         # base case
        return
    print(n)
    print_numbers(n-1)  # recursive call

print_numbers(5)
```

Output

```
5
4
3
2
1
```

5️⃣ Factorial using Recursion (INTERVIEW FAVORITE 🔥)

Mathematical:

```
5! = 5 × 4 × 3 × 2 × 1
```

Code

```
def factorial(n):
    if n == 1:          # base case
        return 1
    return n * factorial(n-1)
```

Trace (Important 🧠)

```
factorial(5)
= 5 * factorial(4)
= 5 * 4 * factorial(3)
= 5 * 4 * 3 * factorial(2)
= 5 * 4 * 3 * 2 * factorial(1)
= 120
```

6️⃣ Fibonacci using Recursion

```
def fib(n):
    if n <= 1:
        return n
    return fib(n-1) + fib(n-2)
```

📌 Interview note: Recursive Fibonacci is **inefficient**.

7️⃣ Recursion vs Loop (INTERVIEW 🔥)

Recursion	Loop
Function calls itself	Repeats block
More readable	Faster
Uses stack memory	Less memory
Risk of stack overflow	Safe

8️⃣ Visualizing Recursion (STACK 🧠)

Example: `factorial(3)`

```
factorial(3)
  factorial(2)
    factorial(1)
```

Then returns in **reverse order**.

9️⃣ Common Recursion Problems (with examples)

🔷 Sum of digits

```
def sum_digits(n):
    if n == 0:
        return 0
    return n % 10 + sum_digits(n // 10)
```

🔷 Reverse a string

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

🔷 Power of number

```
def power(a, b):
    if b == 0:
        return 1
    return a * power(a, b-1)
```

🔟 Common Mistakes (VERY IMPORTANT ⚠️)

❌ Missing base case

❌ Wrong base condition

❌ Too deep recursion

❌ Using recursion where loop is better

1️⃣1️⃣ Recursion Limit in Python

Python has a recursion limit:

```
import sys
sys.getrecursionlimit()
```

Default ≈ **1000**

Changing limit (NOT recommended):

```
sys.setrecursionlimit(2000)
```

1️⃣2️⃣ Tail Recursion (INTERVIEW NOTE)

Python **does NOT optimize** tail recursion.

```
def fact(n, result=1):
    if n == 0:
        return result
    return fact(n-1, result*n)
```

Still uses stack.

1️⃣3️⃣ Mental Model (Remember Forever 🧠)

- Recursion = **Function solving smaller version of itself**
- Base case = **STOP sign**
- Stack = **Call history**

1️⃣4️⃣ Interview Questions & Answers

Q1: What happens if no base case?

Infinite recursion → `RecursionError`

Q2: Is recursion faster than loops?

❌ No (loops are faster in Python)

Q3: Where is recursion useful?

Tree, graph, divide-and-conquer problems

✅ Final Summary

- Recursion = function calling itself
- Needs base + recursive case
- Elegant but memory heavy
- Important for interviews

🚀 Next Topics (Pick One)

- **Recursion practice problems (step-by-step)**
- **Backtracking**
- **Tree recursion**
- **Dynamic Programming vs Recursion**

Tell me what you want next 💪

🐍 Lambda Functions

1️⃣ What is a Lambda Function?

Simple definition

A **lambda function** is an **anonymous (nameless), one-line function**.

Interview one-liner

> A lambda function is a small anonymous function defined using the lambda keyword.

2️⃣ Why Lambda is used

- Write **short functions quickly**
- Avoid boilerplate `def`
- Used with `map()`, `filter()`, `sorted()`
- Improves readability (when used correctly)

3️⃣ Syntax of Lambda

```
lambda arguments: expression
```

📌 Returns the expression result automatically

📌 No `return` keyword

📌 One expression only

4️⃣ Basic Examples

Example 1: Add two numbers

```
add = lambda a, b: a + b
print(add(2, 3))
```

Equivalent `def`:

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

Example 2: Square a number

```
square = lambda x: x * x
```

5️⃣ Lambda with Conditional (IMPORTANT 🔥)

```
check_even = lambda x: "Even" if x % 2 == 0 else "Odd"

check_even(5)
```

📌 Ternary only (no full `if-else` blocks).

6️⃣ Lambda with `map()`

Used to **transform data**.

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

squares = list(map(lambda x: x * x, nums))
```

7️⃣ Lambda with `filter()`

Used to **filter data**.

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

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

8️⃣ Lambda with `sorted()` (INTERVIEW FAVORITE 🔥)

Sort by length

```
words = ["apple", "kiwi", "banana"]

sorted(words, key=lambda x: len(x))
```

Sort list of tuples

```
students = [("A", 80), ("B", 95), ("C", 70)]

sorted(students, key=lambda x: x[1])
```

9️⃣ Lambda with `reduce()` (Advanced 🔥)

```
from functools import reduce

nums = [1, 2, 3, 4]

total = reduce(lambda a, b: a + b, nums)
```

🔟 Lambda vs Normal Function (INTERVIEW 🔥)

Lambda	def
Anonymous	Named
One expression	Multiple statements
Short & quick	Detailed logic
No docstring	Supports docstring

1️⃣1️⃣ Common Mistakes

❌ Using lambda for complex logic

❌ Forgetting lambda returns automatically

❌ Poor readability

❌ Using lambda everywhere

1️⃣2️⃣ When NOT to use Lambda ❌

- Multiple conditions
- Loops
- Error handling
- Reusable complex logic

Use `def` instead.

1️⃣3️⃣ Mental Model (Remember Forever 🧠)

- Lambda = **Inline function**
- Use once, small logic
- Think of it as a **formula**

1️⃣4️⃣ Interview Questions & Answers

Q1: Can lambda have multiple expressions?

❌ No (only one)

Q2: Can lambda contain loops?

❌ No

Q3: Is lambda faster?

❌ No significant difference

Q4: Can lambda return multiple values?

✅ Yes (as tuple)

```
f = lambda x: (x, x*x)
```

1️⃣5️⃣ Practice Examples 🧠

Try these:

1. Find max of two numbers using lambda
2. Sort a list of dictionaries by value
3. Filter strings starting with "a"
4. Convert list of strings to uppercase

✅ Final Summary

- Lambda = anonymous one-line function
- Best with `map`, `filter`, `sorted`
- Improves readability when used wisely
- Common interview topic

🚀 Next Topics (Pick One)

- **map vs filter vs reduce**
- **List comprehension vs lambda**
- **Decorators**
- **Higher-order functions**

Exception Handling

1️⃣ What is an Exception?

Simple definition

An **exception** is a runtime error that interrupts the normal flow of a program.

Interview one-liner

> An exception is an error that occurs during program execution.

2️⃣ Why Exception Handling is used

- Prevent program crash
- Handle runtime errors gracefully
- Improve user experience
- Write safe & robust code

Without handling:

```
x = 10 / 0   # ZeroDivisionError
```

Program crashes ❌

3️⃣ Keywords in Exception Handling

Keyword	Purpose
`try`	Code that may raise error
`except`	Handle error
`else`	Runs if no error
`finally`	Always runs

4️⃣ Basic `try-except`

```
try:
    x = 10 / 0
except ZeroDivisionError:
    print("Cannot divide by zero")
```

5️⃣ Handling Multiple Exceptions

```
try:
    a = int("abc")
except ValueError:
    print("Invalid conversion")
except ZeroDivisionError:
    print("Division error")
```

6️⃣ Catching Multiple Exceptions Together

```
try:
    x = int("abc")
except (ValueError, TypeError):
    print("Invalid input")
```

7️⃣ `try-except-else`

When to use

Use `else` for code that should run **only if no exception occurs**.

```
try:
    x = int(input("Enter number: "))
except ValueError:
    print("Invalid input")
else:
    print("You entered:", x)
```

8️⃣ `finally` Block (VERY IMPORTANT 🔥)

What it does

`finally` runs **no matter what**.

```
try:
    f = open("file.txt")
```

```
except FileNotFoundError:
    print("File not found")
finally:
    print("Done")
```

📌 Used for:

- Closing files
- Releasing resources
- Cleanup code

9️⃣ Complete Flow Example (INTERVIEW 🔥)

```
try:
    a = int(input("Enter A: "))
    b = int(input("Enter B: "))
    result = a / b
except ZeroDivisionError:
    print("Cannot divide by zero")
except ValueError:
    print("Invalid input")
else:
    print("Result:", result)
finally:
    print("Program ended")
```

🔟 Raising Exceptions Manually

```
age = -5
if age < 0:
    raise ValueError("Age cannot be negative")
```

1️⃣1️⃣ Custom Exceptions (Basic 🔥)

```
class MyError(Exception):
    pass

raise MyError("Something went wrong")
```

1️⃣2️⃣ Common Built-in Exceptions

Exception	Reason
`ZeroDivisionError`	Divide by zero
`ValueError`	Wrong value
`TypeError`	Wrong type
`IndexError`	Index out of range
`KeyError`	Missing key
`FileNotFoundError`	File missing

1️⃣3️⃣ Common Mistakes ⚠️

❌ Catching all exceptions blindly

```
except:
    pass
```

❌ Ignoring errors

❌ Putting too much code in `try`

❌ Not using `finally` for cleanup

1️⃣4️⃣ Mental Model (Remember Forever 🧠)

- `try` → Risky code
- `except` → Safety net
- `else` → Success path
- `finally` → Cleanup crew

1️⃣5️⃣ Interview Questions & Answers

Q1: Difference between `else` and `finally`?

else	finally
Runs if no error	Always runs
Optional	Optional
Logic code	Cleanup

Q2: Can `finally` run without `except`?

✅ Yes

```
try:
    print("Hello")
finally:
    print("Bye")
```

Q3: Best practice for exception handling?

- Catch specific exceptions
- Keep `try` blocks small
- Log errors

1️⃣6️⃣ Practice Questions 🧠

Try these:

1. Handle division by zero
2. Handle invalid input conversion
3. Read file safely
4. Create custom exception for age validation

✅ Final Summary

- Exceptions prevent crashes
- `try-except` handles runtime errors
- `else` runs on success
- `finally` always runs

- Essential for real-world code

🚀 Next Topics (Pick One)

- **File Handling**
- **Custom Exceptions (Advanced)**
- **Context Managers (`with`)**
- **Debugging & Logging**

🐍 Closure and Namespace

PART 1️⃣ — Namespace

1️⃣ What is a Namespace?

Simple definition

A **namespace** is a **mapping between names and objects**.

Interview one-liner

> A namespace is a container that holds names and their corresponding objects.

```
x = 10
```

Here:

- `x` → name
- `10` → object
- Mapping stored in a namespace

2️⃣ Why Namespace is needed

- Avoid name conflicts
- Organize variables
- Enable scope control

Example problem without namespace:

```
x = 10
x = "hello"  # overwritten
```

Namespace separates **where names live**.

3️⃣ Types of Namespaces (INTERVIEW FAVORITE 🔥)

Python has **4 main namespaces**:

Namespace	Description
Built-in	Python default names
Global	Names at module level
Enclosing	Outer function names
Local	Inside current function

🔷 Built-in Namespace

```
print, len, int
```

```
import builtins
dir(builtins)
```

🔷 Global Namespace

```
x = 10   # global
```

🔷 Local Namespace

```
def func():
    y = 20  # local
```

🔷 Enclosing Namespace

```
def outer():
    z = 30
    def inner():
        print(z)
```

4️⃣ LEGB Rule (VERY IMPORTANT 🔥)

Python searches variables in this order:

```
L → E → G → B
```

Level	Meaning
L	Local
E	Enclosing
G	Global
B	Built-in

Example

```
x = "global"

def outer():
    x = "enclosing"
    def inner():
        x = "local"
        print(x)
    inner()

outer()
```

Output:

```
local
```

5️⃣ `global` vs `nonlocal`

`global`

Modifies global variable

```
x = 10
def f():
    global x
    x = 20
```

`nonlocal`

Modifies enclosing variable

```
def outer():
    x = 10
    def inner():
        nonlocal x
        x = 20
```

🧠 PART 2️⃣ — Closures

6️⃣ What is a Closure?

Simple definition

A **closure** is a function that **remembers variables from its enclosing scope**, even after that scope is gone.

Interview one-liner

> A closure is a function that retains access to variables from its enclosing scope.

7️⃣ Why Closures are used

- Data hiding
- Maintain state
- Avoid global variables
- Functional programming

8️⃣ Basic Closure Example

```
def outer():
    x = 10
    def inner():
        print(x)
    return inner

f = outer()
f()
```

Output

```
10
```

📌 `x` still exists even though `outer()` finished.

9️⃣ Closure with Modification (`nonlocal`) 🔥

```
def counter():
    count = 0
    def increment():
        nonlocal count
        count += 1
        return count
    return increment

c = counter()
print(c())  # 1
print(c())  # 2
```

🔟 How Closure Works Internally (INTERVIEW 🔥)

- Inner function stores **reference to outer variables**
- Stored in `__closure__`

```
print(c.__closure__)
```

1️⃣1️⃣ Closure vs Class (INTERVIEW COMPARISON 🔥)

Closure	Class
Lightweight	Heavy
No `self`	Uses `self`
Functional style	OOP style
Short-lived state	Long-lived state

1️⃣2️⃣ Common Mistake in Closures ⚠️

Late Binding Problem

```
funcs = []
for i in range(3):
    def f():
        print(i)
    funcs.append(f)

funcs[0]()  # prints 2 (unexpected)
```

Fix

```
def f(x=i):
    print(x)
```

1️⃣3️⃣ Mental Model (Remember Forever 🧠)

Namespace

- Namespace = **Dictionary**
- Name → Object mapping

Closure

- Closure = **Backpack**

- Function carries remembered variables

1️⃣4️⃣ Interview Questions & Answers

Q1: What is LEGB?

Local → Enclosing → Global → Built-in

Q2: Why use closures instead of globals?

Encapsulation + safety

Q3: Can a closure modify outer variables?

✅ Yes, using `nonlocal`

Q4: How to check if a function is a closure?

Check `__closure__`

✅ Final Summary

- Namespace controls **where names live**
- LEGB defines lookup order
- Closures remember enclosing variables
- Closures are powerful & interview-important

🚀 Next Topics (Pick One)

- **Decorators (built on closures)**
- **LEGB deep dive with examples**
- **Closures vs Lambda**
- **Scope & lifetime of variables**

🐍 Decorators

1️⃣ What is a Decorator?

Simple definition

A **decorator** is a function that **modifies another function's behavior without changing its code**.

Interview one-liner

> A decorator is a function that wraps another function to extend or modify its behavior.

2️⃣ Why Decorators are used

- Add extra functionality (logging, auth, timing)
- Keep code **clean**
- Follow **DRY** principle
- Widely used in frameworks (Flask, Django)

3️⃣ Functions are Objects (Foundation 🔥)

In Python:

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

x = greet
x()
```

📌 Functions can be:

- Passed as arguments
- Returned from functions

This makes decorators possible.

4️⃣ Decorator WITHOUT `@` (Core Concept)

Step-by-step example

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

Now decorate manually:

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

say_hello = my_decorator(say_hello)
say_hello()
```

Output

```
Beforefunction
Hello
Afterfunction
```

5️⃣ Decorator WITH `@` Syntax (Pythonic 🔥)

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

@my_decorator
defsay_hello():
print("Hello")

say_hello()
```

📌 `@my_decorator` is just **syntactic sugar**.

6️⃣ Decorator with Arguments (`args` , `*kwargs`) ⭐

Most real functions have parameters.

```
defmy_decorator(func):
defwrapper(*args, **kwargs):
print("Before function")
        result = func(*args, **kwargs)
print("After function")
return result
return wrapper
```

Example

```
@my_decorator
defadd(a, b):
return a + b

print(add(2,3))
```

7️⃣ Decorator that RETURNS value (IMPORTANT 🔥)

Always return the function result if needed.

❌ Wrong:

```
func(*args)
```

✅ Correct:

```
return func(*args)
```

8️⃣ Real-World Decorator Examples

🔷 Timing Decorator (Interview Favorite 🔥)

```
import time

deftimer(func):
defwrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
print(f"Time: {end - start}")
return result
return wrapper

@timer
defslow_func():
    time.sleep(1)
```

```
slow_func()
```

🔷 Authorization Decorator

```
defrequire_login(func):
defwrapper(user):
ifnot user.get("logged_in"):
print("Access denied")
return
return func(user)
return wrapper
```

9️⃣ `functools.wraps` (VERY IMPORTANT 🔥)

Without it:

- Function name lost
- Docstring lost

Proper way:

```
from functoolsimport wraps

defmy_decorator(func):
    @wraps(func)
defwrapper(*args, **kwargs):
return func(*args, **kwargs)
return wrapper
```

📌 Interview must-know.

🔟 Decorators with Arguments (ADVANCED 🔥)

```
defrepeat(n):
defdecorator(func):
defwrapper(*args, **kwargs):
for _inrange(n):
            func(*args, **kwargs)
return wrapper
return decorator
```

Usage:

```
@repeat(3)
defhello():
print("Hello")
```

1️⃣1️⃣ Multiple Decorators (Order Matters 🔥)

```
@decorator1
@decorator2
deffunc():
```

```
pass
```

Execution order:

```
decorator1(decorator2(func))
```

1️⃣2️⃣ Decorator vs Closure (INTERVIEW 🔥)

Decorator	Closure
Special use of closure	General concept
Modifies function	Stores state
Uses `@` syntax	No special syntax

1️⃣3️⃣ Common Mistakes ⚠️

❌ Forgetting `*args, **kwargs`

❌ Not returning function result

❌ Losing metadata (no `wraps`)

❌ Overusing decorators

1️⃣4️⃣ Mental Model (Remember Forever 🧠)

- Decorator = **Gift wrap**
- Function = **Gift**
- Wrapper = **Wrapping paper**

1️⃣5️⃣ Interview Questions & Answers

Q1: What is a decorator?

A function that modifies another function.

Q2: Why `@wraps` is used?

To preserve original function metadata.

Q3: Can a decorator change return value?

✅ Yes

Q4: Are decorators executed at runtime?

Decorators run **at function definition time**.

✅ Final Summary

- Decorators extend function behavior
- Built using closures
- `@` syntax = clean code
- Widely used in frameworks
- Interview-critical topic

🚀 Next Topics (Pick One)

- **Class decorators**
- **Built-in decorators (`@staticmethod`, `@classmethod`)**
- **Decorators vs middleware**
- **Advanced decorator interview problems**

🐍 OOPS in Python

1️⃣ What is OOPS?

Simple definition

Object-Oriented Programming (OOP) is a programming style that organizes code using **classes and objects**.

Interview one-liner

> OOP is a programming paradigm based on objects that contain data and methods.

2️⃣ Class and Object

🔷 Class

A **class** is a **blueprint** for creating objects.

```
classPerson:
pass
```

🔷 Object

An **object** is an **instance of a class**.

```
p1 = Person()
```

📌 You can create multiple objects from one class.

3️⃣ Example: Class & Object

```
classPerson:
    name ="Unknown"# class variable

p1 = Person()
p2 = Person()

print(p1.name)
print(p2.name)
```

4️⃣ Constructor (`__init__`) 🔥

What is a constructor?

A **constructor** is a special method that runs **automatically when an object is created**.

```
classPerson:
def__init__(self, name, age):
self.name = name# instance variable
self.age = age

p1 = Person("Swalih",25)
```

📌 `self` → refers to the **current object**.

5️⃣ Methods in a Class

What is a method?

A **method** is a function defined inside a class.

```
classPerson:
defgreet(self):
print("Hello")
```

6️⃣ Types of Variables (VERY IMPORTANT 🔥)

🔷 1. Instance Variables

- Belong to an **object**
- Defined using `self`

```
classPerson:
def__init__(self, name):
self.name = name
```

Each object has its **own copy**.

🔷 2. Static / Class Variables

- Shared among **all objects**
- Defined outside methods

```
classPerson:
    species ="Human"
```

Access:

```
Person.species
```

🔷 3. Local Variables

- Defined **inside methods**
- Exist only during method execution

```
classTest:
defshow(self):
```

```
    x =10# local variable
```

7️⃣ Types of Methods (INTERVIEW FAVORITE 🔥)

🔷 1. Instance Method

- Works with **object data**
- Uses `self`

```
classPerson:
defgreet(self):
print(self.name)
```

🔷 2. Class Method

- Works with **class data**
- Uses `cls`
- Defined using `@classmethod`

```
classPerson:
    species ="Human"

    @classmethod
defshow_species(cls):
print(cls.species)
```

🔷 3. Static Method

- Utility function
- No `self` or `cls`
- Defined using `@staticmethod`

```
classMathUtils:
    @staticmethod
defadd(a, b):
return a + b
```

8️⃣ Complete Example (ALL TOGETHER 🔥)

```
classStudent:
    school ="ABC School"# class variable

def__init__(self, name, marks):
self.name = name# instance variable
self.marks = marks

defdisplay(self):# instance method
print(self.name,self.marks)

    @classmethod
```

```
defchange_school(cls, new_name):
    cls.school = new_name

  @staticmethod
defis_pass(marks):
return marks >=40
```

Usage

```
s1 = Student("A",85)
s2 = Student("B",35)

s1.display()
print(Student.is_pass(s1.marks))

Student.change_school("XYZ School")
```

9️⃣ Comparison Table (INTERVIEW 🔥)

Variables

Type	Belongs to
Instance	Object
Class	Class
Local	Method

Methods

Method	Uses
Instance	`self`
Class	`cls`
Static	None

🔟 Common Mistakes ⚠️

❌ Forgetting `self`

❌ Accessing instance variables via class

❌ Using static method for object data

❌ Confusing class & instance variables

1️⃣1️⃣ Mental Model (Remember Forever 🧠)

- Class → **Blueprint**
- Object → **Real thing**
- Instance variable → **Personal data**
- Class variable → **Shared data**
- Instance method → **Uses object**
- Class method → **Uses class**
- Static method → **Helper**

1️⃣2️⃣ Interview Questions & Answers

Q1: Why `self` is required?

To refer to the current object.

Q2: Can static method access instance data?

❌ No

Q3: When to use class method?

When logic is related to the class, not object.

✅ Final Summary

- OOP organizes code using classes & objects
- Constructors initialize objects
- Variables & methods have types
- Core topic for interviews

🚀 Next OOPS Topics (Pick One)

- **Inheritance**
- **Encapsulation**
- **Polymorphism**
- **Abstraction**

🐍 Inheritance

1️⃣ What is Inheritance?

Simple definition

Inheritance allows a class (**child**) to **reuse properties and methods** of another class (**parent**).

Interview one-liner

> Inheritance is an OOP concept where a child class acquires the properties and behavior of a parent class.

2️⃣ Why Inheritance is used

- Code reusability
- Avoid duplication
- Easy maintenance
- Logical class hierarchy
- Polymorphism support

3️⃣ Basic Syntax

```
classParent:
pass

classChild(Parent):
pass
```

📌 `Child` **inherits** from `Parent`.

5️⃣ Constructor in Inheritance (`super()`) ⭐

Parent constructor

```
classAnimal:
def__init__(self, name):
self.name = name
```

Child constructor

```
classDog(Animal):
def__init__(self, name, breed):
super().__init__(name)
self.breed = breed
```

📌 `super()` calls **parent constructor**.

Types of Inheritance

Inheritance allows a class to **reuse** properties and methods of another class.

1️⃣ Single Inheritance

Definition

A **single child class** inherits from **one parent class**.

Diagram

```
Parent → Child
```

Example

```
classAnimal:
defspeak(self):
print("Animal speaks")

classDog(Animal):
defbark(self):
print("Dog barks")

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

📌 Most common & simplest form.

2️⃣ Multilevel Inheritance

Definition

A class is derived from a class which is **already derived** from another class.

Diagram

```
Grandparent → Parent → Child
```

Example

```
classA:
defshowA(self):
print("Class A")

classB(A):
defshowB(self):
print("Class B")

classC(B):
defshowC(self):
print("Class C")

c = C()
c.showA()
c.showB()
c.showC()
```

📌 Child gets access to **all ancestors**.

3️⃣ Hierarchical Inheritance

Definition

Multiple child classes inherit from **one parent class**.

Diagram

```
    Parent
    /   \
  Child1  Child2
```

Example

```
classAnimal:
defeat(self):
print("Eating")

classDog(Animal):
defbark(self):
print("Bark")

classCat(Animal):
defmeow(self):
print("Meow")

d = Dog()
c = Cat()
d.eat()
c.eat()
```

📌 One parent → many children.

4️⃣ Multiple Inheritance ⚠️

Definition

A child class inherits from **more than one parent class**.

Diagram

```
Parent1   Parent2
    \   /
     Child
```

Example

```
classFather:
defskill(self):
print("Driving")

classMother:
defskill(self):
print("Cooking")

classChild(Father, Mother):
pass

c = Child()
c.skill()
```

📌 Python resolves conflict using **MRO**.

5️⃣ Hybrid Inheritance

Definition

A combination of **two or more types of inheritance**.

Diagram

```
A
      / \
B   C
      \ /
       D
```

Example

```
classA:
defshowA(self):
print("A")

classB(A):
defshowB(self):
print("B")
```

```
classC(A):
defshowC(self):
print("C")

classD(B, C):
defshowD(self):
print("D")

d = D()
d.showA()
```

📌 Uses **multiple + hierarchical inheritance**.

🔥 Diamond Problem (INTERVIEW FAVORITE)

Occurs in **multiple inheritance**.

```
classA:
defshow(self):
print("A")

classB(A):
defshow(self):
print("B")

classC(A):
defshow(self):
print("C")

classD(B, C):
pass

d = D()
d.show()
```

Output

```
B
```

📌 Because of **MRO**.

```
print(D.mro())
```

🧠 MRO (Method Resolution Order)

Python follows:

```
Child → Parent1 → Parent2 →object
```

Uses **C3 Linearization**.

🔍 Comparison Table (Interview 🔥)

Type	Parents	Children
Single	1	1
Multilevel	1	1
Hierarchical	1	Many
Multiple	Many	1
Hybrid	Many	Many

⚠️ Common Interview Traps

❌ Confusing multiple & multilevel

❌ Ignoring MRO

❌ Overusing multiple inheritance

❌ Forgetting `super()`

🧠 Mental Model (Remember Forever)

- Single → One road
- Multilevel → Ladder
- Hierarchical → Tree
- Multiple → Merge roads
- Hybrid → Mix of all

✅ Interview One-Line Answer

> Python supports five types of inheritance: Single, Multilevel, Hierarchical, Multiple, and Hybrid, with method resolution handled using MRO.

🚀 Next OOPS Topics (Recommended)

- **Encapsulation**
- **Polymorphism**
- **Abstraction**
- **Diamond problem deep dive**

🐍 Polymorphism

1️⃣ What is Polymorphism?

Simple definition

Polymorphism means **"many forms"**.

Interview one-liner

> Polymorphism allows the same method name to behave differently based on the object or context.

2️⃣ Why Polymorphism is used

- Code flexibility
- Reusability
- Loose coupling

- Cleaner & scalable design

3️⃣ Polymorphism in Python (Important Note 🔥)

📌 Python does **NOT** support:

- Method overloading like Java (same name, different params)

📌 Python **DOES** support:

- Method overriding
- Duck typing
- Operator overloading
- Function polymorphism

4️⃣ Types of Polymorphism in Python

🔷 1. Method Overriding (Runtime Polymorphism)

Definition

Child class provides its **own version** of parent method.

Example

```
classAnimal:
defsound(self):
print("Animal sound")

classDog(Animal):
defsound(self):
print("Bark")

classCat(Animal):
defsound(self):
print("Meow")
```

Usage

```
animals = [Dog(), Cat()]

for ain animals:
    a.sound()
```

Output

```
Bark
Meow
```

📌 Same method name → different behavior.

🔷 2. Duck Typing (Python-Specific 🔥)

Definition

> If it looks like a duck and quacks like a duck, it’s a duck.

Python cares about **behavior**, not type.

Example

```
classCar:
defmove(self):
print("Car moving")

classBike:
defmove(self):
print("Bike moving")

deftravel(vehicle):
    vehicle.move()

travel(Car())
travel(Bike())
```

📌 No inheritance required.

🔷 3. Function Polymorphism (Built-in)

Same function works with different data types.

Example

```
print(len("Python"))
print(len([1,2,3]))
print(len((1,2)))
```

🔷 4. Operator Overloading

Definition

Same operator behaves differently for different objects.

Example

```
print(10 +20)
print("Hello " +"World")
print([1,2] + [3,4])
```

Custom Operator Overloading 🔥

```
classPoint:
def__init__(self, x):
self.x = x

def__add__(self, other):
returnself.x + other.x

p1 = Point(10)
p2 = Point(20)
```

```
print(p1 + p2)# 30
```

🔷 5. Method Overloading (Python Style ⚠️)

Python does NOT support traditional overloading.

Instead use:

- Default arguments
- `args`

Example

```
defadd(a, b=0):
return a + b

print(add(5))
print(add(5,3))
```

5️⃣ Polymorphism with Inheritance (Core 🔥)

```
classShape:
defarea(self):
pass

classSquare(Shape):
defarea(self):
print("Square area")

classCircle(Shape):
defarea(self):
print("Circle area")
```

6️⃣ Real-World Example 🔥

```
classPayment:
defpay(self):
pass

classUPI(Payment):
defpay(self):
print("Paid via UPI")

classCard(Payment):
defpay(self):
print("Paid via Card")

defprocess(payment):
    payment.pay()

process(UPI())
```

```
process(Card())
```

7️⃣ Polymorphism vs Inheritance (Interview 🔥)

Inheritance	Polymorphism
IS-A relationship	Many forms
Code reuse	Behavior flexibility
Structure	Behavior

8️⃣ Common Mistakes ⚠️

❌ Thinking inheritance is required

❌ Confusing overloading & overriding

❌ Using `isinstance()` instead of polymorphism

❌ Writing long if-else instead of polymorphism

9️⃣ Mental Model (Remember Forever 🧠)

- Same **method name**
- Different **objects**
- Different **behavior**

🔟 Interview Questions & Answers

Q1: What is polymorphism?

Same interface, different behavior.

Q2: Does Python support method overloading?

❌ Not traditionally.

Q3: What is duck typing?

Behavior-based typing.

Q4: What is operator overloading?

Redefining operators using magic methods.

✅ Final Summary

- Polymorphism = many forms
- Python supports runtime polymorphism
- Duck typing is powerful
- Core OOPS pillar
- Heavy interview topic

🚀 Next OOPS Topics (Recommended)

- **Encapsulation**
- **Abstraction**
- **Magic methods (`__str__`, `__len__`)**
- **Real-world OOPS project**

🐍 Abstraction

1️⃣ What is Abstraction?

Simple definition

Abstraction means **hiding implementation details and showing only essential features**.

Interview one-liner

> Abstraction focuses on what an object does, not how it does it.

📌 User knows **what to use**, not **how it works internally**.

2️⃣ Why Abstraction is used

- Reduce complexity
- Hide internal logic
- Improve security
- Enforce structure
- Support polymorphism

3️⃣ Real-World Analogy 🧠

ATM Machine

- You know: **withdraw(), deposit()**
- You don't know: **internal banking logic**

That's abstraction.

4️⃣ How Abstraction is implemented in Python

Python supports abstraction using:

1. **Abstract Base Classes (ABC)**
2. **Abstract methods**
3. **Interfaces-like behavior**

5️⃣ Abstract Base Class (ABC)

Step 1: Import ABC module

```
from abcimport ABC, abstractmethod
```

Step 2: Create abstract class

```
classShape(ABC):
    @abstractmethod
defarea(self):
pass
```

📌 Abstract class **cannot be instantiated**.

6️⃣ Implementing Abstract Methods (IMPORTANT 🔥)

```
classSquare(Shape):
defarea(self):
print("Area of square")
```

```
classCircle(Shape):
defarea(self):
print("Area of circle")
```

Usage

```
s = Square()
c = Circle()

s.area()
c.area()
```

7️⃣ What happens if method is NOT implemented? ⚠️

```
classTriangle(Shape):
pass

t = Triangle()# ❌ TypeError
```

📌 Python **forces implementation**.

8️⃣ Abstraction with Constructor 🔥

```
classVehicle(ABC):
def__init__(self, name):
self.name = name

    @abstractmethod
defmove(self):
pass
```

```
classCar(Vehicle):
defmove(self):
print(self.name,"moves on road")
```

9️⃣ Abstraction vs Encapsulation (INTERVIEW 🔥)

Abstraction	Encapsulation
Hide logic	Hide data
Interface	Implementation
What to do	How to protect

Abstraction	Encapsulation
Uses ABC	Uses access control

🔟 Interface in Python? (IMPORTANT)

Python does **not have true interfaces** like Java.

👉 **Abstract classes act as interfaces**.

1️⃣1️⃣ Multiple Abstract Methods

```
classPayment(ABC):
    @abstractmethod
defpay(self):
pass

    @abstractmethod
defrefund(self):
pass
```

1️⃣2️⃣ Abstraction + Polymorphism 🔥

```
classPayment(ABC):
    @abstractmethod
defpay(self):
pass

classUPI(Payment):
defpay(self):
print("Paid via UPI")

classCard(Payment):
defpay(self):
print("Paid via Card")

defprocess(payment):
    payment.pay()
```

1️⃣3️⃣ Common Mistakes ⚠️

❌ Trying to create object of abstract class

❌ Forgetting to implement all abstract methods

❌ Confusing abstraction with encapsulation

❌ Overusing abstraction

1️⃣4️⃣ Mental Model (Remember Forever 🧠)

- Abstraction = **Remote control**
- You press buttons
- TV handles logic

1️⃣5️⃣ Interview Questions & Answers

Q1: Can we create object of abstract class?

❌ No

Q2: Can abstract class have normal methods?

✅ Yes

Q3: Does Python support interfaces?

❌ Not directly (uses ABC)

Q4: Why abstraction is important?

Cleaner, safer, scalable code

✅ Final Summary

- Abstraction hides implementation
- Uses `abc` module
- Enforces method implementation
- Supports polymorphism
- Core OOPS pillar

🚀 What's Next? (Recommended Order)

- **Encapsulation**
- **Magic / Dunder methods**
- **Real-world OOPS mini project**
- **OOPS interview questions**

🐍 Encapsulation

1️⃣ What is Encapsulation?

Simple definition

Encapsulation means **wrapping data and methods together** and **controlling access to data**.

Interview one-liner

> Encapsulation is the process of restricting direct access to data and exposing it through methods.

2️⃣ Why Encapsulation is used

- Protect data from misuse
- Improve security
- Control modification
- Maintain integrity
- Clean API design

3️⃣ Encapsulation in Python (Important Note 🔥)

⚠️ Python does **not** have strict access modifiers like Java (`private` , `protected`).

Instead, Python uses **naming conventions**.

4️⃣ Access Modifiers in Python

Python has **3 levels of access**:

Modifier	Syntax	Meaning
Public	`var`	Accessible everywhere
Protected	`_var`	Internal use (convention)
Private	`__var`	Name mangling

5️⃣ Public Members

```
classUser:
def__init__(self, name):
self.name = name# public

u = User("Swalih")
print(u.name)
```

✔️ Accessible from anywhere

6️⃣ Protected Members (`_var`) 🟡

```
classUser:
def__init__(self, age):
self._age = age
```

📌 Means: **"Use internally or in subclass"**

```
classAdmin(User):
defshow_age(self):
print(self._age)
```

⚠️ Still accessible, but **should not be used directly**.

7️⃣ Private Members (`__var`) 🔥

```
classUser:
def__init__(self, password):
self.__password = password
```

❌ Direct access not allowed:

```
u.__password# Error
```

Name Mangling (IMPORTANT 🔥)

Python internally changes:

```
__password → _User__password
```

```
print(u._User__password)# Works
```

📌 This is **name mangling**, not true privacy.

8️⃣ Encapsulation Using Getter & Setter (BEST PRACTICE 🔥)

```
classUser:
def__init__(self):
self.__age =0

defset_age(self, age):
if age <0:
print("Invalid age")
else:
self.__age = age

defget_age(self):
returnself.__age
```

Usage

```
u = User()
u.set_age(25)
print(u.get_age())
```

9️⃣ Pythonic Way: `@property` Decorator ⭐⭐⭐

🔥 **Interview favorite**

```
classUser:
def__init__(self):
self.__age =0

    @property
defage(self):
returnself.__age

    @age.setter
defage(self, value):
if value <0:
raise ValueError("Age cannot be negative")
self.__age = value
```

Usage

```
u = User()
u.age =25
print(u.age)
```

📌 Looks like variable access but uses methods.

🔟 Encapsulation vs Abstraction (Interview 🔥)

Encapsulation	Abstraction
Data hiding	Logic hiding
Access control	Interface design
How data is protected	What operations are allowed

1️⃣1️⃣ Real-World Example 🔥

```
classBankAccount:
def__init__(self):
self.__balance =0

defdeposit(self, amount):
self.__balance += amount

defwithdraw(self, amount):
if amount <=self.__balance:
self.__balance -= amount
else:
print("Insufficient funds")

defget_balance(self):
returnself.__balance
```

1️⃣2️⃣ Common Mistakes ⚠️

❌ Accessing private variables directly

❌ Confusing `_var` with `__var`

❌ Overusing getters/setters

❌ Ignoring `@property`

1️⃣3️⃣ Mental Model (Remember Forever 🧠)

- Encapsulation = **Capsule**
- Data inside
- Access through controlled doors (methods)

1️⃣4️⃣ Interview Questions & Answers

Q1: Does Python support private variables?

⚠️ Not truly, uses name mangling.

Q2: Difference between `_var` and `__var`?

- `_var` → convention
- `__var` → name mangling

Q3: Best way to implement encapsulation?

Using `@property`

✅ Final Summary

- Encapsulation protects data

- Python uses naming conventions
- `__var` for private data
- `@property` is the Pythonic way
- Core OOPS pillar

🚀 What's Next?

- **Magic (Dunder) Methods**
- **OOPS Interview Questions**
- **Real-World OOPS Project**
- **Design Patterns (Basics)**

🐍 Iterable and Iterator

1️⃣ What is an Iterable?

Simple definition

An **iterable** is an object that **can be looped over**.

Interview one-liner

> An iterable is an object that implements the __iter__() method.

Examples of Iterables

```
list,tuple,set,dict, string,range
```

```
nums = [1,2,3]# iterable
```

You can do:

```
for xin nums:
print(x)
```

2️⃣ What is an Iterator?

Simple definition

An **iterator** is an object that **returns one element at a time**.

Interview one-liner

> An iterator is an object that implements both __iter__() and __next__().

Example

```
nums = [1,2,3]
it =iter(nums)# iterator

print(next(it))# 1
```

```
print(next(it))# 2
print(next(it))# 3
```

After last element:

```
next(it)# StopIteration
```

3️⃣ Key Difference (Core Idea 🔥)

> Iterable = something you can get an iterator from
>
> **Iterator = something that gives values one by one**

4️⃣ How `for` loop REALLY works (INTERVIEW GOLD 🔥)

```
for xin nums:
print(x)
```

Internally:

```
it =iter(nums)
whileTrue:
try:
        x =next(it)
print(x)
except StopIteration:
break
```

📌 This is why iterators are important.

5️⃣ Check Iterable vs Iterator

```
from collections.abcimport Iterable, Iterator

isinstance(nums, Iterable)# True
isinstance(nums, Iterator)# False
```

```
isinstance(it, Iterator)# True
```

6️⃣ Can an Object be BOTH?

✅ Yes.

Example: `range`

```
r =range(5)
```

- `r` is iterable
- `iter(r)` returns an iterator

But:

```
it =iter(r)
```

- `it` is an iterator
- `it` is also iterable (iterator returns itself in `__iter__()`)

7️⃣ One-time vs Multi-time Use 🔥

Iterable

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

for xin nums:
print(x)

for xin nums:
print(x)# works again
```

Iterator

```
it =iter(nums)

for xin it:
print(x)

for xin it:
print(x)# ❌ nothing prints
```

📌 Iterators are **exhausted**.

8️⃣ Creating Your Own Iterator (IMPORTANT 🔥)

```
classCount:
def__init__(self, max):
self.max =max
self.current =0

def__iter__(self):
returnself

def__next__(self):
ifself.current <self.max:
self.current +=1
returnself.current
else:
raise StopIteration
```

Usage

```
c = Count(3)
for xin c:
```

```
print(x)
```

9️⃣ Iterable vs Iterator (INTERVIEW TABLE 🔥)

Feature	Iterable	Iterator
Can loop	✅	✅
Has `__iter__()`	✅	✅
Has `__next__()`	❌	✅
Stores all data	Usually	No
One-time use	❌	✅

🔟 Real-World Analogy 🧠

- **Iterable** → Playlist
- **Iterator** → Music player

You can restart playlist,

but player continues where it stopped.

1️⃣1️⃣ Common Mistakes ⚠️

❌ Thinking iterable = iterator

❌ Reusing exhausted iterator

❌ Forgetting `StopIteration`

❌ Overusing iterators when list is fine

1️⃣2️⃣ Interview Questions & Answers

Q1: Is list an iterator?

❌ No, it's iterable.

Q2: Why iterators are memory efficient?

They generate values **on demand**.

Q3: Can iterator be reset?

❌ No (create a new one).

✅ Final Summary

- Iterable → can produce iterator
- Iterator → produces values one by one
- Iterators are one-time use
- `for` loop uses iterator internally
- Very important interview topic

🚀 Next Topics (Recommended)

- **Generators vs Iterators**
- `yield` **keyword**
- **Lazy evaluation**
- **Memory optimization in Python**

🐍 Generators

1️⃣ What is a Generator?

Simple definition

A **generator** is a special function that **produces values one at a time** using the `yield` keyword.

Interview one-liner

> A generator is a function that returns an iterator and yields values lazily.

2️⃣ Why Generators are used

- Memory efficiency
- Lazy evaluation
- Handle large data
- Improve performance
- Simplify iterator code

3️⃣ Generator vs Normal Function (CORE 🔥)

Normal function

```
defnumbers():
return [1,2,3]
```

- Executes fully
- Stores all data in memory

Generator function

```
defnumbers():
yield1
yield2
yield3
```

- Executes **step by step**
- Does not store all values

4️⃣ How `yield` Works (VERY IMPORTANT 🔥)

- `yield` pauses function execution
- Saves function state
- Resumes from where it stopped

```
defgen():
print("Start")
yield1
print("Middle")
yield2
```

```
print("End")
```

Execution

```
g = gen()
next(g)# Start → 1
next(g)# Middle → 2
next(g)# End → StopIteration
```

5️⃣ Generator Example (Basic)

```
defcount_up(n):
    i =1
while i <= n:
yield i
        i +=1
```

```
for numin count_up(5):
print(num)
```

6️⃣ Generator vs Iterator (INTERVIEW 🔥)

Generator	Iterator
Easy to write	Complex
Uses `yield`	Uses `__iter__` & `__next__`
Less code	More boilerplate
Lazy	Lazy

📌 Generator **is a type of iterator**.

7️⃣ Generator Expression (Pythonic 🔥)

Like list comprehension but **lazy**.

```
gen = (x*xfor xinrange(5))
```

```
for valin gen:
print(val)
```

📌 Uses `()` instead of `[]`.

8️⃣ Memory Efficiency (IMPORTANT 🔥)

```
list(range(1000000))# heavy memory
range(1000000)# generator-like
```

📌 Generator generates one value at a time.

9️⃣ Sending Values to Generator (`send()`) 🔥

```
defcounter():
    i =0
whileTrue:
        x =yield i
if xisnotNone:
            i = x
        i +=1
```

1️⃣0️⃣ Closing Generator

```
g.close()
```

Raises `GeneratorExit`.

1️⃣1️⃣ Common Mistakes ⚠️

❌ Using generator twice

❌ Forgetting generator is exhausted

❌ Expecting random access

❌ Confusing list comprehension with generator expression

1️⃣2️⃣ Real-World Use Cases 🔥

- Reading large files line by line
- Streaming data
- Infinite sequences
- Pipeline processing

1️⃣3️⃣ Mental Model (Remember Forever 🧠)

- Generator = **Pause & Resume machine**
- `yield` = **Pause button**
- `next()` = **Resume**

1️⃣4️⃣ Interview Questions & Answers

Q1: Is generator iterable?

✅ Yes

Q2: Is generator an iterator?

✅ Yes

Q3: Difference between `yield` and `return`?

yield	return
Pauses	Ends
Multiple values	Single value

Q4: Can generator be reused?

❌ No

✅ Final Summary

- Generators produce values lazily
- Use `yield`
- Memory efficient
- Simplify iterator creation
- Important interview topic

🚀 Next Topics (Recommended)

- **Generator vs List comprehension**
- **`yield from`**
- **Async generators**
- **File handling with generators**

Extra Topic

🐍 Magic Methods

1️⃣ What are Magic Methods?

Simple definition

Magic methods are special methods in Python that **start and end with double underscores** (`__`).

Interview one-liner

> Magic methods allow us to define how objects behave with built-in operations.

Examples:

```
__init__, __str__, __len__, __add__
```

📌 Also called **dunder methods**.

2️⃣ Why Magic Methods are used

- Customize object behavior
- Operator overloading
- Make objects iterable
- Improve debugging & readability
- Integrate with Python internals

3️⃣ Constructor Magic Method: `__init__` 🔥

Called **automatically** when object is created.

```
classUser:
def__init__(self, name):
self.name = name
```

```
u = User("Swalih")
```

4️⃣ String Representation: `__str__` & `__repr__` ⭐

`__str__` (User-friendly)

```
classUser:
def__init__(self, name):
self.name = name

def__str__(self):
returnf"User name is {self.name}"

print(User("Swalih"))
```

`__repr__` (Developer-friendly)

```
def__repr__(self):
returnf"User({self.name})"
```

📌 If `__str__` is missing, Python uses `__repr__`.

5️⃣ Length Magic Method: `__len__`

```
classMyList:
def__init__(self, items):
self.items = items

def__len__(self):
returnlen(self.items)

print(len(MyList([1,2,3])))
```

6️⃣ Operator Overloading (INTERVIEW FAVORITE 🔥)

`__add__` (`+`)

```
classPoint:
def__init__(self, x):
self.x = x

def__add__(self, other):
returnself.x + other.x

p1 = Point(10)
p2 = Point(20)

print(p1 + p2)
```

Common Operator Methods

Operator	Method
`+`	`__add__`
`-`	`__sub__`
`*`	`__mul__`
`/`	`__truediv__`
`==`	`__eq__`
`<`	`__lt__`

7️⃣ Comparison Magic Methods

```
classStudent:
def__init__(self, marks):
self.marks = marks

def__gt__(self, other):
returnself.marks > other.marks

s1 = Student(80)
s2 = Student(70)

print(s1 > s2)
```

8️⃣ Callable Objects: `__call__` 🔥

Make objects behave like functions.

```
classAdder:
def__call__(self, a, b):
return a + b

add = Adder()
print(add(2,3))
```

9️⃣ Attribute Access Magic Methods

`__getattr__`

Called when attribute is **not found**.

```
classTest:
def__getattr__(self, name):
returnf"{name} not found"

t = Test()
print(t.age)
```

`__setattr__`

Called when setting attributes.

```
def__setattr__(self, name, value):
print("Setting", name)
```

```
self.__dict__[name] = value
```

🔟 Iteration Magic Methods 🔥

Make object iterable

```
classCount:
def__init__(self, max):
self.max =max
self.current =0

def__iter__(self):
returnself

def__next__(self):
ifself.current <self.max:
self.current +=1
returnself.current
raise StopIteration
```

1️⃣1️⃣ Context Manager: `__enter__` & `__exit__` 🔥

```
classFileManager:
def__enter__(self):
print("Enter")
returnself

def__exit__(self, exc_type, exc_val, exc_tb):
print("Exit")
```

Used in:

```
with FileManager():
print("Inside")
```

1️⃣2️⃣ Important Magic Methods (Must Know 🔥)

Method	Purpose
`__init__`	Constructor
`__str__`	Print output
`__repr__`	Debug
`__len__`	Length
`__add__`	`+`
`__eq__`	`==`
`__call__`	Callable
`__iter__`	Iterator
`__next__`	Next value
`__enter__`	Context
`__exit__`	Context

1️⃣3️⃣ Common Mistakes ⚠️

❌ Forgetting to return value

❌ Infinite recursion in `__setattr__`

❌ Overusing magic methods

❌ Confusing `__str__` and `__repr__`

1️⃣4️⃣ Mental Model (Remember Forever 🧠)

- Magic methods = **Hooks**
- Python calls them **automatically**
- You just define behavior

1️⃣5️⃣ Interview Questions & Answers

Q1: What are magic methods?

Special methods that define object behavior.

Q2: When is `__init__` called?

At object creation.

Q3: Difference between `__str__` and `__repr__`?

User vs developer output.

Q4: Can we overload operators in Python?

✅ Yes, using magic methods.

✅ Final Summary

- Magic methods customize object behavior
- Enable operator overloading
- Used by Python internally
- Important for advanced OOPS & interviews

🚀 Next Topics (Recommended)

- **Context managers (`with`) deep dive**
- `dataclass` **& magic methods**
- **Real-world OOPS mini project**
- **Advanced interview questions on dunder methods**

🐍 Monkey Patching

1️⃣ What is Monkey Patching?

Simple definition

Monkey patching means **modifying or extending existing code at runtime** without changing the original source code.

Interview one-liner

> Monkey patching is the technique of dynamically modifying a class or module at runtime.

📌 Python allows this because **everything is an object**.

2️⃣ Why Monkey Patching is possible in Python

Because Python is:

- Dynamically typed
- Interpreted
- Runtime flexible
- Object-oriented

You can:

- Add methods
- Replace functions
- Modify class behavior **after definition**

3️⃣ Basic Monkey Patching Example (Function)

```
classA:
defshow(self):
print("Original show")

defnew_show(self):
print("Patched show")

A.show = new_show# monkey patch
```

Usage

```
a = A()
a.show()
```

Output

```
Patchedshow
```

📌 We replaced `show()` **at runtime**.

4️⃣ Monkey Patching an Object (Instance-level)

```
classA:
defgreet(self):
print("Hello")

a = A()

defnew_greet():
print("Hi")

a.greet = new_greet# patch only this object
a.greet()
```

📌 Only `a` is affected, not other instances.

5️⃣ Monkey Patching a Module

```
import math

deffake_sqrt(x):
return"Not allowed"

math.sqrt = fake_sqrt
print(math.sqrt(9))
```

⚠️ Dangerous if misused.

6️⃣ Real-World Use Cases (LEGIT 🔥)

✅ Testing (Mocking)

```
deffake_api():
return"Fake response"

api.call = fake_api
```

Used in:

- Unit testing
- Mocking external APIs

✅ Hotfixing Bugs

- Temporary patch without redeploying full system

✅ Adding logging / monitoring

```
original = func

defwrapper():
print("Log")
    original()
```

7️⃣ Monkey Patching vs Decorators (INTERVIEW 🔥)

Monkey Patching	Decorator
Runtime change	Compile-time (definition-time)
Global impact	Local to function
Risky	Safer
Less readable	Cleaner

📌 Prefer **decorators** when possible.

8️⃣ Risks & Disadvantages ⚠️

❌ Hard to debug

❌ Unexpected behavior

❌ Breaks library updates
❌ Global side effects
❌ Poor readability

9️⃣ Best Practices (IMPORTANT 🔥)

✔ Use only when unavoidable
✔ Document clearly
✔ Prefer mocking libraries (`unittest.mock`)
✔ Avoid patching core libraries
✔ Limit scope (instance-level)

🔟 Monkey Patching with `unittest.mock` (SAFE WAY ⭐)

```
from unittest.mockimport patch

with patch("math.sqrt", return_value=100):
print(math.sqrt(9))
```

📌 Automatically restored after block.

1️⃣1️⃣ Mental Model (Remember Forever 🧠)

- Monkey patching = **Changing engine while car is running**
- Powerful but dangerous

1️⃣2️⃣ Interview Questions & Answers

Q1: What is monkey patching?

Runtime modification of code behavior.

Q2: Is monkey patching good practice?

❌ Generally no, except testing.

Q3: Why Python allows monkey patching?

Dynamic and object-based design.

Q4: Safer alternative?

Decorators or mocking frameworks.

✅ Final Summary

- Monkey patching modifies behavior at runtime
- Python allows it due to dynamic nature
- Useful in testing & hotfixes
- Dangerous if abused
- Prefer safer alternatives

🚀 Next Advanced Topics (Pick One)

- `unittest.mock` **deep dive**
- **Metaclasses**

- **Descriptors**
- **Python internals (how objects work)**

🐍 Python Core Concepts (Interview-Ready Guide)

1️⃣ Single-Threaded vs Multi-Threading in Python

🔷 What does "Single-Threaded" mean?

Simple definition

A **single-threaded program** executes **one task at a time**, sequentially.

```
print("Task 1")
print("Task 2")
```

📌 Only one execution path.

🔷 Is Python Single-Threaded?

👉 **Python itself is NOT single-threaded**,

👉 but **CPython (default Python)** has **GIL (Global Interpreter Lock)**.

🔥 Global Interpreter Lock (GIL)

- Only **one thread executes Python bytecode at a time**
- Even on multi-core CPUs

📌 So:

- **CPU-bound tasks** → behave like single-threaded
- **I/O-bound tasks** → benefit from multithreading

🔷 Multi-Threading in Python

Simple definition

Multithreading allows multiple threads to exist within one process.

```
import threading

deftask():
print("Running task")

t = threading.Thread(target=task)
t.start()
```

Where multithreading is useful

✅ File I/O

✅ Network requests

✅ API calls

✅ Waiting tasks

Where it is NOT useful

❌ Heavy computations (use multiprocessing)

Interview one-liner

> Python supports multithreading, but due to the GIL, it is best suited for I/O-bound tasks, not CPU-bound tasks.

2️⃣ Multi-Paradigm Language (VERY IMPORTANT 🔥)

🔷 What does Multi-Paradigm mean?

Simple definition

A **multi-paradigm language** supports **multiple programming styles**.

🔷 Paradigms Supported by Python

🟢 Procedural Programming

```
defadd(a, b):
return a + b
```

🔵 Object-Oriented Programming (OOP)

```
classUser:
deflogin(self):
pass
```

🟣 Functional Programming

```
nums =list(map(lambda x: x*x, [1,2,3]))
```

🟠 Imperative Programming

```
x =10
x +=1
```

Interview one-liner

> Python is a multi-paradigm language because it supports procedural, object-oriented, and functional programming styles.

3️⃣ Automatic Memory Management 🔥

🔷 What is Automatic Memory Management?

Simple definition

Python **automatically allocates and deallocates memory**—the programmer doesn't do it manually.

🔷 How Python Manages Memory

1️⃣ Reference Counting

```
a = []
b = a
```

Object deleted when reference count = 0.

2️⃣ Garbage Collection

Handles **circular references**.

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

Interview one-liner

> Python uses automatic memory management through reference counting and garbage collection.

🔷 Why this is important

✅ Prevents memory leaks

✅ Safer code

❌ Slight performance overhead

4️⃣ Integration and Extensibility 🔥

🔷 What does Integration mean?

Python can **work with other languages and systems**.

Examples:

- Call C/C++ code
- Use Java libraries
- Interact with OS, databases, APIs

🔷 What does Extensibility mean?

Python can be **extended using other languages**.

Examples:

- NumPy uses C internally
- TensorFlow uses C++ backend

Example: C extension concept

```
Python → C API → Native code
```

Interview one-liner

> Python is highly integrable and extensible, allowing it to interface with and be extended by C, C++, and other languages.

5️⃣ Dynamically Typed Language 🔥

🔷 What is Dynamic Typing?

Simple definition

You **don't declare variable types**—Python decides at runtime.

```
x =10
x ="Hello"
```

🔷 How Python handles this

- Variables are **references**
- Objects have **types**
- Type checking happens **at runtime**

Dynamic vs Static Typing

Feature	Python	C / Java
Type declaration	❌ No	✅ Yes
Flexibility	High	Low
Errors	Runtime	Compile-time

Interview one-liner

> Python is dynamically typed because variable types are determined at runtime, not at compile time.

🔥 Combined Interview Summary (MEMORIZE THIS)

> Python is a multi-paradigm, dynamically typed language with automatic memory management. While it supports multithreading, CPython's GIL limits true parallel execution for CPU-bound tasks. Python is also highly integrable and extensible with languages like C and C++.

🧠 Mental Model (Remember Forever)

- **Single thread** → One brain
- **Multithreading** → Multiple hands
- **Multi-paradigm** → Many thinking styles
- **Auto memory** → Self-cleaning house
- **Dynamic typing** → Labels change, boxes don't
- **Extensibility** → Python + Superpowers

🚀 Want to go deeper next?

Pick one 👇

- **GIL deep dive**
- **Multiprocessing vs multithreading**
- **Async programming (async / await)**
- **Python internals (CPython architecture)**