

Table of Contents

1	■ Programming Fundamentals	1
1.1	Why Programming Languages Exist	1
1.2	History & Classification	10
1.3	Why Python	18
2	■ Python Basics	24
2.1	Getting Started	24
2.2	Syntax Essentials	28
2.3	Conclusion & Next Steps	31
3	■ Python Variables	34
3.1	What is a Variable?	34
3.2	Why Do We Use Variables?	34
3.3	Creating a Variable	35
3.4	Variable Naming Rules	36
3.5	Dynamic Typing	37
3.6	Swapping Two Variables	38
3.7	Variable Naming Styles	38
3.8	User Input and Output with Variables	39
3.9	Placement Quick Points	40
3.10	Interview Questions	41
3.11	Practice	41
4	■ Python Keywords	43
4.1	What is a Keyword?	43
4.2	How Many Keywords Are There in Python?	43
4.3	All 35 Keywords Grouped by Category	44

4.4	Placement Quick Points	46
4.5	Interview Questions	47
4.6	Practice	47
5	■ Python Data Types	49
5.1	What is a Data Type?	49
5.2	Data Types at a Glance	50
5.3	Why Do We Need Data Types?	50
5.4	Overview of Python Data Types	51
5.5	Detailed Data Types	52
5.6	Type Conversion (Type Casting)	53
5.7	Checking Data Types with type()	54
5.8	Placement Quick Points	55
5.9	Interview Questions	55
5.10	Practice	56
6	■ Operators	57
6.1	What is an Operator?	57
6.2	What are Operands?	57
6.3	Why Do We Use Operators?	58
6.4	Types of Operators in Python	59
6.5	Arithmetic Operators	60
6.6	Important: / vs // vs %	66
6.7	Comparison Operators	83
6.8	Assignment Operators	87
6.9	Logical Operators	90
6.10	Membership Operators	93
6.11	Identity Operators	96
6.12	Bitwise Operators	98
6.13	Operator Precedence	103

6.14	Complete Placement Eligibility Example	105
6.15	Common Fresher Mistakes	107
6.16	Placement Quick Points	107
6.17	Interview Questions	108
6.18	Practice	110
7	■ Data Structures	112
7.1	Data Structures at a Glance	112
7.2	Strings	112
7.3	Lists	136
7.4	Tuples	153
7.5	Indexing	169
7.6	Slicing	175
7.7	Sets	181
7.8	Dictionaries	203
8	■ Advanced Data Structures	227
8.1	Packing & Unpacking	227
8.2	Comprehensions	232
8.3	Nested Structures	239
9	■ Control Flow	254
9.1	Conditional Statements	254
9.2	Loops	275
9.3	Loop Control Statements	288
9.4	Pattern Matching	291
10	■ Functions	300
10.1	Defining & Calling	301
10.2	Argument Types	306
10.3	Scope & Return	311
10.4	Functional Tools	316

10.5	Built-in vs User-defined	324
11	■ Modules & File Handling	338
11.1	Modules & Packages	338
11.2	File Handling	342
12	■ Error Handling	348
12.1	Exceptions Basics	348
12.2	try / except Blocks	350
12.3	Raising & Custom	354
13	■ Object-Oriented Programming	362
13.1	Classes and Objects	362
13.2	Inheritance	406
13.3	Encapsulation	427
13.4	Polymorphism	447
13.5	Abstraction	464
13.6	Inner Classes	470
14	■ Advanced Python	476
14.1	Iterators	476
14.2	Generators	481
14.3	Decorators	487

Programming Fundamentals

Level: 0 | Version: v1.0 | Author: Meptrasoft

Before writing your first line of Python code, you need to understand **why** programming languages exist. This module uses a **Food Delivery Order System** (like Swiggy or Zomato) as our real-world example to explain: - What a programming language is - Why programming languages were invented - How programming languages evolved over time - Why Python is a great language to start with

TEXT

Copy

HUMAN IDEA → PROGRAMMING LANGUAGE → MACHINE INSTRUCTIONS → COMPUTER ACTION

Why Programming Languages Exist

What Is a Programming Language?

A **programming language** is a set of instructions that humans write and computers execute. It acts as a bridge between **how humans think** (using words, logic, and steps) and **how computers work** (using electrical signals — ON and OFF, represented as **1**s and **0**s).

Real-World Analogy: The Language Translator

Imagine two people trying to communicate: - A person from Tamil Nadu speaks **Tamil** ("வணக்கம்"). - A person from China speaks **Mandarin Chinese** ("你好"). - They cannot understand each other directly because they speak completely different native languages. - **Solution:** They need a **Human Translator** who translates Tamil into Chinese so both sides can understand.

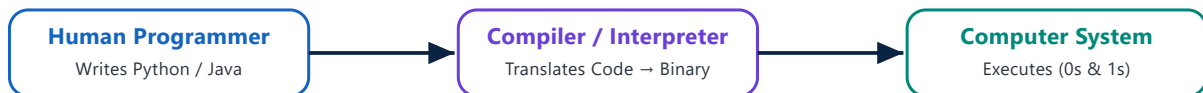
Now apply this exact same idea to computers: - **Humans** communicate using natural languages like English or Tamil. - **Computers** do not understand human languages — their hardware system only understands electrical signals (**1**s and **0**s). - **Solution:** We write our

Ideas in a **Programming Language** (like Python or Java). Then, a translator tool (a **Compiler** or **Interpreter**) translates our high-level code into machine instructions that the computer hardware system understands!

1. Human Language Translation (Tamil to Chinese)



2. Computer Language Translation (Human to Hardware System)



Just as a human translator bridges two foreign languages, a Compiler or Interpreter translates high-level programming code into machine binary.

Analogy: A human translator connects Tamil to Chinese, while a programming language translator (Compiler/Interpreter) connects human code to computer binary hardware.

Think about a food delivery app. A human describes the task in plain English:

TEXT

Copy

"Take the customer's order, calculate the total price, apply a discount, add the delivery fee, and show the final bill."

A computer cannot understand English directly. A programming language allows us to write these steps in a clear, step-by-step format that a computer can understand:

TEXT

Copy

```
GET order
CALCULATE total
APPLY discount
ADD delivery_fee
SHOW final_bill
```

- **Algorithm:** A step-by-step plan or set of rules to solve a problem.
- **Pseudocode:** Writing algorithm steps in simple English before writing actual code. The steps above (GET order, CALCULATE total, etc.) are pseudocode.

Human Language	Programming Language
Flexible, can have different meanings	Strict, precise, and clear
Understood by people	Understood by a compiler or interpreter
Example: "Give a small discount of about 10%"	Example: <code>total = total - (total * 0.10)</code>

KEY POINT

Core Concept

A programming language bridges human logic and machine code. In a high-level language like Python, we write clear steps. A compiler or interpreter then helps the computer run those steps.

TEXT

[Copy](#)

PROGRAMMING LANGUAGE → PRECISE INSTRUCTIONS A COMPUTER CAN FOLLOW

What is Binary & Why Is It Complex for Humans?

What is Binary?

At the physical hardware level, computers are built from millions of tiny electronic switches called **transistors**. An electronic switch can only exist in two states: - **1** = **Switch ON** (electricity is flowing) - **0** = **Switch OFF** (no electricity)

Because computer hardware operates strictly on ON/OFF electrical states, computers use the **Binary System** (a number system with only two digits: **0** and **1**). Every letter, number, image, video, and instruction inside a computer is represented as combinations of **0**s and **1**s.

What Does Binary Look Like vs Human Code?

Look at how the simple word "Hello" is represented across different levels:

Level	What It Looks Like	Who Can Read It Easily?
Human English	Hello	Humans
Python Code	print("Hello")	Humans (Easy to read and write)
Binary Machine Code	01001000 01100101 01101100 01101100 01101111	Computer Processor (Hardware switches)

Notice how a simple 5-letter word requires 40 individual 0s and 1s in binary!

Why is Binary so Complex for Humans?

- Unnatural for Human Brains:** Humans think in words, concepts, and base-10 numbers (0 through 9). Binary forces every single thought into a rigid grid of 0s and 1s.
- Impossible to Read and Spot Mistakes:** If you write 01001000 01100101 and accidentally type a single 0 instead of a 1 (01001000 01100100), human eyes cannot easily catch the mistake. Finding one wrong digit out of thousands of lines of binary is like finding a single broken switch in a giant city grid.
- Hardware Specific:** Machine code written for one type of processor may not work directly on another type. High-level languages like Python reduce this problem by hiding many hardware details. With the right interpreter and libraries, the same Python code can often run on different devices.

TEXT

Copy

BINARY (0s & 1s) → VERY COMPLEX FOR HUMANS → EASY FOR HARDWARE SWITCHES
HIGH-LEVEL CODE → EASY FOR HUMANS TO READ → TRANSLATED TO BINARY FOR HARDWARE

Machine Code and the Translation Gap

Computers only understand **machine code**. Machine code consists of binary digits (0 and 1) that control computer hardware directly. Humans cannot write complex software directly in binary because it is too complex, slow, and error-prone.

TEXT

Copy

```
01100100 01101111 01100101 01110011 ... ← what the machine actually runs
```

Programming languages solve this problem by letting us write readable source code. A **compiler** or **interpreter** then processes that source code so the computer can execute the required instructions.

TEXT

Copy

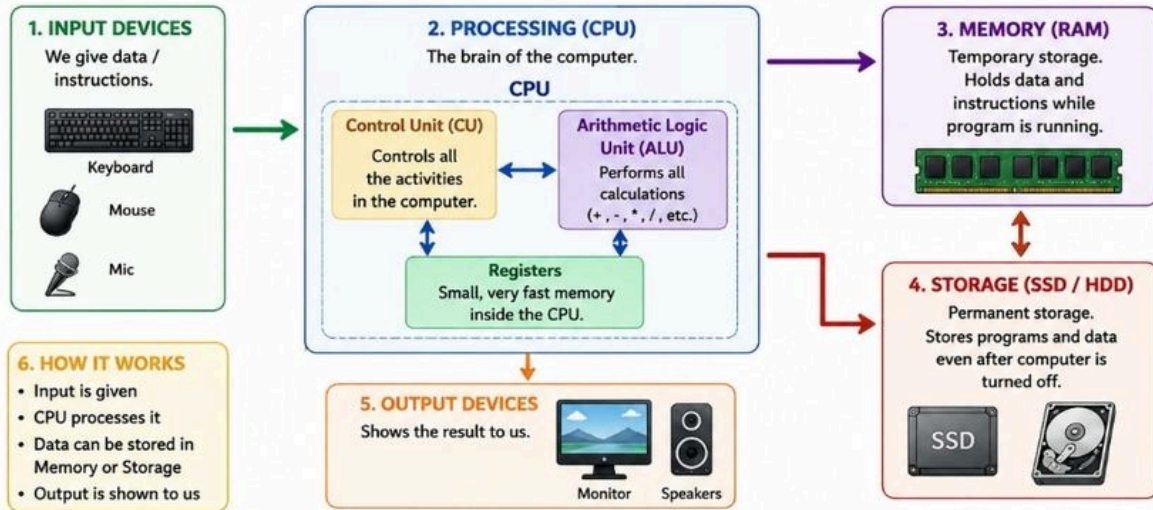
```
HUMAN INTENT
  ↓ (written in a programming language)
SOURCE CODE
  ↓ (translated by compiler/interpreter)
MACHINE CODE
  ↓ (executed by hardware)
RESULT
```

How a Computer Works & How a Program Runs

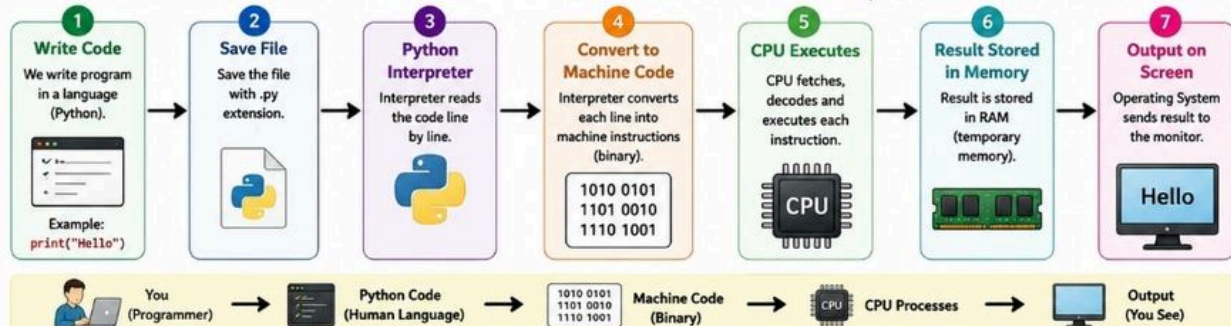
To write code effectively, students must understand two fundamental concepts: **How a Computer Works** (the hardware parts) and **How a Program Runs** (from code to execution).

HOW A COMPUTER WORKS (For Students)

A computer takes input, processes it, stores data and gives output.



FLOW: HOW A PROGRAM RUNS (From Code to Output)



Components of a Computer System

Component	Full Form / Type	Function in Computer System
Input Devices	Keyboard, Mouse, Mic	Receives commands, text, and data from the user
CPU	Central Processing Unit	The "brain" of the computer that processes all instructions
↳ Control Unit (CU)	Internal CPU Unit	Directs and coordinates all hardware activities
↳ ALU	Arithmetic Logic Unit	Performs calculations (+, -, *, /) and logical decisions
↳ Registers	Fast Memory inside CPU	High-speed temporary locations holding current instructions
Memory	RAM (Random Access Memory)	Temporary volatile memory holding data while program is running
Storage	SSD / HDD	Permanent non-volatile storage saving files even when powered off
Output Devices	Monitor, Speakers, Printer	Displays results, audio, or output back to the user

7 Steps: How a Python Program Runs

- 1. Write Code:** The programmer writes Python instructions in an editor (e.g. `print("Hello")`).
- 2. Save File:** The file is saved on permanent storage (SSD/HDD) with a `.py` extension.
- 3. Python Interpreter:** When launched, the interpreter reads the `.py` source file line by line.
- 4. Process the Code:** The Python interpreter checks and processes the code so the computer can run it.
- 5. CPU Executes:** The CPU fetches, decodes, and executes each binary instruction using its CU and ALU.
- 6. Result Stored in Memory:** Calculation results are temporarily held in RAM.

7. **Output on Screen:** The Operating System sends the final result to the monitor screen (Hello).

Without programming languages, every instruction — like calculating a bill or sending an email — would have to be typed manually in binary.

What Happens Without a Programming Language?

Imagine building a food delivery app using only binary 0s and 1s. It would take years, and finding mistakes would be almost impossible.

Without a Programming Language	With a Programming Language
Instructions are only 0s and 1s	Instructions use clear words and keywords
Extremely slow and full of errors	Fast and easy to write
Impossible to read or fix errors	Easy to read, check, and update
Works on only one specific processor	Can often run on different computers with the right compiler, interpreter, or runtime
No reusable code modules	Can reuse code using functions and libraries
Cannot build large software systems	Can build large, complex applications

TEXT

Copy

NO PROGRAMMING LANGUAGE → BUILDING SOFTWARE IS NEARLY IMPOSSIBLE

What Becomes Possible With Programming Languages?

Programming languages allow us to turn ideas into software like mobile apps, websites, games, banking portals, and AI systems. For example, a food delivery company uses programs to:

- Take and verify customer orders
- Calculate item totals, discounts, and taxes
- Track delivery drivers on a map in real time
- Process online payments securely
- Recommend popular restaurants to users
- Send order updates and receipts

TEXT

Copy

IDEA → PROGRAMMING LANGUAGE → WORKING SOFTWARE → REAL-WORLD IMPACT

Why Should We Learn Programming?

Programming is the skill of teaching a computer how to solve problems. Learning to program: - Improves logical thinking and problem-solving skills - Is useful in many fields like data analysis, automation, finance, and engineering - Opens up career opportunities in software development, data science, testing, and cloud operations

TEXT

Copy

LEARN PROGRAMMING → SOLVE PROBLEMS WITH CODE → BUILD REAL SOFTWARE

History & Classification

A Short History of Programming Languages

As computer hardware improved, programming languages evolved to make writing programs easier for humans.

TEXT

[Copy](#)

1940s	MACHINE CODE	→ raw binary (0s and 1s)
1950s	ASSEMBLY LANGUAGE	→ short codes (ADD, MOV, JMP)
1950s-60s	FORTRAN, COBOL	→ first high-level languages, closer to English
1970s	C	→ structured programming, efficient and fast
1980s	C++	→ added Object-Oriented Programming (OOP)
1990s	Java, Python, JS	→ portable, easy to use, huge feature libraries
2000s+	Go, Rust, Swift	→ built for modern web, cloud, and mobile apps

Generations of Programming Languages

Programming languages are categorized into **generations (GL)** based on how close they are to machine hardware versus human language.

Generation	Name	Example	How Close to Human Language
1GL	Machine Language	Binary (0s and 1s)	Closest to hardware
2GL	Assembly Language	ADD, MOV, JMP	Very low level
3GL	High-Level Language	C, Java, Python	Easy to read, close to English
4GL	Very High-Level Language	SQL	Focuses on <i>what</i> data is needed
5GL	Goal-Based Language	Prolog	Focuses on goals or rules rather than step-by-step instructions

TEXT

Copy

1GL → 2GL → 3GL → 4GL → 5GL

CLOSER TO HARDWARE → CLOSER TO HUMAN THOUGHT

Python is a **3rd generation language (3GL)**. It is easy to read while telling the computer step-by-step what to do.

Types of Programming Languages (Paradigms)

A **programming paradigm** is a style or way of writing code.

Type	Core Idea	Example
Procedural	Step-by-step sequence of instructions	Running <code>calculate_total()</code> in order
Object-Oriented (OOP)	Models real things using objects (data + actions)	Creating <code>Order</code> , <code>Customer</code> , and <code>Driver</code>
Functional	Builds programs by combining small, reusable functions	Filtering a list of prices using functions
Scripting	Short scripts written to automate tasks quickly	A script that sends daily sales reports

TEXT

[Copy](#)

```

PROCEDURAL    → SEQUENCE OF STEPS
OBJECT-ORIENTED → REAL-WORLD OBJECTS
FUNCTIONAL    → REUSABLE FUNCTIONS
SCRIPTING     → QUICK AUTOMATION
  
```

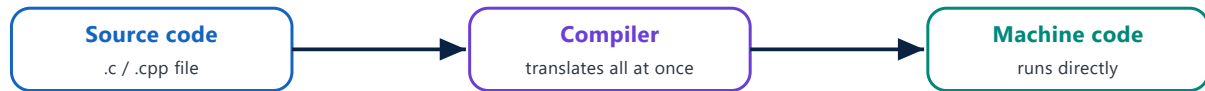
Python supports **all four** styles, making it extremely flexible.

Popular Programming Languages

Language	Primary Use Case
Python	Data science, Artificial Intelligence, Web development, Automation
Java	Large enterprise software, Android apps
C	Operating systems, embedded systems
C++	Game engines, high-speed software
JavaScript	Web development (front-end and back-end)
SQL	Querying and managing database data
Go	Cloud services and server backends
Swift / Kotlin	iOS (Apple) and Android mobile apps

Compiled vs Interpreted Languages

Compiled (e.g. C, C++)



Interpreted (e.g. Python)



Python is commonly called interpreted: the interpreter runs your program step by step.

Compiled languages translate the whole program before running; interpreted languages like Python are run step by step by an interpreter.

Programming languages translate code in one of two main ways:

Compiled Language	Interpreted Language
Translates the entire code before running	Translates and runs code line by line
Generates an <code>.exe</code> or binary file	Needs an interpreter to run
Examples: C, C++	Examples: Python, JavaScript
Syntax errors are found before creating the executable	Errors may appear when the interpreter reaches the problem line
Fast execution speed	Faster to write and test code

TEXT

Copy

COMPILED → SOURCE CODE → FULL TRANSLATION → EXECUTABLE → RUN
 INTERPRETED → SOURCE CODE → INTERPRETER RUNS IT LINE BY LINE

REMEMBER

Exam & Placement Tip

- **Python is commonly called an interpreted language:** The interpreter runs the program step by step. If a runtime error occurs on line 10, earlier lines may run before the program stops.
- **C/C++ are compiled languages:** The compiler checks the source code and creates an executable only if compilation succeeds. Runtime errors can still happen when the executable runs.

Programming Languages vs SQL

SQL (Structured Query Language) is used specifically for databases, whereas Python is a general-purpose programming language.

Programming Language (e.g. Python)	SQL
General-purpose — builds full software systems	Special-purpose — manages database data
Tells the computer <i>how</i> to do a task step by step	Tells the computer <i>what</i> data to get
Has built-in loops, conditions, and functions	Used mainly to search, insert, or update data
Example: Calculate total bill for an order	Example: <code>SELECT * FROM orders WHERE status='Confirmed'</code>

TEXT

Copy

PYTHON → BUILDS APPLICATION LOGIC
SQL → STORES AND GETS DATA FROM THE DATABASE

In a real app, Python runs the application while SQL fetches order details from a database.

Python vs C++ vs Java

PYTHON

Copy

```
print("Order Confirmed")
```

Output:

TEXT

Copy

```
Order Confirmed
```

The same output in C++ or Java requires extra setup lines:

TEXT

Copy

```
C++  → #include <iostream>
      int main() {
          std::cout << "Order Confirmed";
          return 0;
      }

Java  → public class Main {
          public static void main(String[] args) {
              System.out.println("Order Confirmed");
          }
      }
```

Feature	Python	C++	Java
Type System	Dynamic (no data type declared)	Static (must declare data type)	Static (must declare data type)
Execution	Interpreted	Compiled to machine code	Compiled to bytecode (JVM)
Code Style	Simple and clean	Uses braces and symbols	Uses braces and extra classes
Beginner Friendliness	High	Moderate / Low	Moderate
Primary Use	AI, Data Science, Web, Automation	Game engines, high-speed software	Enterprise systems, Android

TEXT

[Copy](#)

PYTHON → EASY TO READ AND WRITE
 C++ → HIGH SPEED AND HARDWARE CONTROL
 JAVA → PORTABLE ACROSS OPERATING SYSTEMS

Advantages and Disadvantages of Python, C++, and Java

Language	Advantages	Disadvantages
Python	Easy to learn, clean code, huge library support	Slower execution speed than C/C++
C++	Very fast execution, full hardware memory control	Harder to learn, long complex code
Java	Portable ("write once, run anywhere"), secure	Requires extra setup code

Why Python

Why Choose Python as Your First Language?

Python removes unnecessary syntax rules so you can focus on learning **logic and problem-solving**.

PYTHON

Copy

```
price = 200
quantity = 2
total = price * quantity
print(total)
```

Output:

TEXT

Copy

400

Python lets you write and run code in just four simple lines.

Advantage	Why It Helps Beginners
Simple English-like code	Focus on logic instead of complex symbols
No rigid variable type rules	Easy to write code without extra declarations
Huge standard library	Built-in tools for math, dates, files, and more
Large community	Easy to find solutions to any problem online
Multi-purpose	Learn one language for Web, Data, AI, and Automation

TEXT

Copy

SIMPLE SYNTAX + HUGE COMMUNITY → EASY LEARNING CURVE

Career Scope of Python

Python is widely used in many areas of software development.

Job Role	What You Do
Data Analyst / Data Scientist	Analyze data and find business insights
Machine Learning / AI Engineer	Build smart predictive software
Web / Backend Developer	Build websites and server APIs
Automation Engineer	Write scripts to automate repetitive tasks
QA / Software Tester	Write automated scripts to test software
DevOps / Cloud Engineer	Automate cloud deployment systems

TEXT

Copy

ONE LANGUAGE → MANY JOB ROLES → HIGH DEMAND

Common Student Mistakes

X Mistake	✓ Correct Understanding
Thinking a programming language is the computer	A language is just a tool to give instructions to a computer
Thinking one language is best for every single task	Different languages are good for different tasks
Ignoring compiled vs interpreted differences	This affects how errors appear and how fast code runs
Thinking SQL can replace Python	SQL manages database data; Python builds application logic
Thinking Python is too slow for real jobs	Python is used by major companies (Google, Netflix, YouTube)
Skipping basic concepts to learn advanced topics directly	Basics make advanced topics much easier to understand

Placement Quick Points

TEXT

[Copy](#)

PROGRAMMING LANGUAGE → BRIDGES HUMAN LOGIC AND MACHINE CODE
 MACHINE CODE → BINARY (1s AND 0s), NOT HUMAN-READABLE
 COMPILER → TRANSLATES ENTIRE CODE BEFORE RUNNING
 INTERPRETER → TRANSLATES AND RUNS CODE LINE BY LINE
 1GL → 5GL → FROM RAW BINARY TO GOAL-BASED LANGUAGES
 PROCEDURAL → STEP-BY-STEP INSTRUCTIONS
 OOP → MODELS REAL-WORLD OBJECTS
 FUNCTIONAL → BUILDS SOFTWARE WITH REUSABLE FUNCTIONS
 SCRIPTING → QUICK TASK AUTOMATION
 PYTHON → INTERPRETED, DYNAMICALLY TYPED, HIGH-LEVEL
 SQL → QUERIES AND MANAGES DATABASE DATA
 PYTHON'S EDGE → EASY TO READ + HUGE LIBRARIES + MANY JOB ROLES

Interview Questions

Q1. What is a programming language?

A set of instructions written by a human that a computer translates and executes.

Q2. How does the human translator analogy explain how compilers and interpreters work?

Just as a human translator translates Tamil to Chinese so two people can communicate, a compiler/interpreter translates human programming code (Python/Java) into machine binary code (0s and 1s) so computer hardware can execute it.

Q3. Why do programming languages exist?

Computers only understand binary code (0s and 1s). Programming languages allow humans to write code in readable words.

Q4. What is machine code?

Machine code is the low-level instruction form that a computer processor can execute directly.

Q5. Why is binary code complex for humans?

Binary requires converting everything into endless 0s and 1s. It is unnatural for human minds, extremely hard to read, almost impossible to debug for small errors, and tied to specific hardware.

Q6. What is the difference between a compiler and an interpreter?

A compiler translates the whole program before running it. An interpreter translates and runs the code line by line.

Q7. Is Python compiled or interpreted?

Python is commonly called an interpreted language. Its interpreter runs the program step by step.

Q8. What are generations of programming languages?

A classification (1GL to 5GL) based on how close a language is to machine code versus human language.

Q9. Which generation does Python belong to?

Python is a 3rd generation language (3GL).

Q10. What is a programming paradigm?

A style or approach to writing and organizing code (e.g., procedural, object-oriented, functional).

Q11. What is the difference between Python and SQL?

Python is a general-purpose programming language used to build software. SQL is used specifically to query and manage database data.

Q12. What is the main difference between Python, C++, and Java?

Python is simple and dynamically typed; C++ is compiled for maximum speed; Java runs on a virtual machine (JVM) for portability across platforms.

Q13. Why is Python beginner-friendly?

It uses simple syntax close to English, requires minimal setup code, and has huge community support.

Practice

1. Explain in simple words what a programming language is.
2. Explain the Language Translator Analogy (Tamil speaker, Chinese speaker, and Human Translator) and how it relates to Python and a Computer.
3. Write down the steps to calculate a food delivery bill in plain English.
4. Why can't a computer understand plain English directly?
5. What is binary code, and why is writing programs directly in binary extremely complex for humans? Give 3 reasons.
6. What is machine code and why don't programmers write code directly in machine code?
7. List three problems if programming languages did not exist.

8. List the 5 generations of programming languages in order (1GL to 5GL).
 9. Which generation does Python belong to?
 10. Give one example of procedural, object-oriented, functional, and scripting programming.
 11. Name 5 popular programming languages and one common use for each.
 12. Explain the difference between a compiler and an interpreter.
 13. Is Python compiled or interpreted? Explain.
 14. How is a programming language different from SQL?
 15. Compare Python, C++, and Java in terms of ease of learning and execution style.
 16. Give two advantages and one disadvantage of Python.
 17. Why is Python a great choice for beginners?
 18. Name three career paths you can choose after learning Python.
-

End of Level 0



Python Basics

Level: 1 | Version: v1.0 | Author: Meptrasoft

Every programmer starts with the basics. This module explains what Python is, how to set it up on your computer, how code is written, and how basic syntax rules work.

PYTHON

Copy

```
print("Welcome to Python Programming!")
```

Output:

TEXT

Copy

Welcome to Python Programming!

Getting Started

Introduction to Python

Python is a high-level, general-purpose programming language designed to be easy to read and write.

"High-level" means Python handles many hardware details automatically, so you can focus on solving your problem.

PYTHON

Copy

```
print("Hello, Student!")
```

Output:

TEXT

Copy

Hello, Student!

In Python, a single `print()` statement displays text on the screen without needing complex setup code. This is why Python is a popular **first language** for students.

TEXT

Copy

IDEA → PYTHON CODE → OUTPUT
(very simple to translate ideas into code)

Compiled vs Interpreted in Python

Language Type	How It Runs	Example
Compiled Language	Code is translated before running	C, C++
Interpreted Language	Code is usually run by an interpreter	Python, JavaScript

Python automatically converts code into internal **bytecode** and then runs it using the Python interpreter.

History of Python

Python was created by **Guido van Rossum** and first released in **1991**.

The name "Python" comes from the famous British comedy show *Monty Python's Flying Circus* — not from the snake!

Version	Year	Key Change
Python 1.0	1994	First stable release
Python 2.0	2000	Added list features and memory cleanup
Python 3.0	2008	Modern syntax cleanup (Python 3 is the current standard)
Python 3.x	Today	Actively maintained and used across industries

TEXT

Copy

1991 → CREATED 2000 → PYTHON 2 2008 → PYTHON 3 (current version)

This entire course uses **Python 3**.

Features of Python

Feature	Meaning
Simple & Readable	Syntax is very close to plain English
Interpreted	Code runs through the Python interpreter
Free & Open Source	Free to download, use, and share
Large Standard Library	Has built-in tools for math, files, dates, and networking
Portable	The same code runs on Windows, Mac, and Linux
Multi-purpose	Used for Web apps, Data Science, AI, and Automation

TEXT

Copy

READABLE + FREE + PORTABLE + HUGE LIBRARY → BEGINNER-FRIENDLY & INDUSTRY-READY

Applications of Python

Field	Real-World Example
Web Development	Backend of websites (using Django or Flask)
Data Science	Analyzing data and building charts (Pandas, NumPy)
Artificial Intelligence	Machine learning models (TensorFlow, PyTorch)
Automation / Scripting	Automating routine file tasks
Software Testing	Automated testing of applications
College Projects	Student management systems, billing apps, quizzes

Installation and Setup

Follow these simple steps to install Python on your computer:

TEXT

[Copy](#)

1. Open your browser and go to `python.org/downloads`
2. Download the Python installer for your operating system (Windows / Mac / Linux)
3. Run the installer – IMPORTANT: Check the box "Add Python to PATH"
4. Open Command Prompt / Terminal and type: `python --version`

Verify your installation:

TEXT

[Copy](#)

```
$ python --version
Python 3.12.3
```

If your terminal says `'python' is not recognized`, Python was not added to PATH during installation. Re-run the installer and check the "Add Python to PATH" box.

Python IDEs (Code Editors)

An **IDE** (Integrated Development Environment) is software that provides a code editor, execution tools, and error checking in one place.

Editor / IDE	Best For
IDLE	Comes built-in with Python; good for complete beginners
VS Code	Free, lightweight, and most widely used in software companies
PyCharm	Feature-packed IDE popular for large Python projects
Thonny	Designed specifically for learning Python in schools/colleges
Jupyter Notebook	Runs code in blocks; popular for Data Science

Online Code Editors

If you don't want to install software immediately, you can practice Python online in your browser:

- **online-python.com** — Fast, simple, no signup required
- **Replit** — Full online coding environment
- **Google Colab** — Free cloud notebook for Python practice

Syntax Essentials

Python Syntax Rules

Syntax means the grammar rules of a programming language. Python syntax is clean: - No semicolons `;` required at the end of lines - No curly braces `{}` required to mark code blocks - Code statements are written **one per line**

PYTHON

Copy

```
print("Welcome to College")  
print("Python Programming Class")
```

Output:

TEXT

Copy

```
Welcome to College  
Python Programming Class
```

TEXT

Copy

```
NO SEMICOLONS | NO CURLY BRACES | SIMPLE & CLEAN WRITING
```

Indentation in Python

Indentation (spaces at the start of a line) is **mandatory** in Python. Python uses indentation to define code blocks.

PYTHON

Copy

```
if 5 > 2:  
    print("Five is greater than two!")  
    print("This line is also inside the if block.")  
print("This line is outside the if block.")
```

Output:

TEXT

Copy

```
Five is greater than two!  
This line is also inside the if block.  
This line is outside the if block.
```

If indentation is inconsistent or missing, Python raises an `IndentationError`:

PYTHON

Copy

```
if 5 > 2:  
print("Five is greater than two!")
```

Output:

TEXT

Copy

```
IndentationError: expected an indented block
```

TEXT

Copy

```
CORRECT INDENTATION → CONSISTENT SPACING (4 SPACES RECOMMENDED)  
INCORRECT SPACING → IndentationError
```

Always use **4 spaces** for each indent level. Do not mix tabs and spaces.

Comments and Notes in Python

A **comment** is a note written in code for human readers. Python completely ignores comments when running the program.

Single-line comments start with `#`:

PYTHON

Copy

```
# This line prints a welcome message  
print("Hello, World!") # Output welcome text
```

For longer notes at the start of a file or function, Python often uses triple-quoted strings called **docstrings**:

PYTHON

Copy

```
"""
Program: College Registration System
Author: Student
Description: This script demonstrates basic Python setup and output.
"""
print("System Ready")
```

Type	Syntax	Usage
Single-Line Comment	# text	Short notes on one line
Docstring / Multi-Line Note	""" text """	Longer documentation text

Conclusion & Next Steps

In this module, you learned: 1. What Python is and why it is so popular 2. How to install Python and set up your editor (VS Code, IDLE, or online tools) 3. Basic syntax rules, mandatory indentation, and comments 4. How to display output using `print()`

Now that your setup is ready and you understand basic syntax, you are ready to learn **Core Python** — where we explore **Variables**, **Keywords**, **Data Types**, **Type Conversion**, **Input/Output**, and **Operators** in detail.

Placement Quick Points

- Python was created by Guido van Rossum in 1991.
- Python 3 is the current actively supported version.
- Python code runs using the Python interpreter.

- Always check "Add Python to PATH" during installation.
- Indentation is mandatory in Python and defines code blocks.
- 4 spaces per indentation level is the standard convention.
- `#` is used for single-line comments; Python ignores comments.
- `print()` is the built-in function used to display output.

Interview Questions

Q1. What is Python and who created it?

Python is a high-level, interpreted programming language created by Guido van Rossum in 1991.

Q2. Is Python compiled or interpreted?

Python is interpreted. It first converts code to internal bytecode and then runs it using the Python interpreter.

Q3. Why is indentation mandatory in Python?

Indentation defines code blocks instead of using curly braces `{}`.

Q4. What happens if you use wrong indentation in Python?

Python raises an `IndentationError`.

Q5. How do you write comments in Python?

Use `#` for comments. Triple quotes `"""` are often used for docstrings or longer notes.

Q6. What tool is used to display output in Python?

The built-in `print()` function.

Practice

1. Write a Python statement to print your name and college name.
2. Install Python on your computer and verify it using `python --version`.

3. Create a `.py` file and run it from VS Code or Command Prompt.
4. Add a single-line comment above a `print()` statement explaining what it does.
5. Create a docstring or multi-line note describing your first Python program.
6. Write a code snippet with correct indentation and observe the output.
7. Fix the error in this code:

PYTHON

Copy

```
if 10 > 5:  
    print("10 is greater than 5")
```

End of Level 1 — Foundations

Python Variables

Level: 1 | Version: v1.0 | Author: Meptrasoft

What is a Variable?

A **variable** is a name that refers to a value in a program. For beginners, you can think of it like a labeled box that helps you find and use data.

PYTHON

Copy

```
student_name = "Arun"  
age = 21  
print(age)
```

In this example: - `student_name` holds the text value `"Arun"` - `age` holds the number value `21`

Output:

TEXT

Copy

```
21
```

Why Do We Use Variables?

Variables allow us to **store, reuse, and update data** throughout our program.

PYTHON

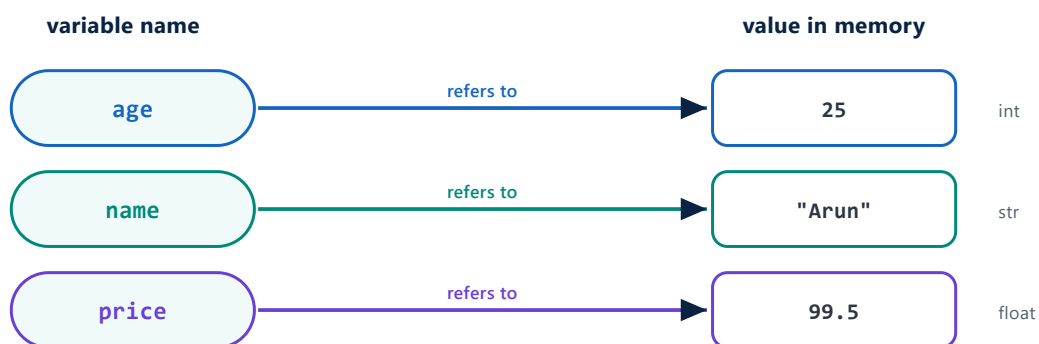
Copy

```
price = 100
quantity = 5
total = price * quantity

print(total) # Output: 500
```

Without variables, we would have to type raw numbers repeatedly, making code hard to read and modify.

Creating a Variable



Assignment like `age = 25` binds the name on the left to the value on the right.

A variable is a name that refers to a value stored in memory. Assignment binds the name to the value.

In Python, a variable is created automatically when you assign a value using the `=` (assignment) operator.

PYTHON

Copy

```
score = 90
```

Python detects data types automatically:

PYTHON

Copy

```
age = 21          # Integer (whole number)
price = 99.5      # Float (decimal number)
name = "Arun"     # String (text)
is_admitted = True # Boolean (True/False)
```

Variable Naming Rules

Python enforces strict **rules** for variable names. If you break these rules, Python gives a syntax error.

Rule	✗ Invalid Name	✓ Valid Name
Must start with a letter or underscore (<code>_</code>)	<code>1student</code>	<code>student1</code> , <code>_student</code>
Can contain letters, numbers, and underscores	<code>student@name</code>	<code>student_name</code>
Cannot contain spaces	<code>student name</code>	<code>student_name</code>
Cannot use special characters (<code>!</code> , <code>@</code> , <code>#</code> , <code>-</code>)	<code>student-name</code>	<code>student_name</code>
Cannot use Python keywords (<code>class</code> , <code>if</code> , <code>for</code>)	<code>class</code>	<code>class_name</code>

Python variable names are **case-sensitive**. This means `age`, `Age`, and `AGE` are three different names.

Case-Sensitivity Example

PYTHON

Copy

```
age = 21
Age = 25
AGE = 30

print(age) # 21
print(Age) # 25
print(AGE) # 30
```

`age`, `Age`, and `AGE` are three completely different variables in Python.

Dynamic Typing

In Python, you do not need to declare a variable's data type. A variable name can refer to a number first, and later refer to a string. This is called **dynamic typing**.

PYTHON

Copy

```
data = 10
print(data) # 10

data = "Python Programming"
print(data) # Python Programming
```

Output:

TEXT

Copy

```
10
Python Programming
```

Swapping Two Variables

In Python, swapping the values of two variables is very simple:

PYTHON

Copy

```
a = 10
b = 20

# Swapping in one simple line
a, b = b, a

print("a =", a) # a = 20
print("b =", b) # b = 10
```

Variable Naming Styles

Style Name	Example	Commonly Used For
snake_case	student_name	Recommended for Python variables and functions
camelCase	studentName	Used in languages like JavaScript and Java
PascalCase	StudentName	Used for class names in Python
UPPER_CASE	MAX_LIMIT	Used for constant values

Always use descriptive **snake_case** names in Python:

PYTHON

Copy

```
# Good, clear variable name
total_marks = 450

# Bad, unclear variable name
x = 450
```

User Input and Output with Variables

Displaying Output with `print()` and f-strings

The `print()` function displays variables on screen. **f-strings** (formatted string literals) allow you to insert variable values directly inside text:

PYTHON

Copy

```
name = "Arun"
age = 21

# Using f-string for clear output
print(f"Student Name: {name}, Age: {age}")
```

Output:

TEXT

Copy

```
Student Name: Arun, Age: 21
```

Reading Input with `input()`

The `input()` function reads input entered by the user.

Important: `input()` always returns data as a string (`str`), even if the user types numbers.

PYTHON

Copy

```
user_name = input("Enter your name: ")  
print(f"Hello, {user_name}!")
```

If you want to use the input as a number, convert it:

PYTHON

Copy

```
age = int(input("Enter your age: "))  
print(f"Age: {age}")
```

Placement Quick Points

- A variable is a name that refers to a value.
- `=` is the assignment operator used to store values.
- Python uses **dynamic typing** (no explicit type declaration needed).
- Variable names are **case-sensitive** (`age` is different from `Age`).
- Use **snake_case** for naming Python variables (`student_name`).
- Swap variables easily using `a, b = b, a`.
- `input()` function always returns user input as a string (`str`).
- Use f-strings (`f"Name: {name}"`) to print formatted output easily.

Interview Questions

Q1. What is a variable in Python?

A name that refers to a value in a program.

Q2. Is Python dynamically typed? Explain.

Yes. A variable's data type is decided at runtime based on the value assigned to it, and can change over time.

Q3. What is the difference between `=` and `==`?

`=` is used to assign a value to a variable; `==` is used to compare whether two values are equal.

Q4. Are Python variable names case-sensitive?

Yes. `score`, `Score`, and `SCORE` are three different variables.

Q5. What naming convention is recommended for Python variables?

`snake_case` (all lowercase letters separated by underscores).

Q6. How do you swap two variables in Python without using a temporary variable?

By using tuple assignment: `a, b = b, a`.

Q7. What data type does `input()` return?

`input()` always returns data as a string (`str`).

Practice

1. Create variables to store your name, age, and college score. Print them using an f-string.

2. Check which of these variable names are valid: `1st_name`, `_score`, `total-marks`, `user_age`, `class`.
 3. Convert `studentRegistrationNumber` into standard Python `snake_case`.
 4. Create two variables `x = 5` and `y = 15`. Swap their values and print the result.
 5. What happens if you run `2name = "Arun"`? Fix the error.
 6. Take a user's city name using `input()` and display a welcome message.
-

End of Python Variables

Python Keywords

Level: 1 | Version: v1.0 | Author: Meptrasoft

What is a Keyword?

A **keyword** is a reserved word that has a fixed, special meaning in Python.

Because keywords are reserved by Python for building code structure, you **cannot use keywords as variable names, function names, or class names.**

PYTHON

Copy

```
# Incorrect: Using 'if' (a keyword) as a variable name raises a SyntaxError
if = 10          # Wrong

# Correct: Use a normal name
if_value = 10   # Correct
```

Output:

TEXT

Copy

KEYWORD → RESERVED WORD IN PYTHON → CANNOT BE USED AS A VARIABLE NAME

How Many Keywords Are There in Python?

In current Python versions, there are **35 reserved keywords.**

You can check the full list of keywords directly in Python using the built-in `keyword` module:

PYTHON

Copy

```
import keyword

# Print all keywords in your Python version
print(keyword.kwlist)

# Print total count of keywords
print(len(keyword.kwlist)) # Output: 35
```

You can also check whether any word is a keyword using `keyword.iskeyword()`:

PYTHON

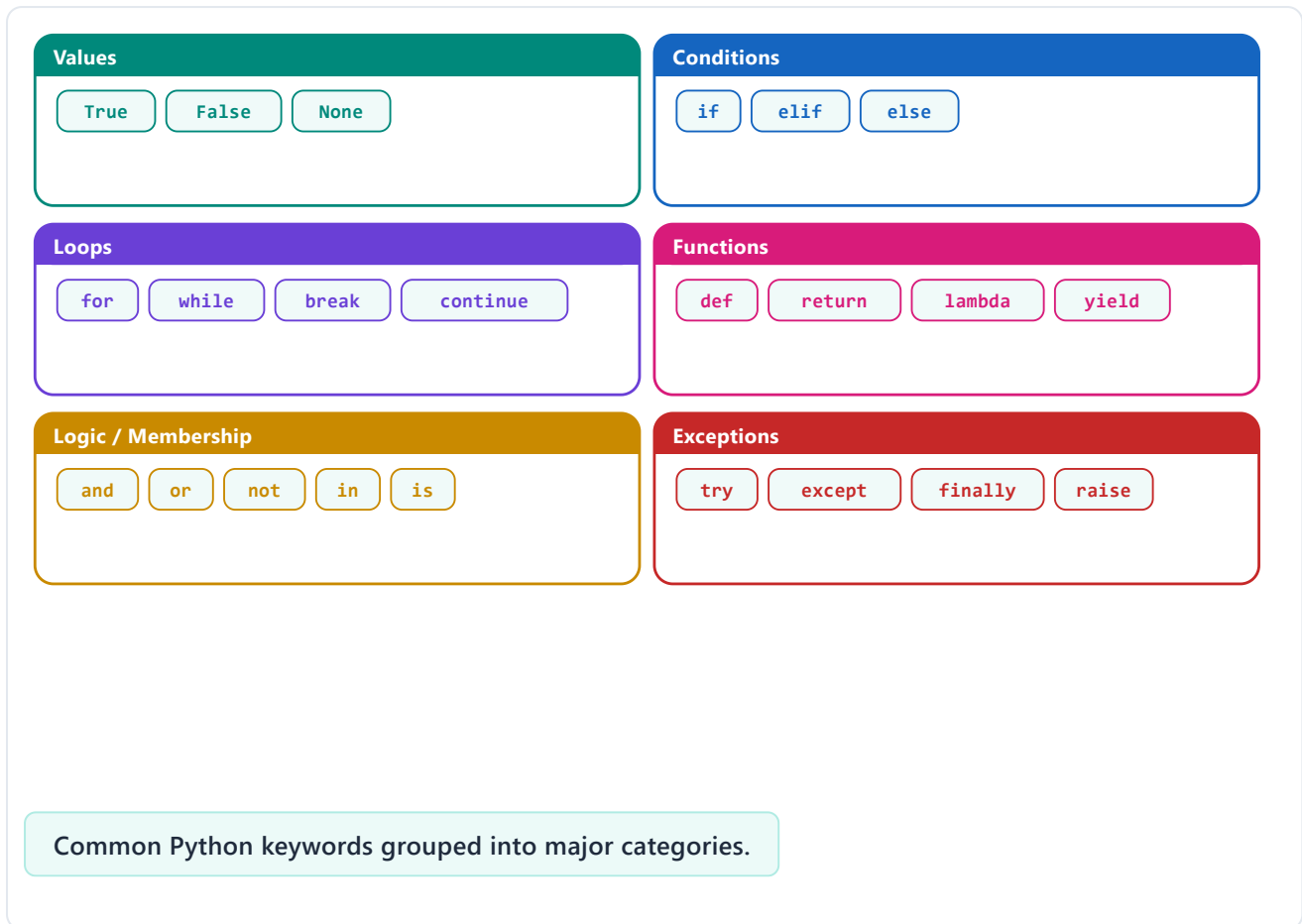
Copy

```
import keyword

print(keyword.iskeyword("for")) # True (for is a keyword)
print(keyword.iskeyword("total")) # False (total is a normal variable name)
```

All 35 Keywords Grouped by Category

The diagram below shows the main keyword groups. The table after it gives a complete beginner-friendly list of all 35 reserved keywords.



Here is a complete breakdown of the 35 reserved keywords by category:

Category	Keywords	What They Do
Values	<code>True</code> , <code>False</code> , <code>None</code>	Represent Boolean truth values and the absence of a value
Conditions	<code>if</code> , <code>elif</code> , <code>else</code>	Used for making decisions in code
Loops	<code>for</code> , <code>while</code> , <code>break</code> , <code>continue</code> , <code>pass</code>	Used to repeat actions or control loop flow
Functions & Classes	<code>def</code> , <code>return</code> , <code>lambda</code> , <code>yield</code> , <code>class</code>	Used to create functions and classes
Logic & Comparison	<code>and</code> , <code>or</code> , <code>not</code> , <code>in</code> , <code>is</code>	Used for logical tests and membership checks
Imports & Scope	<code>import</code> , <code>from</code> , <code>as</code> , <code>global</code> , <code>nonlocal</code>	Used to load modules and manage variable scope
Error Handling	<code>try</code> , <code>except</code> , <code>finally</code> , <code>raise</code> , <code>assert</code>	Used to handle runtime errors gracefully
Other Control Keywords	<code>with</code> , <code>del</code> , <code>async</code> , <code>await</code>	Used for context managers, deleting names, and asynchronous programming

Note: Only three keywords start with a capital letter in Python: `True`, `False`, and `None`. All other keywords are written in lowercase.

Placement Quick Points

- Python has 35 reserved keywords.

- Keywords cannot be used as variable names.
- Attempting to use a keyword as a variable name raises a `SyntaxError`.
- Only `True`, `False`, and `None` start with a capital letter.
- Use `keyword.kwlist` to see the full list of keywords.
- Use `keyword.iskeyword("word")` to check if a word is a keyword.

Interview Questions

Q1. What is a keyword in Python?

A reserved word that has a fixed, predefined meaning in Python syntax.

Q2. Can you use `class` as a variable name in Python?

No. `class` is a reserved keyword; using it as a variable name raises a `SyntaxError`.

Q3. Which Python keywords start with a capital letter?

Only three: `True`, `False`, and `None`.

Q4. How can you check if a word is a Python keyword using code?

Import the `keyword` module and use `keyword.iskeyword("word")`.

Q5. What error occurs if you use a keyword as a variable name?

`SyntaxError: invalid syntax`.

Practice

1. Write a program using `import keyword` to print all 35 Python keywords.

2. Check if `eval`, `exec`, `async`, and `await` are Python keywords using `keyword.iskeyword()`.
3. Fix the syntax error in this code:

PYTHON

Copy

```
for = "Fourth Year Student"
```

1. Name three keywords used for conditional statements.
2. Name three keywords used for handling exceptions/errors.

End of Python Keywords

Python Data Types

Level: 1 | Version: v1.0 | Author: Meptrasoft

What is a Data Type?

A **data type** tells Python what kind of value a name refers to and what operations can be performed on that value.

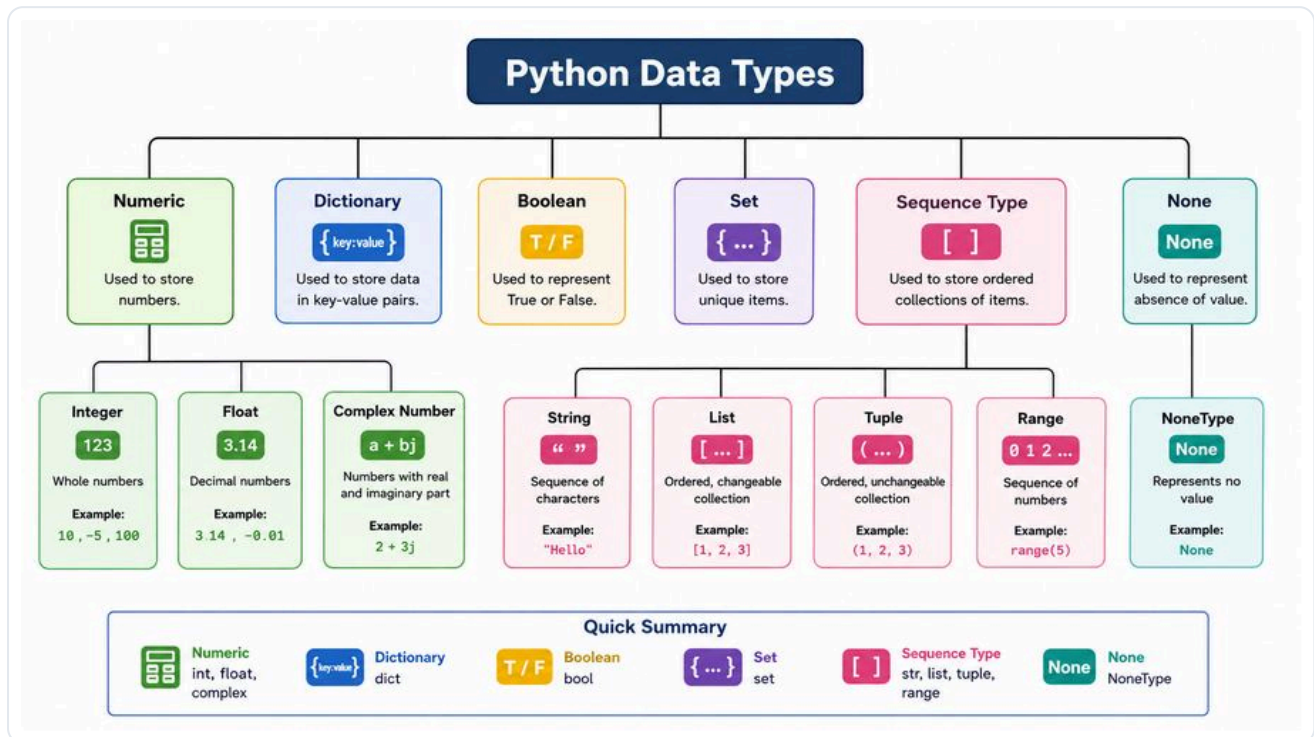
PYTHON

Copy

```
age = 21          # int (integer)
name = "Arun"     # str (string)
price = 99.5      # float (decimal number)
```

In Python, every value has a data type. Python detects the data type automatically when you assign a value to a name.

Data Types at a Glance



Why Do We Need Data Types?

Data types tell Python how to handle data. For example, the `+` operator behaves differently depending on data types:

PYTHON

Copy

```
# Adding two numbers
a = 10
b = 20
print(a + b) # Output: 30
```

Overview of Python Data Types

Category	Data Type	Example	Description
Numeric	<code>int</code>	<code>10</code> , <code>-5</code>	Whole numbers
Numeric	<code>float</code>	<code>10.5</code> , <code>99.9</code>	Numbers with decimal points
Numeric	<code>complex</code>	<code>2 + 3j</code>	Complex numbers with real and imaginary parts
Text	<code>str</code>	<code>"Python"</code>	Text enclosed in single or double quotes
Boolean	<code>bool</code>	<code>True</code> , <code>False</code>	Logical truth values
Sequence	<code>list</code>	<code>[1, 2, 3]</code>	Ordered, changeable list of values
Sequence	<code>tuple</code>	<code>(1, 2, 3)</code>	Ordered, unchangeable tuple of values
Sequence	<code>range</code>	<code>range(5)</code>	Generated sequence of numbers
Set	<code>set</code>	<code>{1, 2, 3}</code>	Unordered collection of unique values
Mapping	<code>dict</code>	<code>{"name": "Arun"}</code>	Key-value pairs
Special value	<code>NoneType</code>	<code>None</code>	Represents the absence of a value

Detailed Data Types

1. Numeric Types

PYTHON

Copy

```
age = 21          # int
price = 45000.50  # float
c_num = 3 + 4j    # complex
```

2. Text Type (`str`)

Strings store text data enclosed in single (`'...'`) or double (`"..."`) quotes:

PYTHON

Copy

```
course = "Python Programming"
```

3. Boolean Type (`bool`)

Booleans store either `True` or `False` (always capitalized):

PYTHON

Copy

```
is_enrolled = True
is_passed = False
```

4. Collection Types Summary

Type	Example	Ordered?	Changeable (Mutable)?	Duplicates Allowed?
list	<code>["Python", "Java", "SQL"]</code>	Yes	Yes	Yes
tuple	<code>("Python", "Java")</code>	Yes	No	Yes
set	<code>{1, 2, 3}</code>	No	Yes	No
dict	<code>{"name": "Arun", "age": 21}</code>	Yes	Yes	Keys: No

Type Conversion (Type Casting)

Type conversion means changing a value from one data type to another. In Python, type conversion happens in two ways: **Implicit Conversion** and **Explicit Conversion**.

1. Implicit Conversion (Automatic)

Python can automatically convert compatible numeric types during calculations.

PYTHON

Copy

```
num_int = 10      # int
num_float = 2.5   # float

total = num_int + num_float

print(total)      # Output: 12.5
print(type(total)) # Output: <class 'float'>
```

Python treated `10` (`int`) as `10.0` (`float`) before adding, so the result became a `float`.

2. Explicit Conversion (Manual)

Explicit conversion is done by the programmer using built-in functions like `int()`, `float()`, and `str()`.

Converting String to Integer / Float

PYTHON

Copy

```
age_text = "21"
age_num = int(age_text)

print(age_num + 1) # Output: 22
```

Converting Number to String

PYTHON

Copy

```
marks = 95
marks_text = str(marks)

print("Your marks are: " + marks_text)
```

Warning: Converting non-numeric text to an integer causes a `ValueError`:

PYTHON

Copy

```
val = int("twenty") # Raises ValueError: invalid literal for int()
```

Checking Data Types with `type()`

Use the built-in `type()` function to check the data type of a value:

PYTHON

Copy

```
x = 10
y = "Hello"

print(type(x)) # <class 'int'>
print(type(y)) # <class 'str'>
```

Placement Quick Points

- Use `type()` to check the data type of a value.
- `int`, `float`, and `complex` are numeric types.
- `str` stores text inside quotes.
- `bool` stores `True` or `False`.
- Lists (`list`) are changeable; Tuples (`tuple`) are unchangeable.
- Sets (`set`) store unique values and remove duplicates automatically.
- Dictionaries (`dict`) store data as `key: value` pairs.
- Implicit type conversion is done automatically by Python when compatible values are used together.
- Explicit type conversion uses `int()`, `float()`, and `str()`.
- Converting invalid text like `int("hello")` raises a `ValueError`.

Interview Questions

Q1. What is a data type in Python?

A classification that tells Python what kind of value a name refers to and what operations can be performed on that value.

Q2. How do you check a value's data type?

By using the built-in `type()` function (e.g., `type(value)`).

Q3. What is the difference between implicit and explicit type conversion?

Implicit conversion is done automatically by Python when compatible types are used together. Explicit conversion is done manually by the programmer using functions like `int()`, `float()`, or `str()`.

Q4. What is the difference between a list and a tuple?

A `list` is mutable (can be changed after creation), while a `tuple` is immutable (cannot be changed).

Q5. What happens if you try `int("abc")` in Python?

It raises a `ValueError` because `"abc"` is not a valid numeric string.

Q6. What output does `"5" + "5"` give in Python?

`"55"` because both operands are strings, so string concatenation takes place.

Practice

1. Create variables for `int`, `float`, `str`, and `bool`, and print their data types using `type()`.
2. Convert the string `"100"` into an integer and add `50` to it.
3. Convert the integer `450` to a string and combine it with `"Total Marks: "`.
4. Create a list of 3 colors and change the first color to `"Blue"`.
5. Create a set `{10, 20, 20, 30}` and print it. Explain why `20` appears only once.
6. What error occurs when running `int("nineteen")`? Fix it.

End of Python Data Types

Operators

Level: 2 | Version: v1.0 | Author: Meptrasoft

Operators let us calculate, compare, and combine values. This module uses a **Student Placement Eligibility System** as the running example.

What is an Operator?

An **operator** is a symbol or keyword used to perform an operation on values. Variable names can be used in expressions because they refer to values.

PYTHON

Copy

```
cgpa = 8.5
bonus = 0.5
final_cgpa = cgpa + bonus
print(final_cgpa)
```

Output:

TEXT

Copy

```
9.0
```

Here, **+** is an **operator**. It performs addition.

What are Operands?

The values on which an operator performs an operation are called **operands**.

PYTHON

Copy

```
cgpa = 8.5  
bonus = 0.5  
print(cgpa + bonus)
```

Here:

- `cgpa` → Operand
- `+` → Operator
- `bonus` → Operand

Output:

TEXT

Copy

```
9.0
```

Why Do We Use Operators?

Operators are used to:

- Perform calculations
- Compare values
- Check conditions
- Combine multiple conditions
- Assign values
- Check membership
- Compare objects

Example:

PYTHON

Copy

```
cgpa = 8.5  
print(cgpa >= 7.0)
```

Output:

TEXT

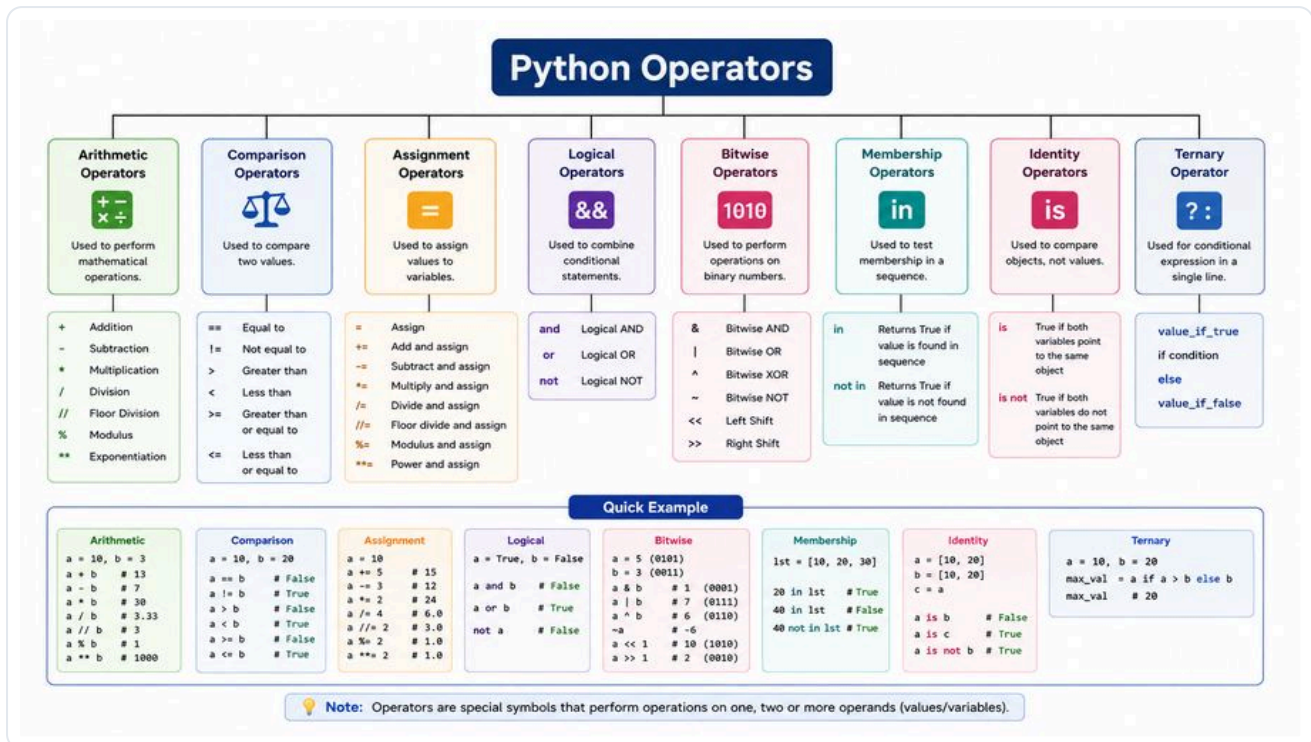
Copy

True

This can be used to check whether a student meets a company's CGPA requirement.

Types of Operators in Python

Operator Type	Purpose
Arithmetic	Perform mathematical calculations
Comparison	Compare two values
Assignment	Assign or update values
Logical	Combine conditions
Membership	Check whether a value exists
Identity	Compare object identity
Bitwise	Perform operations on binary values



We will use this data throughout the topic.

PYTHON

Copy

```
cgpa = 8.5
age = 21
backlogs = 0
skills = ["Python", "SQL", "Java"]
is_placed = False
```

Arithmetic Operators

Arithmetic operators perform mathematical calculations.

Operator	Name	Example
+	Addition	10 + 5
-	Subtraction	10 - 5
*	Multiplication	10 * 5
/	Division	10 / 5
//	Floor Division	10 // 3
%	Modulus	10 % 3
**	Exponentiation	2 ** 3

Addition +

Adds two values.

PYTHON

Copy

```
cgpa = 8.0
bonus = 0.5
final_cgpa = cgpa + bonus
print(final_cgpa)
```

Output:

TEXT

Copy

```
8.5
```

Subtraction -

Subtracts one value from another.

PYTHON

Copy

```
required_cgpa = 7.0
student_cgpa = 8.5
difference = student_cgpa - required_cgpa
print(difference)
```

Output:

TEXT

Copy

1.5

Multiplication *

Multiplies values.

PYTHON

Copy

```
monthly_stipend = 15000
months = 6
total_stipend = monthly_stipend * months
print(total_stipend)
```

Output:

TEXT

Copy

90000

Division /

Divides one value by another.

PYTHON

Copy

```
total_marks = 450
subjects = 5
total_marks = 450
total_marks = 450
average = total_marks / subjects
print(average)
```

Output:

TEXT

Copy

90.0

Important

The `/` operator returns a `float`.

PYTHON

Copy

```
print(10 / 2)
```

Output:

TEXT

Copy

5.0

Floor Division `//`

Returns the floor of the division result.

PYTHON

Copy

```
questions = 10
students = 3
questions_per_student = questions // students
print(questions_per_student)
```


TEXT

Copy

3

TEXT

Copy

```
10 / 3 → 3.333...  
10 // 3 → 3
```

Important

Floor division rounds the result **down toward negative infinity**.

PYTHON

Copy

```
print(-10 // 3)
```

Output:

TEXT

Copy

-4

It does not simply remove the decimal part.

Modulus (%)

Returns the remainder.

PYTHON

Copy

```
students = 10  
team_size = 3  
remaining_students = students % team_size  
print(remaining_students)
```

Output:

TEXT

Copy

1

Common Use — Even or Odd

PYTHON

Copy

```
student_id = 102
print(student_id % 2)
```

Output:

TEXT

Copy

0

If:

TEXT

Copy

```
number = 10
number = 10
number % 2 == 0 → Even
number % 2 != 0 → Odd
```

Exponentiation ******

Raises a value to a power.

PYTHON

Copy

```
print(2 ** 3)
```

Output:

TEXT

Copy

8

TEXT

Copy

 $2^3 = 2 \times 2 \times 2 = 8$

Arithmetic Operator Quick Comparison

Expression	Result
<code>10 + 3</code>	<code>13</code>
<code>10 - 3</code>	<code>7</code>
<code>10 * 3</code>	<code>30</code>
<code>10 / 3</code>	<code>3.333...</code>
<code>10 // 3</code>	<code>3</code>
<code>10 % 3</code>	<code>1</code>
<code>10 ** 3</code>	<code>1000</code>

⚠ IMPORTANT

Important: `/` vs `//` vs `%`

These three operators are based on **division**, but each answers a different question.

Let's use one real-world example:

PYTHON

Copy

```
students = 10
team_size = 3
```

We have **10 students**. Each team can contain **3 students**.

/ — Exact Division Result

Use `/` when you need the **actual division result**.

PYTHON

Copy

```
students = 10
team_size = 3
result = students / team_size
print(result)
```

Output:

TEXT

Copy

```
3.3333333333333335
```

What Does It Mean?

Mathematically:

TEXT

Copy

```
10 ÷ 3 = 3.333...
```

The `/` operator gives the **division result**.

When Do We Use `/`?

Use `/` when decimal values are meaningful.

Real-time examples:

- Average marks
- Percentage calculations
- Price calculations
- Speed
- Distance
- Salary averages

Real-Time Example — Average Marks

PYTHON

Copy

```
total_marks = 450
subjects = 5
average = total_marks / subjects
print(average)
```

Output:

TEXT

Copy

```
90.0
```

Easy Rule

TEXT

Copy

```
Need the division result?
↓
Use /
```

// — Complete Groups or Whole Units

Use `//` when you need the **floor of the division result**.

For positive counts, this often answers:

How many complete groups can be formed?

PYTHON

Copy

```
students = 10
team_size = 3
complete_teams = students // team_size
print(complete_teams)
```

Output:

TEXT

Copy

3

Why?

TEXT

Copy

```
Team 1 → 3 students
Team 2 → 3 students
Team 3 → 3 students
Used students → 9
Remaining    → 1
```

We can form only 3 complete teams of size 3.

When Do We Use `//`?

Common coding examples:

- Complete groups
- Full batches
- Full pages
- Complete boxes
- Time-unit conversion
- Row and column calculations
- Chunk calculations

Real-Time Example — Pagination

PYTHON

Copy

```
total_records = 95
records_per_page = 10
total_records = 95
total_records = 95
full_pages = total_records // records_per_page
print(full_pages)
```

Output:

TEXT

Copy

9

There are **9 completely filled pages**. One more partially filled page may still be required.

Real-Time Example — Convert Seconds to Minutes

PYTHON

Copy

```
print(total_seconds = 130)
print(total_seconds)
total_seconds = 130
total_seconds = 130
minutes = total_seconds // 60
print(minutes)
```

Output:

TEXT

Copy

```
130
2
```

There are **2 complete minutes** in 130 seconds.

Important Technical Point

`//` means **floor division**. It rounds the division result **down toward negative infinity**.

PYTHON

Copy

```
print(10 // 3)    # 3
print(-10 // 3)   # -4
```

So do not teach:

Output:

TEXT

Copy

```
3
-4
```

That explanation is incorrect for negative values. Teach:

TEXT

Copy

```
// gives the floor of the division result
```

Easy Rule

TEXT

Copy

```
Need complete groups or whole units?
      ↓
    Use //
```

% — What is Left?

Use % when you need the remainder.

PYTHON

Copy

```
students = 10
team_size = 3
remaining_students = students % team_size
print(remaining_students)
```

Output:

TEXT

Copy

```
1
```

Why?

TEXT

Copy

```
10 students
↓
3 complete teams × 3 students = 9 students
↓
1 student remaining
```

The `%` operator tells us what is left after division.

When Do We Use `%`?

Common coding examples:

- Even or odd checking
- Remaining items
- Time conversion
- Cyclic rotation
- Alternating logic
- Index wrapping
- Divisibility checking

Real-Time Example — Even or Odd

PYTHON

Copy

```
number = 10
print(number % 2)
```

Output:

TEXT

Copy

```
0
```

If a number is exactly divisible by `2`, the remainder is `0`.

PYTHON

Copy

```
number % 2 == 0
```

Means:

Output:

TEXT

Copy

The number is even.

Example:

PYTHON

Copy

```
number = 10
if number % 2 == 0:
    print("Even")
else:
    print("Odd")
```

Real-Time Example — Remaining Seconds

PYTHON

Copy

```
total_seconds = 130
remaining_total_seconds = 130
total_seconds = 130
seconds = total_seconds % 60
print(remaining_seconds)
```

Output:

TEXT

Copy

10

Why?

TEXT

Copy

```
130 seconds
↓
2 complete minutes = 120 seconds
↓
10 seconds remaining
```

Easy Rule

TEXT

Copy

Need to know what is left?
↓
Use %

One Example — `/` vs `//` vs `%`

Use:

PYTHON

Copy

```
students = 10  
team_size = 3
```

Now compare:

PYTHON

Copy

```
team_size = 3  
students = 10  
print(students / team_size)  
print(students // team_size)  
print(students % team_size)
```

Output:

TEXT

Copy

```
3.3333333333333335  
3  
1
```

Operator	Question It Answers	Result
/	What is the division result?	3.333...
//	What is the floor of the division result?	3
%	What is the remainder?	1

Real-time meaning:

TEXT

Copy

```
10 students ÷ 3 students per team
/ → Mathematical division result = 3.333...
// → Complete teams = 3
% → Students left = 1
```

The Most Important Coding Connection

// and % are often used together.

Consider:

PYTHON

Copy

```
print(total_seconds = 130)
```

We want:

Output:

TEXT

Copy

```
2 minutes 10 seconds
```

Code:

PYTHON

Copy

```
total_seconds = 130
minutes = total_seconds // 60
seconds = total_seconds % 60
print(minutes, "minutes")
print(seconds, "seconds")
```

Output:

TEXT

Copy

```
2 minutes
10 seconds
```

How it works:

TEXT

Copy

```
130 // 60 → 2 complete minutes
130 % 60 → 10 remaining seconds
```

This pattern appears frequently in coding.

TEXT

Copy

```
// → Complete units
% → Remaining units
```

Real-Time Example — Pagination

An application has:

PYTHON

Copy

```
total_records = 95
records_per_page = 10
```

Completely filled pages:

PYTHON

Copy

```
full_pages = total_records // records_per_page
print(full_pages)
```

Output:

TEXT

Copy

9

Records on the final partial page:

PYTHON

Copy

```
records_per_page = 10
total_records = 95
remaining_records = total_records % records_per_page
print(remaining_records)
```

Output:

TEXT

Copy

5

So:

TEXT

Copy

```
9 full pages
5 records remaining
```

If the application must display **all records**, it needs **10 total pages**. A common formula is:

PYTHON

Copy

```
records_per_page = 10
total_records = 95
total_pages = (total_records + records_per_page - 1) // records_per_page
print(total_pages)
```

Output:

TEXT

Copy

```
10
```

For beginners, first understand:

TEXT

Copy

```
// → Complete pages  
% → Remaining records
```

Then learn the total-page formula as a coding pattern.

Real-Time Example — Hours and Minutes

PYTHON

Copy

```
total_minutes = 135  
hours = total_minutes // 60  
minutes = total_minutes % 60  
print(hours, "hours")  
print(minutes, "minutes")
```

Output:

TEXT

Copy

```
2 hours  
15 minutes
```

How?

TEXT

Copy

```
135 // 60 → 2 complete hours  
135 % 60 → 15 remaining minutes
```

How to Choose the Correct Operator

Ask yourself one question.

Do I need the division result?

TEXT

Copy

Yes → /

PYTHON

Copy

```
average = total_marks / subjects
```

Do I need the floor of the result, often representing complete units?

Output:

TEXT

Copy

Yes → //

PYTHON

Copy

```
minutes = total_seconds // 60
```

Do I need the remainder or leftover?

Output:

TEXT

Copy

Yes → %

PYTHON

Copy

```
seconds = total_seconds % 60
```


Permanent Memory Rule

Output:

TEXT

Copy

```
/ → DIVISION RESULT  
// → FLOOR / COMPLETE UNITS  
% → REMAINDER / LEFTOVER
```

Remember the team example:

TEXT

Copy

```
10 Students  
3 Students Per Team  
10 / 3 → 3.333... → Division result  
10 // 3 → 3 → Complete teams  
10 % 3 → 1 → Student left
```

If you remember this one example, you can remember all three operators.

Common Fresher Mistakes — / // %

X Wrong Understanding	✓ Correct Understanding
/ always gives an integer	/ performs division and returns a float
// removes decimals	// performs floor division
% means percentage	% is the modulus operator and returns the remainder
10 // 3 means rounded division	It gives the floor of the division result
// and % are unrelated	They are often used together for complete units and leftovers
95 // 10 gives total pages needed	It gives full pages; a partial page may also be required

Placement Quick Points — / // %

- / performs division and returns a floating-point result.
- // performs floor division and rounds down toward negative infinity.
- % returns the remainder.
- Use / for averages and division results.
- // is often useful for complete groups or units.
- % is useful for leftovers and divisibility checks.
- // and % are commonly used together.
- number % 2 == 0 checks whether a number is even.
- Time conversion commonly uses // and %.
- Pagination logic often uses division-related operators.

Interview Questions — `/` `//` `%`

Q1. What is the difference between `/` and `//`?

`/` performs division. `//` performs floor division.

Q2. What does `%` return?

The remainder after division.

Q3. What is the output of `10 / 3`?

Approximately `3.333`.

Q4. What is the output of `10 // 3`?

`3`.

Q5. What is the output of `10 % 3`?

`1`.

Q6. Why is `-10 // 3` equal to `-4`?

Because floor division rounds down toward negative infinity.

Q7. How do you check whether a number is even?

`number % 2 == 0`.

Q8. How do you convert seconds into minutes and remaining seconds?

`minutes = total_seconds // 60` and `seconds = total_seconds % 60`.

Q9. Which operator is used for average calculation?

`/`.

Q10. Which operator finds leftovers?`%`.**Practice —** `/` `//` `%`

1. Find the result of `20 / 6`.
2. Find the result of `20 // 6`.
3. Find the result of `20 % 6`.
4. Check whether `25` is even or odd.
5. Find how many complete teams of `4` can be created from `18` students.
6. Find how many students remain.
7. Convert `150` seconds into minutes and remaining seconds.
8. Convert `185` minutes into hours and remaining minutes.
9. For `105` records and `10` records per page, find the full pages and remaining records.
10. Explain why `-10 // 3` returns `-4`.

Comparison Operators

Comparison operators compare two values. The result is always a Boolean value (`True` or `False`).

Operator	Meaning
<code>==</code>	Equal to
<code>!=</code>	Not equal to
<code>></code>	Greater than
<code><</code>	Less than
<code>>=</code>	Greater than or equal to
<code><=</code>	Less than or equal to

Let's check placement eligibility.

PYTHON

Copy

```
cgpa = 8.5
required_cgpa = 7.0
```

Equal To `==`

PYTHON

Copy

```
required_cgpa = 7.0
cgpa = 8.5
print(cgpa == required_cgpa)
```

Output:

TEXT

Copy

```
False
```

Not Equal To `!=`

PYTHON

Copy

```
required_cgpa = 7.0  
cgpa = 8.5  
print(cgpa != required_cgpa)
```

Output:

TEXT

Copy

True

Greater Than `>`

PYTHON

Copy

```
required_cgpa = 7.0  
cgpa = 8.5  
print(cgpa > required_cgpa)
```

Output:

TEXT

Copy

True

Less Than `<`

PYTHON

Copy

```
required_cgpa = 7.0  
cgpa = 8.5  
print(cgpa < required_cgpa)
```

Output:

TEXT

Copy

False

Greater Than or Equal To `>=`

PYTHON

Copy

```
required_cgpa = 7.0  
cgpa = 8.5  
print(cgpa >= required_cgpa)
```

Output:

TEXT

Copy

True

Less Than or Equal To `<=`

PYTHON

Copy

```
backlogs = 0  
print(backlogs <= 0)
```

Output:

TEXT

Copy

True

`=` VS `==`

This is one of the most common fresher mistakes.

Operator	Purpose
=	Assignment
==	Comparison

PYTHON

Copy

```
cgpa = 8.5
```

Assigns 8.5 to cgpa.

PYTHON

Copy

```
cgpa = 8.5
cgpa == 8.5
```

Checks whether cgpa is equal to 8.5.

Placement Favorite: Difference between = and ==.

Assignment Operators

Assignment operators assign or update values.

Operator	Example	Same As
=	x = 10	Assign 10
+=	x += 5	x = x + 5
-=	x -= 5	x = x - 5
*=	x *= 5	x = x * 5
/=	x /= 5	x = x / 5
//=	x //= 5	x = x // 5
%=	x %= 5	x = x % 5
**=	x **= 2	x = x ** 2

Assignment =

PYTHON

Copy

```
cgpa = 8.5
```

Addition Assignment +=

PYTHON

Copy

```
cgpa = 8.0
cgpa += 0.5
print(cgpa)
```

Output:

TEXT

Copy

```
8.5
```

This is the same as:

PYTHON

Copy

```
cgpa = 8.0  
cgpa = cgpa + 0.5
```

Subtraction Assignment -=

PYTHON

Copy

```
backlogs = 2  
backlogs -= 1  
print(backlogs)
```

Output:

TEXT

Copy

```
1
```

Multiplication Assignment *=

PYTHON

Copy

```
stipend = 15000  
stipend *= 2  
print(stipend)
```

Output:

TEXT

Copy

```
30000
```

Compound assignment operators make code shorter and cleaner.

Logical Operators

Logical operators combine or modify conditions.

Operator	Meaning
<code>and</code>	Both conditions must be true
<code>or</code>	At least one condition must be true
<code>not</code>	Reverses the truth value

Let's use placement eligibility.

PYTHON

Copy

```
cgpa = 8.5
backlogs = 0
skills = ["Python", "SQL", "Java"]
```

`and`

Returns `True` when both conditions are `True`.

PYTHON

Copy

```
backlogs = 0
cgpa = 8.5
print(cgpa >= 7.0 and backlogs == 0)
```

Output:

TEXT

Copy

True

Visual understanding:

Condition 1	Condition 2	and
True	True	True
True	False	False
False	True	False
False	False	False

Placement Example

PYTHON

Copy

```
backlogs = 0
cgpa = 8.5
eligible = cgpa >= 7.0 and backlogs == 0
print(eligible)
```

The student is eligible only when **both** requirements are satisfied.

or

Returns **True** when at least one condition is **True**.

PYTHON

Copy

```
skills = ["Python", "SQL", "Java"]
print("Python" in skills or "Java" in skills)
```

Output:

TEXT

Copy

True

Visual understanding:

Condition 1	Condition 2	or
True	True	True
True	False	True
False	True	True
False	False	False

Placement Example

A company accepts Python or Java developers.

PYTHON

Copy

```
skills = ["Python", "SQL", "Java"]  
has_skill = "Python" in skills or "Java" in skills  
print(has_skill)
```

At least one required skill is enough.

not

Reverses the truth value.

PYTHON

Copy

```
is_placed = False  
print(not is_placed)
```

Output:

TEXT

Copy

True

TEXT

Copy

```
not True → False  
not False → True
```

Placement Example

PYTHON

Copy

```
is_placed = False  
if not is_placed:  
    print("Student is available for placement")
```

Logical Operator Quick Understanding

Output:

TEXT

Copy

```
Student is available for placement
```

Membership Operators

Membership operators check whether a value exists in a collection.

Operator	Meaning
<code>in</code>	Value exists
<code>not in</code>	Value does not exist

Let's use student skills.

PYTHON

Copy

```
skills = ["Python", "SQL", "Java"]
```

in

PYTHON

Copy

```
skills = ["Python", "SQL", "Java"]  
print("Python" in skills)
```

Output:

TEXT

Copy

```
True
```

not in

PYTHON

Copy

```
skills = ["Python", "SQL", "Java"]  
print("Git" not in skills)
```

Output:

TEXT

Copy

```
True
```

Real-Time Placement Example

PYTHON

Copy

```
skills = ["Python", "SQL", "Java"]  
required_skill = "Python"  
eligible = required_skill in skills  
print(eligible)
```

Membership operators are commonly used with strings, lists, tuples, sets, and dictionaries.

Membership with Dictionary

PYTHON

Copy

```
student = {  
    "name": "Arun",  
    "cgpa": 8.5  
}  
print("name" in student)
```

Output:

TEXT

Copy

True

Important

For dictionaries, `in` checks keys by default.

PYTHON

Copy

```
student = {  
print("Arun" in student)
```

Output:

TEXT

Copy

False

To check values:

PYTHON

Copy

```
student = {  
print("Arun" in student.values())
```

Output:

TEXT

Copy

True

Identity Operators

Identity operators check whether two names refer to the **same object**.

Operator	Meaning
<code>is</code>	Same object
<code>is not</code>	Different objects

Consider:

PYTHON

Copy

```
skills = ["Python", "SQL"]  
student_skills = skills
```

Both names refer to the same list.

PYTHON

Copy

```
student_skills = skills  
skills = ["Python", "SQL"]  
print(skills is student_skills)
```

Output:

TEXT

Copy

True

Now create another list:

PYTHON

Copy

```
skills = ["Python", "SQL"]
required_skills = ["Python", "SQL"]
print(skills == required_skills)
print(skills is required_skills)
```

Output:

TEXT

Copy

```
True
False
```

Why?

TEXT

Copy

```
== → Same value?
is → Same object?
```

The lists contain the same values but are different objects.

== VS **is**

Operator	Checks
==	Value equality
is	Object identity

PYTHON

Copy

```
a = ["Python", "SQL"]
b = ["Python", "SQL"]
print(a == b) # True
print(a is b) # False
```

Placement Favorite: Difference between **==** and **is**.

Correct Use of `is`

`is` is commonly used with `None`.

PYTHON

Copy

```
result = None
print(result is None)
```

Recommended:

PYTHON

Copy

```
result = None
if result is None:
    print("No result available")
```

Avoid using `is` for normal value comparison.

PYTHON

Copy

```
age = 21
age is 21 # Avoid
```

Use:

PYTHON

Copy

```
age = 21
age == 21
```

Bitwise Operators

Bitwise operators perform operations on the binary representation of integers.

Operator	Name
&	Bitwise AND
	Bitwise OR
^	Bitwise XOR
~	Bitwise NOT
<<	Left Shift
>>	Right Shift

Consider:

PYTHON

Copy

```
a = 5
b = 3
print(b)
```

Binary values:

Output:

TEXT

Copy

3

Bitwise AND &

PYTHON

Copy

```
print(5 & 3)
```

Binary:

Output:

TEXT

Copy

1

Output:

TEXT

Copy

1

Bitwise OR $\mid$

PYTHON

Copy

```
print(5 | 3)
```

Binary:

Output:

TEXT

Copy

7

Output:

TEXT

Copy

7

Bitwise XOR $\wedge$

Returns 1 when the bits are different.

Binary:

Output:

TEXT

Copy

6

Output:

TEXT

Copy

6

Bitwise NOT ~

In Python:

PYTHON

Copy

```
~x = -(x + 1)
```

Example:

PYTHON

Copy

```
print(~5)
```

Output:

TEXT

Copy

-6

Left Shift <<

Shifts bits to the left.

PYTHON

Copy

```
print(5 << 1)
```

Output:

TEXT

Copy

10

For non-negative integers, shifting left by one position is equivalent to multiplying by 2.

TEXT

Copy

5 << 1 → 10

Right Shift >>

Shifts bits to the right.

PYTHON

Copy

```
print(5 >> 1)
```

Output:

TEXT

Copy

2

For non-negative integers, shifting right by one position behaves like floor division by 2.

TEXT

Copy

5 >> 1 → 2

Real-Time Usage of Bitwise Operators

Bitwise operators are commonly seen in permission flags, low-level programming, networking, embedded systems, image processing, and binary data handling. For fresher placement preparation, understand the **basic operation and output**.

Operator Precedence

When multiple operators are used in one expression, Python follows an order of evaluation.

PYTHON

Copy

```
result = 10 + 5 * 2
print(result)
```

Output:

TEXT

Copy

20

Why not 30? Multiplication happens before addition.

TEXT

Copy

```
10 + 5 * 2
10 + 10
20
```

Using parentheses:

PYTHON

Copy

```
result = (10 + 5) * 2
print(result)
```


TEXT

Copy

30

Simplified Precedence Order

Priority	Operators
1	<code>()</code>
2	<code>**</code>
3	Unary <code>+</code> , <code>-</code> , <code>~</code>
4	<code>*</code> , <code>/</code> , <code>//</code> , <code>%</code>
5	<code>+</code> , <code>-</code>
6	<code><<</code> , <code>>></code>
7	<code>&</code>
8	<code>^</code>
9	<code>\ </code>
10	Comparisons, <code>in</code> , <code>is</code>
11	<code>not</code>
12	<code>and</code>
13	<code>or</code>

Beginner Rule

When an expression is difficult to understand, use parentheses.

PYTHON

Copy

```
backlogs = 0
cgpa = 8.5
eligible = (cgpa >= 7.0) and (backlogs == 0)
```

This makes the code easier to read.

Complete Placement Eligibility Example

PYTHON

Copy

```
cgpa = 8.5
age = 21
backlogs = 0
skills = ["Python", "SQL", "Java"]
is_placed = False

has_required_cgpa = cgpa >= 7.0
has_no_backlogs = backlogs == 0
has_required_skill = "Python" in skills
is_available = not is_placed

eligible = (
    has_required_cgpa
    and has_no_backlogs
    and has_required_skill
    and is_available
)

print("Placement Eligible:", eligible)
```

Output:

TEXT

Copy

```
Placement Eligible: True
```

How it works:

TEXT

Copy

```
cgpa >= 7.0      → True
backlogs == 0    → True
"Python" in skills → True
not is_placed    → True
True and True and True and True
                ↓
                True
```

This is how operators work together in real application logic.

Common Fresher Mistakes

X Mistake	Problem	✓ Correct Understanding
<code>=</code> used for comparison	<code>=</code> is assignment	Use <code>==</code>
Confusing <code>/</code> and <code>//</code>	They return different results	<code>/</code> division, <code>//</code> floor division
Thinking <code>%</code> gives percentage	<code>%</code> is modulus	It returns the remainder
Using <code>and</code> when one condition is enough	<code>and</code> requires all conditions	Use <code>or</code> for any condition
Using <code>or</code> when all conditions are required	One true condition is enough for <code>or</code>	Use <code>and</code>
Using <code>is</code> for value comparison	<code>is</code> checks identity	Use <code>==</code>
<code>"Arun" in student</code> expecting dictionary value check	<code>in</code> checks dictionary keys	Use <code>student.values()</code>
Ignoring operator precedence	Output may differ	Use parentheses
Thinking <code>//</code> simply removes decimals	Incorrect for negative values	It floors the result

Placement Quick Points

- Operators perform operations on values; values used with operators are called operands.
- Arithmetic operators perform calculations.

- `/` performs division and returns a float.
- `//` performs floor division.
- `%` returns the remainder.
- `**` performs exponentiation.
- Comparison operators return Boolean results.
- `=` assigns a value; `==` compares values.
- `and` requires all conditions to be `True`.
- `or` requires at least one condition to be `True`.
- `not` reverses a Boolean condition.
- `in` and `not in` are membership operators.
- `is` checks object identity; `==` checks value equality.
- Bitwise operators work with binary representations of integers.
- Parentheses improve expression clarity.

Interview Questions

Q1. What is an operator?

An operator is a symbol or keyword used to perform an operation on values.

Q2. What is an operand?

A value on which an operator performs an operation.

Q3. What is the difference between `/` and `//`?

`/` performs division. `//` performs floor division.

Q4. What does `%` return?

The remainder.

Q5. What is the difference between `=` and `==`?

`=` assigns a value. `==` compares values.

Q6. What is the difference between `and` and `or`?

`and` requires all conditions to be `True`. `or` requires at least one condition to be `True`.

Q7. What does `not` do?

It reverses a Boolean condition.

Q8. What is the difference between `in` and `not in`?

`in` checks whether a value exists. `not in` checks whether it does not exist.

Q9. What is the difference between `==` and `is`?

`==` compares values. `is` compares object identity.

Q10. What does `10 % 3` return?

1.

Q11. What does `2 ** 3` return?

8.

Q12. What does `10 // 3` return?

3.

Q13. What does `-10 // 3` return?

-4.

Q14. What does `in` check in a dictionary?

Dictionary keys by default.

Q15.What is operator precedence?

The order in which Python evaluates operators in an expression.

Practice

Use:

PYTHON

Copy

```
cgpa = 8.5
age = 21
backlogs = 0
skills = ["Python", "SQL", "Java"]
is_placed = False
```

1. Check whether `cgpa` is greater than or equal to `7.0`.
2. Check whether the student has zero backlogs.
3. Check whether `"Python"` exists in `skills`.
4. Check whether `"Git"` does not exist in `skills`.
5. Check whether the student is not placed.
6. Combine CGPA and backlog conditions using `and`.
7. Check whether the student knows `"Python"` or `"Java"`.
8. Increase `cgpa` by `0.5` using an assignment operator.
9. Find the remainder when `10` is divided by `3` (use the `%` operator).
10. Find the result of `2 ** 5`.
11. Explain the difference between `/` and `//`.
12. Explain the difference between `==` and `is`.
13. Predict the output:

PYTHON

Copy

```
print(10 + 5 * 2)
```

1. Predict the output:

PYTHON

Copy

```
print((10 + 5) * 2)
```

1. Predict the output:

PYTHON

Copy

```
print(-10 // 3)
```

1. Find the mistake:

PYTHON

Copy

```
age is 21
```

1. Create a placement eligibility condition using CGPA `>= 7.0`, zero backlogs, `"Python"` in skills, and student not already placed.

End of Level 2

Data Structures

Level: 3 | Version: v1.0 | Author: Meptrasoft

Data Structures at a Glance

A **data structure** is a way to organize data so a program can store it, access it, update it, and process it easily.

Python provides several built-in data structures. In this module, we will focus on strings, lists, tuples, sets, and dictionaries.

List	Tuple	Set	Dict
[1, 2, 3]	(1, 2, 3)	{1, 2, 3}	{"a": 1}
Ordered ✓	Ordered ✓	Ordered ✗	Ordered ✓
Mutable ✓	Mutable ✗	Mutable ✓	Mutable ✓
Duplicates ✓	Duplicates ✓	Duplicates ✗	Keys unique ✓
Ordered, changeable	Fixed, unchangeable	Unique items only	Key : value pairs

Python's four core built-in data structures compared at a glance. Each is covered in depth in its own section below.

Strings

What is a String?

A string is a sequence of characters used to store text.

PYTHON

Copy

```
student_name = "Arun Kumar"
```

The value "Arun Kumar" is a string.

Strings can contain:

- Letters
- Numbers
- Spaces
- Special characters

PYTHON

Copy

```
message = "Hello Python!"  
student_id = "STU101"
```

Why Do We Use Strings?

Strings are used to store and process text data.

Real-world examples:

- Student names
- Email addresses
- User input
- Password text
- File names
- API text data
- Log messages

Example:

```
PYTHON
```

[Copy](#)

```
student_name = "Arun Kumar"
```

In a real application, we may need to:

- Remove extra spaces
- Convert text to lowercase
- Search for a word
- Replace text
- Split the name
- Validate the input

Python string methods help us perform these operations.

Key Features of Strings

Feature	Meaning
Ordered	Characters maintain their position
Immutable	Characters cannot be changed directly
Indexed	Characters can be accessed using indexes
Iterable	Can be processed character by character
Supports Slicing	Part of a string can be extracted

Creating a String

Single Quotes

PYTHON

Copy

```
student_name = 'Arun Kumar'
```

Double Quotes

PYTHON

Copy

```
student_name = "Arun Kumar"
```

Both are valid.

Triple Quotes

Triple quotes are used for multi-line strings.

PYTHON

Copy

```
profile = """Arun Kumar  
CSE Student  
Python Developer"""
```

Accessing String Characters

Strings support indexing.

PYTHON

Copy

```
student_name = "Arun Kumar"  
print(student_name[0]) # A  
print(student_name[1]) # r  
print(student_name[-1]) # r
```

- Index `0` → first character
- Index `-1` → last character

*Positive and negative indexing are covered separately in the **Indexing** topic.*

Strings are Immutable

Strings cannot be changed directly.

PYTHON

Copy

```
student_name = "Arun"  
student_name[0] = "K"
```

Error:

Output:

TEXT

Copy

```
TypeError: 'str' object does not support item assignment
```

Correct Approach

Create a new string.

PYTHON

Copy

```
student_name = "Arun"  
student_name = "K" + student_name[1:]  
print(student_name)
```

Output:

TEXT

Copy

```
Krun
```

Important: String methods usually return a **new string** because strings are immutable.

String Length

Use `len()` to count characters.

PYTHON

Copy

```
student_name = "Arun Kumar"
print(len(student_name))
```

Output:

TEXT

Copy

```
10
```

Spaces are also counted as characters.

Cleaning User Input

Consider input received from a form:

PYTHON

Copy

```
student_name = " Arun Kumar "
```

The string contains unwanted spaces.

`strip()`

Removes spaces from both ends.

PYTHON

Copy

```
student_name = " Arun Kumar "
student_name = student_name.strip()
print(student_name)
```

Output:

TEXT

Copy

```
Arun Kumar
```

Removes spaces from the left side.

PYTHON

Copy

```
student_name = "  Arun Kumar  "
student_name = student_name.lstrip()
```

rstrip()

Removes spaces from the right side.

PYTHON

Copy

```
student_name = "  Arun Kumar  "
student_name = student_name.rstrip()
```

strip() VS lstrip() VS rstrip()

Method	Removes From
strip()	Both sides
lstrip()	Left side
rstrip()	Right side

Changing String Case

Let's use:

PYTHON

Copy

```
student_name = "Arun Kumar"
```

lower()

Converts text to lowercase.

PYTHON

Copy

```
student_name = "Arun Kumar"  
print(student_name.lower())
```

Output:

TEXT

Copy

arun kumar

upper()

Converts text to uppercase.

PYTHON

Copy

```
student_name = "Arun Kumar"  
print(student_name.upper())
```

Output:

TEXT

Copy

ARUN KUMAR

title()

Converts the first character of each word to uppercase.

PYTHON

Copy

```
student_name = "Arun Kumar"  
print(student_name.title())
```

Output:

TEXT

Copy

Arun Kumar

capitalize()

Converts the first character of the string to uppercase and the remaining characters to lowercase.

PYTHON

Copy

```
message = "python DEVELOPER"  
print(message.capitalize())
```

Output:

TEXT

Copy

Python developer

swapcase()

Changes uppercase to lowercase and lowercase to uppercase.

PYTHON

Copy

```
student_name = "Arun Kumar"  
print(student_name.swapcase())
```

Output:

TEXT

Copy

aRUN kUMAR

String Case Methods

Method	Purpose
<code>lower()</code>	Convert to lowercase
<code>upper()</code>	Convert to uppercase
<code>title()</code>	Capitalize each word
<code>capitalize()</code>	Capitalize the string's first character and lowercase the rest
<code>swapcase()</code>	Swap uppercase and lowercase

Searching in a String

Let's use a placement skill string.

PYTHON

Copy

```
skills = "Python SQL Java"
```

`find()`

Returns the starting index of the first matching text.

PYTHON

Copy

```
skills = "Python SQL Java"
print(skills.find("SQL"))
```

Output:

TEXT

Copy

7

If the value is not found:

PYTHON

Copy

```
skills = "Python SQL Java"
print(skills.find("C++"))
```

Output:

TEXT

Copy

-1

index()

Also returns the index of matching text.

PYTHON

Copy

```
skills = "Python SQL Java"
print(skills.index("SQL"))
```

Output:

TEXT

Copy

7

If the value does not exist, `index()` raises a `ValueError`.

find() VS index()

Method	Value Found	Value Not Found
<code>find()</code>	Returns index	Returns <code>-1</code>
<code>index()</code>	Returns index	Raises <code>ValueError</code>

Placement Favorite: Difference between `find()` and `index()`.

Checking Whether Text Exists

Use `in` and `not in`.

PYTHON

Copy

```
skills = "Python SQL Java"
print("Python" in skills)      # True
print("C++" in skills)        # False
print("C++" not in skills)     # True
```

Replacing Text

Use `replace()` to replace text.

PYTHON

Copy

```
skills = "Python SQL Java"
skills = skills.replace("Java", "C++")
print(skills)
```

Output:

TEXT

Copy

Python SQL C++

The original string is not modified directly. `replace()` returns a new string.

Counting Text

Use `count()` to count occurrences.

PYTHON

Copy

```
skills = "Python SQL Python Java"
print(skills.count("Python"))
```

Output:

TEXT

Copy

```
2
```

Splitting a String

Real applications often receive data as text.

PYTHON

Copy

```
skills = "Python,SQL,Java"
```

Use `split()` to convert the string into a list.

PYTHON

Copy

```
skills = "Python,SQL,Java"
skill_list = skills.split(",")
print(skill_list)
```

Output:

TEXT

Copy

```
['Python', 'SQL', 'Java']
```

Joining Strings

Use `join()` to combine multiple strings.

PYTHON

Copy

```
skill_list = ["Python", "SQL", "Java"]
skills = ", ".join(skill_list)
print(skills)
```

TEXT

Copy

Python, SQL, Java

split() VS join()

Method	Conversion
split()	String → List
join()	Iterable of strings → String

TEXT

Copy

```
"Python,SQL,Java"
  ↓ split()
["Python", "SQL", "Java"]
  ↓ join()
"Python, SQL, Java"
```

Very Important in Real Projects: API data processing, CSV-like text, form data, and file processing.

Checking String Content

Python provides methods to validate text.

Method	Checks
<code>isalpha()</code>	Only alphabetic characters
<code>isdigit()</code>	Only digit characters
<code>isalnum()</code>	Only letters and digits
<code>isspace()</code>	Only whitespace characters
<code>islower()</code>	Lowercase
<code>isupper()</code>	Uppercase

`isalpha()`

PYTHON

Copy

```
name = "Arun"
print(name.isalpha()) # True
```

PYTHON

Copy

```
name = "Arun123"
print(name.isalpha()) # False
```

`isdigit()`

PYTHON

Copy

```
student_id = "101"
print(student_id.isdigit()) # True
```

`isalnum()`

PYTHON

Copy

```
student_id = "STU101"
print(student_id.isalnum()) # True
```

Real-Time Input Validation

Consider a student registration form.

PYTHON

Copy

```
student_name = "Arun"  
age = "21"  
student_id = "STU101"  
print(student_name.isalpha()) # True  
print(age.isdigit()) # True  
print(student_id.isalnum()) # True
```

These methods are useful for basic input validation.

Checking Start and End

`startswith()`

Checks whether a string starts with specific text.

PYTHON

Copy

```
student_id = "STU101"  
print(student_id.startswith("STU"))
```

Output:

TEXT

Copy

True

`endswith()`

Checks whether a string ends with specific text.

Output:

TEXT

Copy

True

Real-Time Use

PYTHON

Copy

```
resume_file = "arun_resume.pdf"
if resume_file.endswith(".pdf"):
    print("Valid resume format")
```

Useful for:

- File extension checking
- URL checking
- ID prefix validation
- Email/domain checks

String Concatenation

Strings can be combined using `+`.

PYTHON

Copy

```
first_name = "Arun"
last_name = "Kumar"
full_name = first_name + " " + last_name
print(full_name)
```

Output:

TEXT

Copy

Arun Kumar

String Repetition

Use `*` to repeat a string.

PYTHON

Copy

```
symbol = "-"  
print(symbol * 20)
```

Output:

TEXT

Copy

```
-----
```

String Formatting with f-Strings

Consider student placement data:

PYTHON

Copy

```
student_name = "Arun Kumar"  
cgpa = 8.5  
skill = "Python"
```

Using an f-string:

PYTHON

Copy

```
skill = 'Sample'  
cgpa = 8.5  
student_name = "Arun Kumar"  
message = f"{student_name} has a CGPA of {cgpa} and knows {skill}."  
print(message)
```

Output:

F-strings make formatted output **clean and readable**.

Escape Characters

Escape characters represent special characters inside strings.

Escape Character	Meaning
<code>\n</code>	New line
<code>\t</code>	Tab
<code>\\</code>	Backslash
<code>\"</code>	Double quote
<code>\'</code>	Single quote

New Line

PYTHON

Copy

```
print("Name: Arun\nSkill: Python")
```

Output:

TEXT

Copy

```
Name: Arun
Skill: Python
```

Tab

PYTHON

Copy

```
print("Name\tCGPA")
print("Arun\t8.5")
```

Common String Methods

Method	Purpose
<code>strip()</code>	Remove characters from both ends; whitespace by default
<code>lower()</code>	Convert to lowercase
<code>upper()</code>	Convert to uppercase
<code>title()</code>	Convert each word to title case
<code>capitalize()</code>	Capitalize first character and lowercase the rest
<code>find()</code>	Find text and return <code>-1</code> if missing
<code>index()</code>	Find text and raise error if missing
<code>replace()</code>	Replace text
<code>count()</code>	Count occurrences
<code>split()</code>	Split string into a list
<code>join()</code>	Join an iterable of strings
<code>startswith()</code>	Check starting text
<code>endswith()</code>	Check ending text
<code>isalpha()</code>	Check alphabetic characters
<code>isdigit()</code>	Check digit characters
<code>isalnum()</code>	Check letters and digits

Common Fresher Mistakes

X Mistake	Problem	✓ Correct Understanding
<code>name[0] = "K"</code>	Strings are immutable	Create a new string
<code>age = input()</code> expecting integer	<code>input()</code> returns a string	Use <code>int(input())</code>
<code>find()</code> expected to raise an error	<code>find()</code> returns <code>-1</code>	<code>index()</code> raises <code>ValueError</code>
<code>skills.split()</code> expecting original string to change	Strings are immutable	Store the returned value
<code>"".join([1, 2, 3])</code>	Items are not strings	Convert items to strings
<code>name.isalpha()</code> on <code>"Arun Kumar"</code> expecting <code>True</code>	Space is not alphabetic	Validate words separately if needed
Forgetting spaces in <code>+</code>	Strings join exactly as given	Add <code>" "</code> manually

Practical Example — Cleaning Placement Form Data

PYTHON

Copy

```
student_name = "  arun kumar  "
skills = "Python,SQL,Java"
resume_file = "arun_resume.pdf"
student_name = student_name.strip().title()
skill_list = skills.split(",")
print("Student:", student_name)
print("Skills:", skill_list)
if resume_file.endswith(".pdf"):
    print("Valid Resume")
```

Output:

TEXT

Copy

```
Student: Arun Kumar
Skills: ['Python', 'SQL', 'Java']
Valid Resume
```

This is similar to text cleaning and validation performed in real applications.

Placement Quick Points

- A string is a sequence of characters.
- Strings are ordered and immutable.
- Strings support indexing and slicing.
- String indexing starts from `0`.
- String methods return new strings.
- `strip()` removes characters from both ends; with no argument, it removes surrounding whitespace.
- `find()` returns `-1` when text is not found.
- `index()` raises `ValueError` when text is not found.
- `split()` converts a string into a list.

- `join()` combines an iterable of strings into a string.
- `replace()` returns a new string.
- `in` checks whether text exists.
- f-strings are used for clean string formatting.
- `startswith()` and `endswith()` are useful for validation.

Interview Questions

Q1. What is a string in Python?

A string is an immutable sequence of characters.

Q2. Are strings mutable or immutable?

Immutable.

Q3. Can strings be indexed?

Yes.

Q4. What is the difference between `find()` and `index()`?

`find()` returns `-1` when the value is missing. `index()` raises a `ValueError`.

Q5. What does `strip()` do?

It removes leading and trailing whitespace by default. If characters are provided, it removes those characters from both ends.

Q6. What is the difference between `split()` and `join()`?

`split()` converts a string into a list. `join()` combines an iterable of strings into a string.

Q7. Do string methods modify the original string?

No. Strings are immutable.

Q8. How do you check whether a string contains only digits?

```
value.isdigit()
```

Q9. What is an f-string?

A readable way to insert expressions into a formatted string.

Q10. What does `find()` return if the value is not found?

```
-1
```

Q11. Can we change a character using an index?

No. Strings do not support item assignment.

Q12. What does `join()` require?

An iterable containing strings.

Practice

Use:

PYTHON

Copy

```
student_name = "  arun kumar  "  
skills = "Python,SQL,Java"  
student_id = "STU101"  
resume_file = "arun_resume.pdf"
```

1. Remove extra spaces from `student_name`.
2. Convert the student name to title case.
3. Convert the name to uppercase.
4. Find the index of `"SQL"` in `skills`.
5. Replace `"Java"` with `"C++"`.
6. Convert `skills` into a list.
7. Join the skill list using `" | "`.
8. Check whether `student_id` starts with `"STU"`.
9. Check whether `resume_file` ends with `".pdf"`.

10. Check whether "Python" exists in skills.
11. Count how many times "Python" appears.
12. Explain the difference between find() and index().
13. Find the error:

PYTHON

Copy

```
student_name = " arun kumar "  
student_name[0] = "A"
```

1. Predict the output:

PYTHON

Copy

```
text = "Python"  
print(text.find("Java"))
```

1. Explain why this causes an error:

PYTHON

Copy

```
", ".join([10, 20, 30])
```

Lists

What is a List?

A **list** is a collection used to store multiple values under one name.

PYTHON

Copy

```
skills = ["Python", "SQL", "Java"]
```

Instead of creating multiple variables:

PYTHON

Copy

```
skill1 = "Python"  
skill2 = "SQL"  
skill3 = "Java"
```

We can store all values in one list.

PYTHON

Copy

```
skills = ["Python", "SQL", "Java"]
```

Why Do We Use Lists?

Lists help us store, access, update, and manage multiple values together.

PYTHON

Copy

```
marks = [85, 90, 78, 92]
```

Real-world examples:

- Student marks
- Product names
- Employee records
- File names
- API results
- Model predictions

Key Features of Lists

Feature	Meaning
Ordered	Maintains element order
Mutable	Elements can be changed
Dynamic	Can grow or shrink
Allows Duplicates	Same value can appear multiple times
Heterogeneous	Can store different data types
Indexed	Elements can be accessed using indexes

Creating a List

Empty List

PYTHON

[Copy](#)

```
numbers = []
```

List with Values

PYTHON

[Copy](#)

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

Mixed Data Types

PYTHON

[Copy](#)

```
data = [10, "Python", 3.14, True]
```

Duplicate Values

PYTHON

Copy

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

Lists allow duplicate values.

Nested List

A list can contain another list.

PYTHON

Copy

```
data = [10, [20, 30], 40]
```

Accessing List Elements

List elements are accessed using an **index**.

PYTHON

Copy

```
skills = ["Python", "SQL", "Java"]  
print(skills[0])    # Python  
print(skills[1])    # SQL  
print(skills[-1])   # Java
```

- Index `0` → first element
- Index `-1` → last element

*Positive and negative indexing are covered separately in the **Indexing** topic.*

Updating List Elements

Lists are **mutable**, so their elements can be changed.

PYTHON

Copy

```
skills = ["Python", "SQL", "Java"]
skills[2] = "C++"
print(skills)
```

Output:

TEXT

Copy

```
['Python', 'SQL', 'C++']
```

Adding Elements

append()

Adds one item to the end of the list.

PYTHON

Copy

```
skills = ["Python", "SQL"]
skills.append("Java")
print(skills)
```

Output:

TEXT

Copy

```
['Python', 'SQL', 'Java']
```

extend()

Adds multiple elements from an iterable.

PYTHON

Copy

```
skills = ["Python", "SQL"]
skills.extend(["Java", "C++"])
print(skills)
```

TEXT

Copy

```
['Python', 'SQL', 'Java', 'C++']
```

insert()

Adds an element at a specific index.

PYTHON

Copy

```
skills = ["Python", "Java"]
skills.insert(1, "SQL")
print(skills)
```

Output:

TEXT

Copy

```
['Python', 'SQL', 'Java']
```

append() VS **extend()**

Method	Adds	Example	Result
append()	One item	<code>a.append([3, 4])</code>	<code>[1, 2, [3, 4]]</code>
extend()	Elements from an iterable	<code>a.extend([3, 4])</code>	<code>[1, 2, 3, 4]</code>

Example:

PYTHON

Copy

```
a = [1, 2]
a.append([3, 4])
print(a)
```

Output:

TEXT

Copy

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

PYTHON

Copy

```
a = [1, 2]
a.extend([3, 4])
print(a)
```

Output:

TEXT

Copy

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

Placement Favorite: `append()` vs `extend()`.

Removing Elements

`remove()`

Removes the first matching value.

PYTHON

Copy

```
numbers = [10, 20, 30, 20]
numbers.remove(20)
print(numbers)
```

Output:

TEXT

Copy

```
[10, 30, 20]
```

`pop()`

Removes and returns an element.

PYTHON

Copy

```
numbers = [10, 20, 30]
removed = numbers.pop()
print(removed) # 30
print(numbers) # [10, 20]
```

Remove using an index:

PYTHON

Copy

```
numbers = [10, 20, 30]
numbers.pop(1)
```

clear()

Removes all elements.

PYTHON

Copy

```
numbers = [10, 20, 30]
numbers.clear()
print(numbers) # []
```

remove() VS pop()

Method	Removes By	Returns Removed Value
remove()	Value	No
pop()	Index	Yes

PYTHON

Copy

```
numbers = [10, 20, 30]
numbers.remove(20) # Remove value
numbers.pop(0) # Remove by index
```


Searching and Counting

index()

Returns the index of the first matching value.

PYTHON

Copy

```
numbers = [10, 20, 30]
print(numbers.index(20)) # 1
```

count()

Counts how many times a value appears.

PYTHON

Copy

```
numbers = [10, 20, 20, 30]
print(numbers.count(20)) # 2
```

Membership Operators

Use `in` and `not in` to check whether a value exists.

PYTHON

Copy

```
skills = ["Python", "SQL"]
print("Python" in skills)      # True
print("Java" not in skills)    # True
```

Sorting a List

sort()

Sorts the original list.

PYTHON

Copy

```
numbers = [40, 10, 30, 20]
numbers.sort()
print(numbers)
```

Output:

TEXT

Copy

```
[10, 20, 30, 40]
```

Descending order:

PYTHON

Copy

```
numbers = [40, 10, 30, 20]
numbers.sort(reverse=True)
```

`sort()` VS `sorted()`

<code>sort()</code>	<code>sorted()</code>
List method	Built-in function
Changes the original list	Returns a new sorted list
Returns <code>None</code>	Returns a sorted list

PYTHON

Copy

```
numbers = [30, 10, 20]
result = numbers.sort()
print(result) # None
```

Using `sorted()`:

PYTHON

Copy

```
numbers = [30, 10, 20]
result = sorted(numbers)
print(result)    # [10, 20, 30]
print(numbers)  # [30, 10, 20]
```

Common Placement Question: Why does `list.sort()` return `None`? Because it modifies the original list in place.

Reversing a List

The `reverse()` method reverses the original list.

PYTHON

Copy

```
numbers = [10, 20, 30]
numbers.reverse()
print(numbers)
```

Output:

TEXT

Copy

```
[30, 20, 10]
```

Useful Built-in Functions

Function	Purpose
<code>len()</code>	Number of elements
<code>max()</code>	Largest value
<code>min()</code>	Smallest value
<code>sum()</code>	Sum of numeric values
<code>sorted()</code>	Returns a sorted list

PYTHON

Copy

```
marks = [80, 90, 70, 85]
print(len(marks)) # 4
print(max(marks)) # 90
print(min(marks)) # 70
print(sum(marks)) # 325
```

Copying a List

Consider this code:

PYTHON

Copy

```
a = [10, 20, 30]
b = a
b.append(40)
print(a)
```

Output:

TEXT

Copy

```
[10, 20, 30, 40]
```

Why did `a` change? Because `a` and `b` refer to the same list object.

Use `copy()` to create a separate shallow copy.

PYTHON

Copy

```
a = [10, 20, 30]
b = a.copy()
b.append(40)
print(a) # [10, 20, 30]
print(b) # [10, 20, 30, 40]
```

Shallow copy and deep copy are covered separately in the **Copying Objects** topic.

Common List Methods

Method	Purpose
<code>append()</code>	Add one item
<code>extend()</code>	Add elements from an iterable
<code>insert()</code>	Add at a specific index
<code>remove()</code>	Remove first matching value
<code>pop()</code>	Remove and return an item
<code>index()</code>	Find the index of a value
<code>count()</code>	Count occurrences
<code>sort()</code>	Sort the original list
<code>reverse()</code>	Reverse the original list
<code>copy()</code>	Create a shallow copy
<code>clear()</code>	Remove all items

Common Fresher Mistakes

X Mistake	Problem	✓ Correct Understanding
<code>numbers[1]</code> expecting first item	Index starts at <code>0</code>	Use <code>numbers[0]</code>
<code>data.extend(40)</code>	<code>int</code> is not iterable	Use <code>data.append(40)</code>
<code>result = numbers.sort()</code>	<code>sort()</code> returns <code>None</code>	Use <code>sorted(numbers)</code> if a new list is needed
<code>b = a</code> expecting a copy	Both names refer to the same list	Use <code>a.copy()</code>
<code>numbers.remove(1)</code> expecting index removal	<code>remove()</code> removes by value	Use <code>pop(1)</code> for index
Modifying a list expecting immutability	Lists are mutable	List elements can change

Practical Example — Student Marks

PYTHON

Copy

```
marks = [75, 85, 60, 90]
marks.append(80)
print("Marks:", marks)
print("Highest:", max(marks))
print("Lowest:", min(marks))
print("Total:", sum(marks))
print("Average:", sum(marks) / len(marks))
```

Output:

TEXT

Copy

```
Marks: [75, 85, 60, 90, 80]
Highest: 90
Lowest: 60
Total: 390
Average: 78.0
```

Placement Quick Points

- A list stores multiple values.
- Lists are ordered and mutable.
- Lists allow duplicate values.
- Lists can store different data types.
- List indexing starts from `0`.
- `append()` adds one item.
- `extend()` adds elements from an iterable.
- `remove()` removes by value.
- `pop()` removes by index and returns the removed item.
- `sort()` modifies the original list.
- `sorted()` returns a new sorted list.
- `b = a` does not create a copy.
- Use `copy()` to create a shallow copy.

Interview Questions

Q1. What is a list in Python?

A list is an ordered and mutable collection of elements.

Q2. Can a list contain duplicate values?

Yes.

Q3. Can a list store different data types?

Yes.

Q4. What is the difference between `append()` and `extend()`?

`append()` adds one item, while `extend()` adds elements from an iterable.

Q5. What is the difference between `remove()` and `pop()`?

`remove()` removes by value. `pop()` removes by index and returns the removed item.

Q6. What is the difference between `sort()` and `sorted()`?

`sort()` changes the original list. `sorted()` returns a new sorted list.

Q7. Is a list mutable or immutable?

Mutable.

Q8. Does `b = a` copy a list?

No. Both names refer to the same list.

Q9. How do you create a shallow copy of a list?

`b = a.copy()`.

Q10. What does `list.sort()` return?

`None`.

Practice

1. Create a list of five student marks.
2. Add a new mark using `append()`.
3. Add three marks using `extend()`.
4. Insert a mark at index `2`.
5. Remove a value using `remove()`.
6. Remove the last element using `pop()`.

7. Find how many times `10` appears in a list.
8. Sort a list in ascending and descending order.
9. Find the maximum, minimum, and sum of a numeric list.
10. Explain the output:

PYTHON

Copy

```
a = [1, 2, 3]
b = a
b.append(4)
print(a)
```

1. Find the error:

PYTHON

Copy

```
data = [10, 20]
data.extend(30)
```

1. Explain why this prints `None`:

PYTHON

Copy

```
numbers = [3, 1, 2]
result = numbers.sort()
print(result)
```

Tuples

What is a Tuple?

A **tuple** is a collection used to store multiple values under one name.

PYTHON

Copy

```
skills = ("Python", "SQL", "Java")
```

A **tuple** is similar to a list, but it is **immutable**.

Immutable means tuple elements cannot be replaced, added, or removed after creation.

Why Do We Use Tuples?

Tuples are useful when the collection structure should remain **fixed and unchanged**.

PYTHON

Copy

```
coordinates = (10, 20)
```

Real-world examples:

- Coordinates
- RGB color values
- Date values
- Database records
- Fixed configuration values
- Function return values

Key Features of Tuples

Feature	Meaning
Ordered	Maintains element order
Immutable	Cannot replace, add, or remove elements
Allows Duplicates	Same value can appear multiple times
Heterogeneous	Can store different data types
Indexed	Elements can be accessed using indexes

Creating a Tuple

Empty Tuple

PYTHON

Copy

```
data = ()
```

Tuple with Values

PYTHON

Copy

```
numbers = (10, 20, 30)
```

Mixed Data Types

PYTHON

Copy

```
data = (10, "Python", 3.14, True)
```

Duplicate Values

PYTHON

Copy

```
numbers = (10, 20, 20, 30)
```

Tuples allow duplicate values.

Nested Tuple

A tuple can contain another tuple.

PYTHON

Copy

```
data = (10, (20, 30), 40)
```

Creating a Single-Element Tuple

A single-element tuple must contain a comma.

PYTHON

Copy

```
data = (10,)
```

Without the comma:

PYTHON

Copy

```
data = (10)
print(type(data))
```

Output:

TEXT

Copy

```
<class 'int'>
```

With the comma:

PYTHON

Copy

```
data = (10,)
print(type(data))
```

Output:

TEXT

Copy

```
<class 'tuple'>
```

Placement Favorite: The comma creates a single-element tuple, not the parentheses alone.

Accessing Tuple Elements

Tuple elements are accessed using an **index**.

PYTHON

Copy

```
skills = ("Python", "SQL", "Java")
print(skills[0])    # Python
print(skills[1])    # SQL
print(skills[-1])   # Java
```

- Index `0` → first element
- Index `-1` → last element

*Positive and negative indexing are covered separately in the **Indexing** topic.*

Tuple is Immutable

Tuple elements cannot be replaced after creation.

PYTHON

Copy

```
skills = ("Python", "SQL", "Java")
skills[1] = "C++"
```

Error:

Output:

TEXT

Copy

```
TypeError: 'tuple' object does not support item assignment
```

This is the main difference between a **list** and **tuple**.

Can We Add or Remove Tuple Elements?

No.

Tuples do not support methods such as:

PYTHON

Copy

```
append()  
extend()  
insert()  
remove()  
pop()  
clear()
```

These methods change the collection, but tuples are immutable.

Tuple Methods

Tuples have only two commonly used methods.

Method	Purpose
<code>count()</code>	Counts occurrences of a value
<code>index()</code>	Finds the index of a value

`count()`

Counts how many times a value appears.

PYTHON

Copy

```
numbers = (10, 20, 20, 30)  
print(numbers.count(20)) # 2
```

`index()`

Returns the index of the first matching value.

PYTHON

Copy

```
numbers = (10, 20, 30)
print(numbers.index(20)) # 1
```

Membership Operators

Use `in` and `not in` to check whether a value exists.

PYTHON

Copy

```
skills = ("Python", "SQL", "Java")
print("Python" in skills)      # True
print("C++" not in skills)    # True
```

Useful Built-in Functions

Function	Purpose
<code>len()</code>	Number of elements
<code>max()</code>	Largest value
<code>min()</code>	Smallest value
<code>sum()</code>	Sum of numeric values
<code>sorted()</code>	Returns a sorted list

PYTHON

Copy

```
marks = (80, 90, 70, 85)
print(len(marks)) # 4
print(max(marks)) # 90
print(min(marks)) # 70
print(sum(marks)) # 325
```


Important: `sorted()` Returns a List

PYTHON

Copy

```
numbers = (30, 10, 20)
result = sorted(numbers)
print(result)
print(type(result))
```

Output:

TEXT

Copy

```
[10, 20, 30]
<class 'list'>
```

`Sorted()` does not return a tuple.

Tuple Packing

Storing multiple values in a tuple is called **tuple packing**.

PYTHON

Copy

```
student = ("Arun", 21, 8.5)
```

Python also allows packing without parentheses.

PYTHON

Copy

```
student = "Arun", 21, 8.5
print(type(student))
```

Output:

TEXT

Copy

```
<class 'tuple'>
```

Tuple Unpacking

Assigning tuple elements to multiple names is called **tuple unpacking**.

PYTHON

Copy

```
student = ("Arun", 21, 8.5)
name, age, cgpa = student
print(name)
print(age)
print(cgpa)
```

Output:

TEXT

Copy

```
Arun
21
8.5
```

The number of names and values should match.

PYTHON

Copy

```
student = ("Arun", 21, 8.5)
name, age = student
```

This causes a `ValueError`.

Swapping Using Tuple Unpacking

Python uses packing and unpacking while swapping values.

PYTHON

Copy

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

TEXT

Copy

20 10

Tuple with Mutable Elements

A tuple itself is immutable, but it can contain a mutable object such as a list.

PYTHON

Copy

```
data = (10, [20, 30], 40)
data[1].append(50)
print(data)
```

Output:

TEXT

Copy

```
(10, [20, 30, 50], 40)
```

Why Did the Tuple Change?

The tuple still refers to the **same list object**.

We did not replace the tuple element. We modified the list stored inside it.

PYTHON

Copy

```
data = (10, [20, 30], 40)
data[1] = [100, 200] # Error
```

Replacing a tuple element is not allowed.

Interview Important: Tuple immutability does not automatically make every object inside the tuple immutable.

List vs Tuple

Feature	List	Tuple
Syntax	[]	()
Ordered	Yes	Yes
Mutable	Yes	No
Duplicates	Yes	Yes
Indexed	Yes	Yes
Mixed Data Types	Yes	Yes
Methods	Many	count(), index()
Best Used For	Changeable data	Fixed data

Converting List and Tuple

List to Tuple

PYTHON

Copy

```
numbers = [10, 20, 30]
result = tuple(numbers)
print(result)
```

Output:

TEXT

Copy

```
(10, 20, 30)
```

Tuple to List

PYTHON

Copy

```
numbers = (10, 20, 30)
result = list(numbers)
print(result)
```

Output:

TEXT

Copy

```
[10, 20, 30]
```

How to Modify a Tuple?

A tuple cannot be modified directly.

If modification is required:

1. Convert tuple to list
2. Modify the list
3. Convert it back to tuple

PYTHON

Copy

```
numbers = (10, 20, 30)
data = list(numbers)
data.append(40)
numbers = tuple(data)
print(numbers)
```

Output:

TEXT

Copy

```
(10, 20, 30, 40)
```

Common Fresher Mistakes

X Mistake	Problem	✓ Correct Understanding
<code>(10)</code> considered a tuple	It creates an <code>int</code>	Use <code>(10,)</code>
Trying <code>tuple.append()</code>	Tuples are immutable	Use a list for changeable data
Trying <code>data[0] = 100</code>	Item assignment is not supported	Tuple elements cannot be replaced
Expecting <code>sorted(tuple)</code> to return tuple	<code>sorted()</code> returns a list	Convert using <code>tuple(sorted(data))</code>
Mismatched unpacking	Number of names and values differ	Names should match values
Thinking all objects inside a tuple are immutable	Tuple may contain mutable objects	Inner mutable objects can change

Practical Example — Student Record

PYTHON

Copy

```
student = ("Arun", 21, 8.5)
name, age, cgpa = student
print("Name:", name)
print("Age:", age)
print("CGPA:", cgpa)
```

Output:

TEXT

Copy

```
Name: Arun
Age: 21
CGPA: 8.5
```

A tuple is suitable here when the record values are treated as fixed data.

Placement Quick Points

- A tuple stores multiple values.
- Tuples are ordered and immutable.
- Tuples allow duplicate values.
- Tuples can store different data types.
- Tuple indexing starts from `0`.
- A single-element tuple requires a comma.
- Tuples have `count()` and `index()` methods.
- Tuple packing stores multiple values together.
- Tuple unpacking assigns values to multiple names.
- `sorted(tuple)` returns a list.
- A tuple can contain mutable objects.
- Tuple immutability prevents replacing its elements.

Interview Questions

Q1. What is a tuple in Python?

A tuple is an ordered and immutable collection of elements.

Q2. What is the main difference between a list and a tuple?

A list is mutable, while a tuple is immutable.

Q3. How do you create a single-element tuple?

```
data = (10,)
```

Q4. Why is `(10)` not a tuple?

Parentheses alone do not create a single-element tuple. The comma is required.

Q5. What methods are available for tuples?

```
count() and index()
```

Q6. Can a tuple contain duplicate values?

Yes.

Q7. Can a tuple contain different data types?

Yes.

Q8. What is tuple packing?

Storing multiple values together as a tuple.

Q9. What is tuple unpacking?

Assigning tuple values to multiple names.

Q10. Can a tuple contain a list?

Yes.

Q11. Can the list inside a tuple be modified?

Yes, because the list itself is mutable.

Q12. What does `sorted()` return when used with a tuple?

A list.

Practice

1. Create a tuple containing five numbers.
2. Create a single-element tuple.
3. Access the first and last elements of a tuple.
4. Count how many times `10` appears.
5. Find the index of `20`.
6. Check whether `"Python"` exists in a tuple.
7. Find the maximum, minimum, and sum of a numeric tuple.
8. Pack a student's name, age, and CGPA into a tuple.
9. Unpack the student tuple into three variables.
10. Convert a list into a tuple.
11. Convert a tuple into a list.
12. Predict the output:

PYTHON

Copy

```
data = (10)
print(type(data))
```

1. Find the error:

PYTHON

Copy

```
numbers = (10, 20, 30)
numbers.append(40)
```

1. Explain the output:

PYTHON

Copy

```
data = (10, [20, 30])
data[1].append(40)
print(data)
```

Indexing

Indexing means accessing an element using its **position number**, called an **index**.

PYTHON

Copy

```
numbers = [10, 20, 30, 40, 50]
```

Python supports two types of indexing:

- **Positive Indexing** — starts from **0**
- **Negative Indexing** — starts from **-1**

Positive and Negative Indexing

Element	10	20	30	40	50
Positive Index	0	1	2	3	4
Negative Index	-5	-4	-3	-2	-1

Positive Indexing

Positive indexing starts from **0** and moves from **left to right**.

PYTHON

Copy

```
numbers = [10, 20, 30, 40, 50]
print(numbers[0]) # 10
print(numbers[2]) # 30
print(numbers[4]) # 50
```

Important

The first element is at index **0**, not **1**.

PYTHON

Copy

```
numbers = [10, 20, 30, 40, 50]
numbers[0] # First element
numbers[1] # Second element
```

Negative Indexing

Negative indexing starts from `-1` and accesses elements from **right to left**.

PYTHON

Copy

```
numbers = [10, 20, 30, 40, 50]
print(numbers[-1]) # 50
print(numbers[-2]) # 40
print(numbers[-5]) # 10
```

`-1` always refers to the **last element**.

Positive vs Negative Indexing

Index Type	Starts From	Direction	Example
Positive	<code>0</code>	Left → Right	<code>numbers[0]</code> → <code>10</code>
Negative	<code>-1</code>	Right → Left	<code>numbers[-1]</code> → <code>50</code>

The same element can be accessed using either a positive or negative index.

PYTHON

Copy

```
numbers = [10, 20, 30, 40, 50]
print(numbers[0]) # 10
print(numbers[-5]) # 10
print(numbers[4]) # 50
print(numbers[-1]) # 50
```

Index Range

For a list containing 5 elements:

PYTHON

Copy

```
numbers = [10, 20, 30, 40, 50]
```

Index Type	Valid Range
Positive Index	0 to 4
Negative Index	-1 to -5

General Rule

For a collection with n elements:

- Positive index: 0 to n - 1
- Negative index: -1 to -n

IndexError

Accessing an index outside the valid range causes an IndexError.

PYTHON

Copy

```
numbers = [10, 20, 30, 40, 50]
print(numbers[5]) # IndexError
print(numbers[-6]) # IndexError
```

Valid indexes are:

Output:

TEXT

Copy

Positive: 0, 1, 2, 3, 4
Negative: -1, -2, -3, -4, -5

Indexing Different Data Types

Indexing is not limited to lists. It can also be used with other ordered sequence types.

String

PYTHON

Copy

```
name = "Python"  
print(name[0])    # P  
print(name[-1])   # n
```

List

PYTHON

Copy

```
skills = ["Python", "SQL", "Java"]  
print(skills[0])    # Python  
print(skills[-1])   # Java
```

Tuple

PYTHON

Copy

```
numbers = (10, 20, 30)  
print(numbers[1])    # 20  
print(numbers[-1])   # 30
```

Common Fresher Mistakes

X Mistake	Problem	✓ Correct
<code>numbers[1]</code> expecting first element	Index starts from <code>0</code>	<code>numbers[0]</code>
<code>numbers[len(numbers)]</code>	Index is out of range	<code>numbers[len(numbers) - 1]</code>
<code>numbers[-6]</code> for a 5-item list	Negative index is out of range	Minimum is <code>-5</code>
Thinking <code>-1</code> is the first element	<code>-1</code> means last element	Use <code>0</code> for first

Useful Access Patterns

PYTHON

Copy

```

numbers = [10, 20, 30, 40, 50]
print(numbers[0])    # First element
print(numbers[-1])   # Last element
print(numbers[1])    # Second element
print(numbers[-2])   # Second last element

```

Placement Quick Points

- Indexing is used to access an element by its position.
- Positive indexing starts from `0`.
- Negative indexing starts from `-1`.
- `0` represents the first element.
- `-1` represents the last element.
- For `n` elements, the last positive index is `n - 1`.

- For `n` elements, the first negative index is `-n`.
- An invalid index raises `IndexError`.
- Strings, lists, and tuples support indexing.

Interview Questions

Q1. What is indexing in Python?

Indexing is accessing an element using its position number.

Q2. What is the index of the first element?

`0`.

Q3. What is the index of the last element?

`-1` or `len(collection) - 1`.

Q4. What happens when an invalid index is accessed?

Python raises an `IndexError`.

Q5. What is the valid positive index range for 5 elements?

`0` to `4`.

Q6. What is the valid negative index range for 5 elements?

`-1` to `-5`.

Q7. Do dictionaries support positional indexing?

No. Dictionaries are accessed using keys, not positional indexes.

Practice

Use:

PYTHON

Copy

```
numbers = [10, 20, 30, 40, 50]
```

1. Access the first element.
2. Access the last element using negative indexing.
3. Access `30` using both positive and negative indexing.
4. Access the second-last element.
5. What happens with `numbers[5]`?
6. What happens with `numbers[-6]`?
7. Find the output:

PYTHON

Copy

```
numbers = [10, 20, 30, 40, 50]  
print(numbers[-len(numbers)])
```

Slicing

Slicing means accessing a **portion** of a sequence, instead of a single element.

PYTHON

Copy

```
numbers = [10, 20, 30, 40, 50]  
numbers = [10, 20, 30, 40, 50]  
numbers = [10, 20, 30, 40, 50]  
numbers = [10, 20, 30, 40, 50]  
print(numbers[1:4]) # [20, 30, 40]
```

Slicing works on any ordered sequence, such as **strings** and **lists**.

Basic Syntax

PYTHON

Copy

```
sequence[start:stop]
```


- `start` — the index where the slice begins (**included**)
- `stop` — the index where the slice ends (**excluded**)

The element at `stop` is **not** part of the result.

PYTHON

Copy

```
numbers = [10, 20, 30, 40, 50]
print(numbers[1:4]) # [20, 30, 40]
```

Here the slice starts at index `1` (`20`) and stops **before** index `4` (`50`).

Omitting start or stop

If `start` is omitted, the slice begins at the **start** of the sequence. If `stop` is omitted, the slice runs to the **end** of the sequence.

PYTHON

Copy

```
numbers = [10, 20, 30, 40, 50]
print(numbers[:3]) # [10, 20, 30]
print(numbers[2:]) # [30, 40, 50]
print(numbers[:]) # [10, 20, 30, 40, 50]
```

`numbers[:]` returns **all** the elements.

Slicing Strings

Strings support slicing in the same way.

PYTHON

Copy

```
name = "Python"
print(name[0:3]) # Pyt
print(name[3:]) # hon
print(name[:2]) # Py
```

A common use is skipping the first character:

PYTHON

Copy

```
student_name = "Aarav"  
print(student_name[1:]) # arav
```

Negative Indices in Slicing

Slicing also works with **negative indices**, counting from the **right**.

PYTHON

Copy

```
numbers = [10, 20, 30, 40, 50]  
print(numbers[-3:]) # [30, 40, 50]  
print(numbers[:-2]) # [10, 20, 30]
```

PYTHON

Copy

```
name = "Python"  
print(name[-3:]) # hon
```

The step Value

Slicing can take a third value, called the **step**.

PYTHON

Copy

```
sequence[start:stop:step]
```

The **step** decides how many positions to move for each element.

PYTHON

Copy

```
numbers = [10, 20, 30, 40, 50]  
print(numbers[0:5:2]) # [10, 30, 50]
```

Here the slice takes every **second** element.

Reversing with Slicing

A `step` of `-1` reverses the sequence.

PYTHON

Copy

```
sequence[::-1]
```

PYTHON

Copy

```
numbers = [10, 20, 30, 40, 50]
print(numbers[::-1]) # [50, 40, 30, 20, 10]

name = "Python"
print(name[::-1])    # nohtyP
```

Slicing vs Indexing

Feature	Indexing	Slicing
Access	A single element	A range of elements
Syntax	<code>sequence[i]</code>	<code>sequence[start:stop:step]</code>
Returns	One item	A new sequence (same type)
Example	<code>numbers[1]</code> → <code>20</code>	<code>numbers[1:3]</code> → <code>[20, 30]</code>

Common Fresher Mistakes

X Mistake	Problem	✓ Correct
<code>numbers[1:4]</code> expecting <code>50</code> too	<code>stop</code> is excluded	Use <code>numbers[1:5]</code>
<code>numbers[4:1]</code> expecting a result	<code>start</code> is after <code>stop</code> , so empty	Use <code>numbers[1:4]</code>
Thinking slicing changes the original	Slicing returns a new sequence	Original stays unchanged

Placement Quick Points

- Slicing accesses a **range** of elements.
- Syntax is `sequence[start:stop:step]`.
- `start` is included, `stop` is excluded.
- Omitting `start` begins from the first element.
- Omitting `stop` runs to the last element.
- Negative indices count from the right.
- `sequence[::-1]` reverses the sequence.
- Slicing returns a **new** sequence and leaves the original unchanged.
- Strings and lists both support slicing.

Interview Questions

Q1. What is slicing in Python?

Slicing is accessing a portion (range) of a sequence using `start:stop:step`.

Q2. Is the `stop` index included in a slice?

No. The `stop` index is excluded.

Q3. What does `sequence[::-1]` do?

It reverses the sequence.

Q4. What happens if you omit both `start` and `stop`?

The whole sequence is returned.

Q5. Does slicing modify the original sequence?

No. It returns a new sequence and leaves the original unchanged.

Practice

Use:

PYTHON

Copy

```
numbers = [10, 20, 30, 40, 50]
name = "Python"
```

1. Get the first three elements of `numbers`.
2. Get the last two elements of `numbers`.
3. Get every second element of `numbers`.
4. Reverse `numbers` using slicing.
5. Get `"hon"` from `name`.
6. Reverse `name` using slicing.
7. Find the output:

PYTHON

Copy

```
print(numbers[1:4])
```

Sets

What is a Set?

A **set** is a collection used to store **unique values**.

PYTHON

Copy

```
student_skills = {"Python", "SQL", "Java"}
```

Each value appears only once.

PYTHON

Copy

```
student_skills = {"Python", "SQL", "Python", "Java"}  
print(student_skills)
```

Possible output contains only unique values:

TEXT

Copy

```
{'Java', 'Python', 'SQL'}
```

The displayed order may vary because sets are unordered.

Why Do We Use Sets?

Consider skills collected from a student form:

PYTHON

Copy

```
skills = ["Python", "SQL", "Python", "Java", "SQL"]
```

The list contains duplicate values.

Using a set:

PYTHON

Copy

```
skills = ["Python", "SQL", "Python", "Java", "SQL"]
student_skills = set(skills)
print(student_skills)
```

The duplicate skills are removed.

Output:

TEXT

Copy

```
{'Java', 'Python', 'SQL'}
```

Sets are useful when we need **unique data and set comparisons**.

Real-Time Usage

Sets are commonly used for:

- Removing duplicate values
- Comparing user skills
- Finding common skills
- Finding missing skills
- Combining unique data
- Unique user IDs
- Tags and categories
- Permission comparison

Example:

PYTHON

Copy

```
student_skills = {"Python", "SQL", "Java"}  
print(required_skills = {"Python", "SQL", "Git"})  
print(required_skills)  
print(required_skills)
```

We can easily find:

- Common skills
- Missing skills
- Combined skills

Key Features of Sets

Feature	Meaning
Unique Values	Duplicate values are removed
Unordered	No positional order
Mutable	Elements can be added or removed
No Indexing	Cannot access values using an index
Set Operations	Supports union, intersection, and difference

Creating a Set

We will use student skills throughout this topic.

PYTHON

Copy

```
student_skills = {"Python", "SQL", "Java"}
```

Set with Duplicate Values

PYTHON

Copy

```
student_skills = {"Python", "SQL", "Python", "Java"}
```

Duplicate values are automatically removed.

Creating an Empty Set

Use `set()`.

PYTHON

Copy

```
student_skills = set()
```

Do not use `{}`.

PYTHON

Copy

```
student_skills = {}  
print(type(student_skills))
```

Output:

TEXT

Copy

```
<class 'dict'>
```

`{}` creates an empty dictionary.

PYTHON

Copy

```
student_skills = set()  
print(type(student_skills))
```

Output:

TEXT

Copy

```
<class 'set'>
```

Accessing Set Elements

Sets do not support indexing.

PYTHON

Copy

```
student_skills = {"Python", "SQL", "Java"}  
student_skills = {"Python", "SQL", "Java"}  
student_skills = {"Python", "SQL", "Java"}  
student_skills = ["Python", "SQL", "Git"]  
print(student_skills[0])
```

Error:

Output:

TEXT

Copy

Python

Because sets are unordered, there is no first or last element by position.

To process set values, use a loop.

PYTHON

Copy

```
student_skills = {"Python", "SQL", "Java"}  
for skill in student_skills:  
    print(skill)
```

The iteration order is not guaranteed.

Checking Whether a Value Exists

Use `in` and `not in`.

PYTHON

Copy

```
student_skills = {"Python", "SQL", "Java"}  
print("Python" in student_skills) # True  
print("Git" in student_skills)    # False
```

PYTHON

Copy

```
student_skills = {"Python", "SQL", "Java"}  
print("Git" not in student_skills) # True
```

Membership checking is one of the most common uses of sets.

Adding Elements

`add()`

Adds one element to the set.

PYTHON

Copy

```
student_skills = {"Python", "SQL", "Java"}  
student_skills.add("Git")  
print(student_skills)
```

Now `"Git"` is added to the set.

Adding a Duplicate Value

PYTHON

Copy

```
student_skills = {"Python", "SQL", "Java"}  
student_skills.add("Python")
```

No duplicate is created. The set still contains `"Python"` only once.

Adding Multiple Elements

update()

Adds multiple elements from an iterable.

PYTHON

Copy

```
student_skills = {"Python", "SQL"}
student_skills.update(["Java", "Git"])
print(student_skills)
```

update() can accept values from lists, tuples, sets, and other iterables.

add() VS update()

Method	Adds	Example
add()	One element	skills.add("Git")
update()	Elements from an iterable	skills.update(["Git", "Java"])

Placement Favorite: add() vs update().

Removing Elements

Let's use:

PYTHON

Copy

```
student_skills = {"Python", "SQL", "Java"}
```

remove()

Removes a specific value.

PYTHON

Copy

```
student_skills = {"Python", "SQL", "Java"}
student_skills.remove("Java")
```

If the value does not exist:

PYTHON

Copy

```
student_skills = {"Python", "SQL", "Java"}
student_skills.remove("Git")
```

Python raises a `KeyError`.

`discard()`

Removes a value if it exists.

PYTHON

Copy

```
student_skills = {"Python", "SQL", "Java"}
student_skills.discard("Java")
```

If the value does not exist:

PYTHON

Copy

```
student_skills = {"Python", "SQL", "Java"}
student_skills.discard("Git")
```

No error occurs.

`remove()` VS `discard()`

Method	Value Exists	Value Missing
<code>remove()</code>	Removes value	Raises <code>KeyError</code>
<code>discard()</code>	Removes value	No error

Placement Favorite: Difference between `remove()` and `discard()`.

`pop()`

Removes and returns an arbitrary element.

PYTHON

Copy

```
student_skills = {"Python", "SQL", "Java"}  
removed_skill = student_skills.pop()  
print(removed_skill)
```

Because a set is unordered, do not assume which element `pop()` will remove.

`clear()`

Removes all elements.

PYTHON

Copy

```
student_skills = {"Python", "SQL", "Java"}  
student_skills.clear()  
print(student_skills)
```

Output:

TEXT

Copy

```
set()
```

Comparing Student Skills

Now let's use a real placement example.

The student has:

Output:

TEXT

Copy

Python, SQL, Java

The company requires:

TEXT

Copy

Python, SQL, Git

We can compare these sets using set operations.

Union

Union combines all unique values from both sets.

Use `|` or `union()`.

PYTHON

Copy

```
required_skills = {"Python", "SQL", "Git"}
student_skills = {"Python", "SQL", "Java"}
all_skills = student_skills | required_skills
print(all_skills)
```

Or:

PYTHON

Copy

```
required_skills = {"Python", "SQL", "Git"}
student_skills = {"Python", "SQL", "Java"}
all_skills = student_skills.union(required_skills)
print(all_skills)
```

Result contains:

TEXT

Copy

```
{'Java', 'Git', 'Python', 'SQL'}
```

Meaning

Student skills OR required skills — all unique skills.

Intersection

Intersection finds values common to both sets.

Use `&` or `intersection()`.

PYTHON

Copy

```
required_skills = {"Python", "SQL", "Git"}  
student_skills = {"Python", "SQL", "Java"}  
common_skills = student_skills & required_skills  
print(common_skills)
```

Or:

PYTHON

Copy

```
required_skills = {"Python", "SQL", "Git"}  
student_skills = {"Python", "SQL", "Java"}  
common_skills = student_skills.intersection(required_skills)  
print(common_skills)
```

Result:

Output:

TEXT

Copy

```
{'Python', 'SQL'}
```

Meaning

The student already has these required skills.

Difference

Difference finds values available in one set but not the other.

Use `-` or `difference()`.

Find Missing Skills

PYTHON

Copy

```
student_skills = {"Python", "SQL", "Java"}
required_skills = {"Python", "SQL", "Git"}
missing_skills = required_skills - student_skills
print(missing_skills)
```

Result:

Output:

TEXT

Copy

```
{'Git'}
```

This is a very practical use of sets.

TEXT

Copy

```
Required Skills - Student Skills = Missing Skills
```

Find Extra Skills

PYTHON

Copy

```
required_skills = {"Python", "SQL", "Git"}
student_skills = {"Python", "SQL", "Java"}
extra_skills = student_skills - required_skills
print(extra_skills)
```

Result:

Output:

TEXT

Copy

```
{'Java'}
```

Symmetric Difference

Symmetric difference finds values that exist in **only one of the sets**, but not both.

Use `^` or `symmetric_difference()`.

PYTHON

Copy

```
required_skills = {"Python", "SQL", "Git"}  
student_skills = {"Python", "SQL", "Java"}  
different_skills = student_skills ^ required_skills  
print(different_skills)
```

Result contains:

Output:

TEXT

Copy

```
{'Java', 'Git'}
```

`Python` and `SQL` are not included because they are common to both sets.

Set Operations Comparison

Using:

Operation	Symbol	Meaning	Result
Union	$\cup$	All unique skills	Python, SQL, Java, Git
Intersection	$\&$	Common skills	Python, SQL
Difference	$-$	Skills in first set only	Java
Symmetric Difference	$\wedge$	Skills not common	Java, Git

Visual Understanding

Output:

TEXT

Copy

```
Student Skills = Python, SQL, Java
Required Skills = Python, SQL, Git
Union          → Python, SQL, Java, Git
Intersection   → Python, SQL
Student - Required → Java
Required - Student → Git
Symmetric Difference → Java, Git
```

Subset

A set is a **subset** if all its elements exist in another set.

PYTHON

Copy

```
student_skills = {"Python", "SQL", "Java"}
basic_skills = {"Python", "SQL"}
print(basic_skills.issubset(student_skills))
```

Output:

TEXT

Copy

True

All `basic_skills` exist in `student_skills`.

Using the operator:

PYTHON

Copy

```
student_skills = {"Python", "SQL", "Java"}  
basic_skills = {"Python", "SQL"}  
print(basic_skills <= student_skills)
```

Superset

A set is a **superset** if it contains all elements of another set.

PYTHON

Copy

```
basic_skills = {"Python", "SQL"}  
student_skills = {"Python", "SQL", "Java"}  
print(student_skills.issuperset(basic_skills))
```

Output:

TEXT

Copy

True

Using the operator:

PYTHON

Copy

```
basic_skills = {"Python", "SQL"}  
student_skills = {"Python", "SQL", "Java"}  
print(student_skills >= basic_skills)
```

Subset vs Superset

Concept	Meaning
Subset	All my elements exist in another set
Superset	I contain all elements of another set

Output:

TEXT

Copy

True

Disjoint Sets

Two sets are disjoint if they have **no common elements**.

PYTHON

Copy

```
student_skills = {"Python", "SQL"}  
design_skills = {"Figma", "Photoshop"}  
print(student_skills.isdisjoint(design_skills))
```

Output:

TEXT

Copy

True

There are no common skills.

Set Copying

Consider:

PYTHON

Copy

```
student_skills = {"Python", "SQL"}  
new_skills = student_skills
```

This does not create an independent set.

PYTHON

Copy

```
student_skills = {"Python", "SQL"}  
new_skills = student_skills  
new_skills.add("Java")  
print(student_skills)
```

The original set also changes.

Use `copy()`.

PYTHON

Copy

```
student_skills = {"Python", "SQL"}  
new_skills = student_skills.copy()  
new_skills.add("Java")  
print(student_skills)  
print(new_skills)
```

The sets are now separate.

Common Set Methods

Method	Purpose
<code>add()</code>	Add one element
<code>update()</code>	Add elements from an iterable
<code>remove()</code>	Remove value; error if missing
<code>discard()</code>	Remove value safely
<code>pop()</code>	Remove an arbitrary element
<code>clear()</code>	Remove all elements
<code>union()</code>	Combine unique values
<code>intersection()</code>	Find common values
<code>difference()</code>	Find values in the first set only
<code>symmetric_difference()</code>	Find non-common values
<code>issubset()</code>	Check subset
<code>issuperset()</code>	Check superset
<code>isdisjoint()</code>	Check for no common values
<code>copy()</code>	Create a shallow copy

Common Fresher Mistakes

X Mistake	Problem	✓ Correct Understanding
<code>{}</code> used for empty set	Creates a dictionary	Use <code>set()</code>
<code>skills[0]</code>	Sets do not support indexing	Use membership checks or iteration
Expecting set output order	Sets are unordered	Do not rely on display order
<code>remove()</code> on a missing value	Raises <code>KeyError</code>	Use <code>discard()</code> when absence is acceptable
Expecting duplicates	Sets store unique values	Duplicate values are removed
Thinking <code>pop()</code> removes the last item	Set has no positional last item	<code>pop()</code> removes an arbitrary element
<code>b = a</code> expecting a copy	Both refer to the same set	Use <code>a.copy()</code>

Practical Example — Placement Skill Gap Checker

PYTHON

Copy

```
student_skills = {"Python", "SQL", "Java"}
required_skills = {"Python", "SQL", "Git"}
common_skills = student_skills & required_skills
missing_skills = required_skills - student_skills
extra_skills = student_skills - required_skills
print("Matched Skills:", common_skills)
print("Missing Skills:", missing_skills)
print("Extra Skills:", extra_skills)
```

Possible output:

TEXT

Copy

```
Matched Skills: {'Python', 'SQL'}  
Missing Skills: {'Git'}  
Extra Skills: {'Java'}
```

This same logic can be used in:

- Resume matching
- Job skill matching
- Permission comparison
- Tag comparison
- Recommendation systems
- Data deduplication

Placement Quick Points

- A set stores unique values.
- Sets are unordered.
- Sets are mutable.
- Sets do not support indexing.
- Use `set()` to create an empty set.
- `{}` creates an empty dictionary.
- `add()` adds one element.
- `update()` adds elements from an iterable.
- `remove()` raises `KeyError` if the value is missing.
- `discard()` does not raise an error for a missing value.
- Union combines unique values.
- Intersection finds common values.
- Difference finds values present only in the first set.
- Symmetric difference finds non-common values.
- `in` is commonly used for membership checking.

- `pop()` removes an arbitrary element.

Interview Questions

Q1. What is a set in Python?

A set is an unordered collection of unique elements.

Q2. Does a set allow duplicate values?

No.

Q3. Does a set support indexing?

No.

Q4. How do you create an empty set?

```
data = set()
```

Q5. What does `{}` create?

An empty dictionary.

Q6. What is the difference between `add()` and `update()`?

`add()` adds one element. `update()` adds elements from an iterable.

Q7. What is the difference between `remove()` and `discard()`?

`remove()` raises `KeyError` if the value is missing. `discard()` does not.

Q8. What is union?

It combines all unique elements from both sets.

Q9. What is intersection?

It returns common elements.

Q10.What is difference?

It returns elements present in the first set but not the second.

Q11.What is symmetric difference?

It returns elements that are not common to both sets.

Q12.What does `pop()` remove from a set?

An arbitrary element.

Q13.What is a subset?

A set whose every element exists in another set.

Q14.What are disjoint sets?

Sets with no common elements.

Practice

Use:

PYTHON

Copy

```
student_skills = {"Python", "SQL", "Java"}
required_skills = {"Python", "SQL", "Git"}
```

1. Add `"Git"` to `student_skills`.
2. Add `"HTML"` and `"CSS"` using `update()`.
3. Check whether `"Python"` exists.
4. Remove `"Java"` using `remove()`.
5. Safely remove `"C++"` using `discard()`.
6. Find all unique skills from both sets.
7. Find common skills.
8. Find the student's missing skills.
9. Find the student's extra skills.
10. Find the symmetric difference.
11. Check whether `{"Python", "SQL"}` is a subset of `student_skills`.

12. Create an empty set.
13. Explain why `{}` is not an empty set.
14. Find the error:

PYTHON

Copy

```
print(student_skills[0])
```

1. Explain the difference:

PYTHON

Copy

```
student_skills = {"Python", "SQL", "Java"}  
student_skills.remove("C++")  
student_skills.discard("C++")
```

Dictionaries

What is a Dictionary?

A dictionary is a collection used to store data as **key-value pairs**.

PYTHON

Copy

```
student = {  
    "name": "Arun",  
    "age": 21,  
    "department": "CSE",  
    "cgpa": 8.5,  
    "is_placed": False  
}  
print()
```

Here:

- `"name"` is a **key**
- `"Arun"` is its **value**

Output:

TEXT

Copy

```
"name" → "Arun"  
"age" → 21  
"cgpa" → 8.5
```

Why Do We Use Dictionaries?

Consider a student profile stored using a list:

PYTHON

Copy

```
student = ["Arun", 21, "CSE", 8.5, False]
```

What does `8.5` represent? We need to remember its position.

Using a dictionary:

PYTHON

Copy

```
student = {  
    "name": "Arun",  
    "age": 21,  
    "department": "CSE",  
    "cgpa": 8.5,  
    "is_placed": False  
}
```

Now the data has a clear meaning.

PYTHON

Copy

```
student = {  
    student["cgpa"] # 8.5
```

Dictionaries make structured data easy to understand and access.

Real-Time Usage

Examples:

- User profiles
- Student records
- API responses
- JSON-like data
- Product details
- Application configuration
- Database records
- Model results

Example API-style user data:

PYTHON

Copy

```
user = {  
    "id": 101,  
    "name": "Arun",  
    "email": "arun@gmail.com",  
    "is_active": True  
}
```

Key Features of Dictionaries

Feature	Meaning
Key-Value Pairs	Stores data as <code>key: value</code>
Ordered	Maintains insertion order
Mutable	Values can be changed
Unique Keys	Duplicate keys are not allowed as separate entries
Dynamic	Items can be added or removed
Key-Based Access	Values are accessed using keys

Creating a Dictionary

We will use the same `student` dictionary throughout this topic.

PYTHON

Copy

```
student = {  
    "name": "Arun",  
    "age": 21,  
    "department": "CSE",  
    "cgpa": 8.5,  
    "is_placed": False  
}
```

Empty Dictionary

PYTHON

Copy

```
student = {}
```

Accessing Dictionary Values

Dictionary values are accessed using **keys**.

PYTHON

Copy

```
student = {}  
print(student)  # {}  
student = {}  
student = {"name": "Arun", "cgpa": 8.5}  
student["name"]  
print(student["department"])  
print(student["cgpa"])
```

Output:

Unlike lists, dictionaries do not use positional indexes for normal access.

PYTHON

Copy

```
student = {}  
student[0] # KeyError
```

We access values using keys.

PYTHON

Copy

```
student["name"]
```

Accessing Values Using `get()`

The `get()` method is another way to access a value.

PYTHON

Copy

```
student = {}  
print(student.get("name"))
```

Output:

TEXT

Copy

```
None
```

`[]` VS `get()`

What happens when a key does not exist?

Using `[]`:

PYTHON

Copy

```
student = {}  
print(student["email"])
```


Error:

Output:

TEXT

Copy

KeyError: 'email'

Using `get()`:

PYTHON

Copy

```
student = {}  
print(student.get("email"))
```

Output:

TEXT

Copy

None

We can also provide a default value.

PYTHON

Copy

```
student = {}  
print(student.get("email", "Not Available"))
```

Output:

TEXT

Copy

Not Available

[] vs get() Comparison

Access Method	Missing Key
<code>student["email"]</code>	Raises <code>KeyError</code>
<code>student.get("email")</code>	Returns <code>None</code>
<code>student.get("email", "Not Available")</code>	Returns default value

Placement Favorite: Difference between `[ ]` and `get()`.

Updating a Value

Dictionaries are **mutable**.

We can change an existing value.

PYTHON

Copy

```
student = {}
student = {}
student = {}
student["cgpa"] = 9.0
print(student["cgpa"])
```

Output:

TEXT

Copy

```
9.0
```

Let's update the placement status.

PYTHON

Copy

```
student = {}
student["is_placed"] = True
print(student["is_placed"])
```

Output:

TEXT

Copy

True

Adding a New Key-Value Pair

Assign a value to a new key.

PYTHON

Copy

```
student = {}  
student = {}  
student = {}  
student["email"] = "arun@gmail.com"
```

Now the dictionary contains:

PYTHON

Copy

```
{  
    "name": "Arun",  
    "age": 21,  
    "department": "CSE",  
    "cgpa": 9.0,  
    "is_placed": True,  
    "email": "arun@gmail.com"  
}
```

Important Rule

PYTHON

Copy

```
student["cgpa"] = 9.0
```

Existing key → updates the value

PYTHON

Copy

```
student["email"] = "arun@gmail.com"
```

New key → adds a new key-value pair

Updating Multiple Values Using `update()`

The `update()` method can add or update multiple values.

PYTHON

Copy

```
student = {}
student.update({
    "cgpa": 9.2,
    "is_placed": True
})
```

Now:

PYTHON

Copy

```
student = {}
print(student["cgpa"])      # 9.2
print(student["is_placed"]) # True
```

Adding vs Updating

Operation	Example	Result
Existing key	<code>student["cgpa"] = 9.0</code>	Updates value
New key	<code>student["email"] = "arun@gmail.com"</code>	Adds new item
<code>update()</code>	<code>student.update({...})</code>	Adds or updates items

Removing Dictionary Items

`pop()`

Removes a key and returns its value.

PYTHON

Copy

```
student = {}  
removed = student.pop("age")  
print(removed)
```

Output:

TEXT

Copy

```
21
```

The `"age"` key is removed from the dictionary.

`popitem()`

Removes and returns the last inserted key-value pair.

PYTHON

Copy

```
student = {}  
item = student.popitem()  
print(item)
```

`del`

Deletes a specific key-value pair.

PYTHON

Copy

```
student = {}  
del student["department"]
```

`clear()`

Removes all items.

Output:

TEXT

Copy

```
{}
```

Dictionary Removal Methods

Method	Purpose
<code>pop(key)</code>	Removes a specific key and returns its value
<code>popitem()</code>	Removes the last inserted item
<code>del</code>	Deletes a specific key
<code>clear()</code>	Removes all items

Getting Keys, Values, and Items

Let's use our student dictionary:

PYTHON

Copy

```
student = {  
    "name": "Arun",  
    "age": 21,  
    "department": "CSE",  
    "cgpa": 8.5,  
    "is_placed": False  
}
```

`keys()`

Returns all keys.

PYTHON

Copy

```
student = {  
print(student.keys())
```

Output:

TEXT

Copy

```
dict_keys(['name', 'age', 'department', 'cgpa', 'is_placed'])
```

values()

Returns all values.

PYTHON

Copy

```
student = {  
print(student.values())
```

Output:

TEXT

Copy

```
dict_values(['Arun', 21, 'CSE', 8.5, False])
```

items()

Returns key-value pairs.

PYTHON

Copy

```
student = {  
print(student.items())
```

Each item is represented as a (key, value) pair.

`keys()` VS `values()` VS `items()`

Method	Returns
<code>keys()</code>	Keys
<code>values()</code>	Values
<code>items()</code>	Key-value pairs

Checking if a Key Exists

Use the `in` operator.

PYTHON

Copy

```
student = {
print("name" in student) # True
print("email" in student) # False
```

By default, `in` checks dictionary keys.

PYTHON

Copy

```
student = {
"Arun" in student # False
```

To check values:

PYTHON

Copy

```
student = {
"Arun" in student.values() # True
```

Looping Through a Dictionary

Loop Through Keys

PYTHON

Copy

```
student = {  
for key in student:  
    print(key)
```

Loop Through Values

PYTHON

Copy

```
student = {  
for value in student.values():  
    print(value)
```

Loop Through Key-Value Pairs

PYTHON

Copy

```
student = {  
for key, value in student.items():  
    print(key, ":", value)
```

Output:

TEXT

Copy

```
name : Arun  
age : 21  
department : CSE  
cgpa : 8.5  
is_placed : False
```

Recommended: Use `items()` when both key and value are required.

Duplicate Keys

Dictionary keys must be unique.

PYTHON

Copy

```
student = {  
    "name": "Arun",  
    "name": "Kumar"  
}  
print(student)
```

Output:

TEXT

Copy

```
{'name': 'Kumar'}
```

The latest value replaces the previous value for the same key.

Dictionary Keys

Dictionary keys must be **hashable**.

Common valid key types:

PYTHON

Copy

```
data = {  
    "name": "Arun",  
    1: "First",  
    (10, 20): "Coordinates"  
}
```

Common invalid key:

PYTHON

Copy

```
data = {  
    [1, 2]: "Numbers"  
}
```

This raises a `TypeError` because a list cannot be used as a dictionary key.

Beginner Rule

Use immutable, hashable values such as:

- `str`
- `int`
- `float`
- suitable `tuple` values

as dictionary keys.

Nested Dictionary

Real applications often contain nested dictionaries.

Let's extend our student example.

PYTHON

Copy

```
student = {  
    "name": "Arun",  
    "age": 21,  
    "department": "CSE",  
    "cgpa": 8.5,  
    "is_placed": False,  
    "address": {  
        "city": "Chennai",  
        "state": "Tamil Nadu"  
    }  
}
```

Access the city:

PYTHON

Copy

```
student = {  
print(student = {  
student = {  
student = {  
student = {"name": "Arun", "cgpa": 8.5}  
student = {"name": "Arun", "cgpa": 8.5}  
student["address"]["city"])
```

Output:

TEXT

Copy

Chennai

How It Works

PYTHON

Copy

```
student["address"]
```

Returns:

PYTHON

Copy

```
{  
    "city": "Chennai",  
    "state": "Tamil Nadu"  
}
```

Then:

PYTHON

Copy

```
student["address"]["city"]
```

Returns:

Output:

TEXT

Copy

Chennai

Dictionary Copying

Consider:

PYTHON

Copy

```
student = {  
    student_copy = student
```

This does **not** create an independent dictionary. Both names refer to the same dictionary.

PYTHON

Copy

```
student_copy = student  
student_copy["cgpa"] = 9.5  
print(student["cgpa"])
```

Output:

TEXT

Copy

```
9.5
```

Use `copy()`:

PYTHON

Copy

```
student = {  
    student_copy = student.copy()
```

Now the outer dictionary is copied separately.

*Shallow copy and deep copy are covered separately in the **Copying Objects** topic.*

Common Dictionary Methods

Method	Purpose
<code>get()</code>	Safely access a value
<code>keys()</code>	Get all keys
<code>values()</code>	Get all values
<code>items()</code>	Get key-value pairs
<code>update()</code>	Add or update items
<code>pop()</code>	Remove a specific key
<code>popitem()</code>	Remove the last inserted item
<code>copy()</code>	Create a shallow copy
<code>clear()</code>	Remove all items

Common Fresher Mistakes

X Mistake	Problem	✓ Correct Understanding
<code>student[0]</code>	Dictionary is not accessed by position	Use <code>student["name"]</code>
<code>student["email"]</code> for a missing key	Raises <code>KeyError</code>	Use <code>student.get("email")</code>
Thinking duplicate keys are stored separately	Keys must be unique	Latest value replaces old value
<code>"Arun" in student</code> expecting <code>True</code>	<code>in</code> checks keys	Use <code>"Arun" in student.values()</code>
<code>copy = student</code> expecting a copy	Both refer to the same dictionary	Use <code>student.copy()</code>
Using a list as a key	Lists are unhashable	Use a suitable immutable key

Practical Example — Student Placement Profile

PYTHON

Copy

```
student = {
    "name": "Arun",
    "age": 21,
    "department": "CSE",
    "cgpa": 8.5,
    "is_placed": False
}
student["skills"] = ["Python", "SQL"]
student["cgpa"] = 9.0
student["is_placed"] = True
print("Student Name:", student["name"])
print("CGPA:", student["cgpa"])
print("Skills:", student["skills"])
print("Placement Status:", student["is_placed"])
```

Output:

TEXT

Copy

```
Student Name: Arun
CGPA: 9.0
Skills: ['Python', 'SQL']
Placement Status: True
```

This is similar to how structured records are handled in real applications.

Placement Quick Points

- A dictionary stores data as key-value pairs.
- Dictionaries are mutable.
- Dictionary keys must be unique.
- Values are accessed using keys.
- `[]` raises `KeyError` for a missing key.
- `get()` safely handles missing keys.
- Assigning to an existing key updates its value.

- Assigning to a new key adds a new item.
- `update()` adds or updates multiple items.
- `in` checks dictionary keys by default.
- `items()` provides key-value pairs.
- `pop()` removes a key and returns its value.
- Duplicate keys are overwritten by the latest value.
- Dictionary keys must be hashable.

Interview Questions

Q1. What is a dictionary in Python?

A dictionary is a mutable collection that stores data as key-value pairs.

Q2. Are dictionary keys unique?

Yes.

Q3. Can dictionary values be duplicated?

Yes.

Q4. What is the difference between `[]` and `get()`?

`[]` raises `KeyError` for a missing key. `get()` returns `None` or a default value.

Q5. What does the `in` operator check in a dictionary?

Keys.

Q6. What is the difference between `keys()`, `values()`, and `items()`?

`keys()` returns keys, `values()` returns values, and `items()` returns key-value pairs.

Q7. What happens when a duplicate key is used?

The latest value replaces the previous value.

Q8. Can a list be used as a dictionary key?

No. Lists are unhashable.

Q9. Is a dictionary mutable?

Yes.

Q10. How do you safely access a missing key?

```
student.get("email").
```

Q11. What is a nested dictionary?

A dictionary containing another dictionary as a value.

Q12. Does `b = a` copy a dictionary?

No. Both names refer to the same dictionary.

Practice

Use this dictionary:

PYTHON

Copy

```
student = {  
    "name": "Arun",  
    "age": 21,  
    "department": "CSE",  
    "cgpa": 8.5,  
    "is_placed": False  
}
```

1. Print the student's name.
2. Print the CGPA using `get()`.
3. Add an `"email"` key.
4. Update the CGPA to `9.0`.
5. Change `is_placed` to `True`.
6. Check whether `"department"` exists.
7. Print all keys.

8. Print all values.
9. Print all key-value pairs.
10. Remove the `"age"` key using `pop()`.
11. Loop through the dictionary using `items()`.
12. Create a copy of the student dictionary.
13. Add an `"address"` nested dictionary.
14. Access the city from the nested dictionary.
15. Explain the output:

PYTHON

Copy

```
student = {  
print(student.get("phone"))
```

1. Explain the difference:

PYTHON

Copy

```
student = {  
student["phone"]  
student.get("phone")
```

End of Level 3

Advanced Data Structures

Level: 4 | Version: v1.0 | Author: Meptrasoft

This module builds on Level 3's strings, lists, tuples, sets, and dictionaries. It uses a **College Result Management System** as the running example — storing and processing student names, marks, and full student records.

Packing & Unpacking

What Gets Packed and Unpacked

Recall from Level 3 that **packing** combines multiple comma-separated values into a tuple (parentheses optional), and **unpacking** splits a tuple back into separate names, one value per name — with the name count matching the value count, or Python raises `ValueError`.

PYTHON

Copy

```
student = ("Arun", 78, "Pass")
name, marks, status = student
print(name, marks, status)
```

Output:

TEXT

Copy

```
Arun 78 Pass
```

Unpacking Any Iterable

Unpacking works with any iterable — lists, strings, and the results of functions like `zip()` — not only tuples. (`zip()` pairs items from two iterables by position.)

PYTHON

Copy

```
subjects = ["Maths", "Science", "English"]
first, second, third = subjects
print(first, second, third)
```

Output:

TEXT

Copy

Maths Science English

PYTHON

Copy

```
letters = "abc"
a, b, c = letters
print(a, b, c)
```

Output:

TEXT

Copy

a b c

Multiple Assignment

Python allows assigning several names in a single line, using packing and unpacking behind the scenes.

Same Value to Multiple Names

PYTHON

Copy

```
a = b = c = 0
print(a, b, c)
```

Output:

All three names refer to the same value here — useful for initializing counters or totals together.

Different Values in One Line

PYTHON

Copy

```
name, marks, status = "Arun", 78, "Pass"
print(name, marks, status)
```

Output:

TEXT

Copy

Arun 78 Pass

Swapping Values Using Unpacking

A quick recap from Level 3 — swapping two values still uses pack-then- unpack, with no temporary variable needed:

PYTHON

Copy

```
first_marks, second_marks = 78, 92
first_marks, second_marks = second_marks, first_marks
print(first_marks, second_marks)
```

Output:

TEXT

Copy

92 78

Extended Unpacking with *

When the number of values is unknown or larger than the number of names, * collects the "rest" into a list.

PYTHON

Copy

```
marks = [78, 92, 85, 60, 74]
first, *remaining = marks
print(first)
print(remaining)
```

Output:

TEXT

Copy

```
78
[92, 85, 60, 74]
```

`*` can appear at the start, middle, or end — it always collects whatever is left over into a **list**:

PYTHON

Copy

```
marks = [78, 92, 85, 60, 74]

*rest, last = marks
print(rest, last)

first, *middle, last = marks
print(first, middle, last)
```

Output:

TEXT

Copy

```
[78, 92, 85, 60] 74
78 [92, 85, 60] 74
```

TEXT

Copy

```
*name → COLLECTS REMAINING VALUES → LIST
```

Use only one starred name in one unpacking assignment.

Unpacking in a `for` Loop

Unpacking is very common while looping over a list of tuples — for example, student records.

PYTHON

Copy

```
students = [("Arun", 78), ("Divya", 92), ("Karthik", 85)]

for name, marks in students:
    print(name, "scored", marks)
```

Output:

TEXT

Copy

```
Arun scored 78
Divya scored 92
Karthik scored 85
```

Each tuple in the list is unpacked into `name` and `marks` automatically on every loop iteration.

Ignoring Values with `_`

When a value from unpacking is not needed, `_` is a common convention to mark it as intentionally ignored.

PYTHON

Copy

```
name, _, status = ("Arun", 78, "Pass")
print(name, status)
```

Output:

TEXT

Copy

```
Arun Pass
```

TEXT

Copy

```
_ → CONVENTION FOR "VALUE NOT NEEDED"
```


Comprehensions

A **comprehension** builds a new list, set, or dictionary from an existing iterable, in a single readable line.

List Comprehensions

Without a comprehension, building a new list needs a loop and `append()`:

PYTHON

Copy

```
marks = [78, 92, 85, 60, 74]
updated_marks = []

for mark in marks:
    updated_marks.append(mark + 5)

print(updated_marks)
```

Output:

TEXT

Copy

```
[83, 97, 90, 65, 79]
```

The same result with a **list comprehension**:

PYTHON

Copy

```
marks = [78, 92, 85, 60, 74]
updated_marks = [mark + 5 for mark in marks]
print(updated_marks)
```

Output:

TEXT

Copy

```
[ EXPRESSION for ITEM in ITERABLE ]
```

Comprehension with a Condition (Filter)

Adding `if` at the end keeps only the items that satisfy a condition.

PYTHON

Copy

```
marks = [78, 92, 85, 60, 45]
passed = [mark for mark in marks if mark >= 50]
print(passed)
```

Output:

TEXT

Copy

```
[78, 92, 85, 60]
```

TEXT

Copy

```
[ EXPRESSION for ITEM in ITERABLE if CONDITION ]
```

Comprehension with `if...else` (Transform Every Item)

An `if...else` placed *before* the `for` transforms every item instead of filtering it out.

PYTHON

Copy

```
marks = [78, 92, 45, 60, 30]
result = ["Pass" if mark >= 50 else "Fail" for mark in marks]
print(result)
```

Output:

TEXT

Copy

```
['Pass', 'Pass', 'Fail', 'Pass', 'Fail']
```

Filtering form (<code>if</code> at the end)	Transforming form (<code>if...else</code> before <code>for</code>)
Removes items	Keeps every item, changes its value
<code>[x for x in data if cond]</code>	<code>[a if cond else b for x in data]</code>
Result can be shorter than the input	Result is always the same length as the input

List Comprehension vs Loop

Loop with <code>append()</code>	List Comprehension
Multiple lines	Single line
Needs an empty list created first	Builds the list directly
More explicit for complex logic	More readable for simple transforms

TEXT

Copy

SIMPLE TRANSFORM / FILTER → COMPREHENSION
COMPLEX MULTI-STEP LOGIC → LOOP

Nested List Comprehensions

A comprehension can contain another `for`, which is useful for working with a list of lists (for example, marks per subject, per student).

PYTHON

Copy

```
subject_marks = [[78, 92, 85], [60, 74, 88]]

flat_marks = [mark for row in subject_marks for mark in row]
print(flat_marks)
```

Output:

TEXT

Copy

```
[78, 92, 85, 60, 74, 88]
```

TEXT

Copy

```
subject_marks:
[ [78, 92, 85], [60, 74, 88] ]
  ↓ ↓ ↓       ↓ ↓ ↓
  flattened into one list
```

Reading order matters: the `for` clauses run left to right, exactly like nested `for` loops written normally:

PYTHON

Copy

```
subject_marks = [[78, 92, 85], [60, 74, 88]]
flat_marks = []
for row in subject_marks:
    for mark in row:
        flat_marks.append(mark)
```

Building a Nested List with a Comprehension

A comprehension can also *create* nested structure, such as a small multiplication-style grid of pass marks:

PYTHON

Copy

```
students = ["Arun", "Divya"]
subjects = ["Maths", "Science"]

grid = [[f"{s}-{sub}" for sub in subjects] for s in students]
print(grid)
```

Output:

TEXT

Copy

```
[['Arun-Maths', 'Arun-Science'], ['Divya-Maths', 'Divya-Science']]
```

Readability Caution

Nested comprehensions save lines but can quickly become hard to read. Beyond one level of nesting, a normal loop is usually clearer for interview and placement code:

TEXT

Copy

```
1 LEVEL OF NESTING → COMPREHENSION IS FINE
2+ LEVELS / COMPLEX LOGIC → PREFER A NORMAL LOOP
```

Set Comprehensions

A **set comprehension** builds a set — automatically removing duplicate values — using `{}` instead of `[]`.

PYTHON

Copy

```
grades = ["A", "B", "A", "C", "B", "A"]
unique_grades = {grade for grade in grades}
print(unique_grades)
```

Output:

TEXT

Copy

```
{'B', 'A', 'C'}
```

The displayed order may vary because sets are unordered.

TEXT

Copy

```
{ EXPRESSION for ITEM in ITERABLE } → {} means SET
```

Set comprehensions are useful for collecting unique subjects, unique roll numbers, or unique grades from repeated records:

PYTHON

Copy

```
records = [("Arun", "Maths"), ("Divya", "Science"), ("Karthik", "Maths")]
subjects_offered = {subject for name, subject in records}
print(subjects_offered)
```

Output:

TEXT

Copy

```
{'Science', 'Maths'}
```

List Comprehension	Set Comprehension
Uses <code>[]</code>	Uses <code>{}</code>
Result is a list	Result is a set
Duplicates are kept	Duplicates are removed automatically
Order is preserved	Order is not guaranteed

Dictionary Comprehensions

A dictionary comprehension builds a dictionary using `{key: value for ...}`.

`zip()` pairs the two lists by position, as introduced earlier in this module.

PYTHON

Copy

```
students = ["Arun", "Divya", "Karthik"]
marks = [78, 92, 85]

result = {name: mark for name, mark in zip(students, marks)}
print(result)
```

Output:

TEXT

Copy

```
{'Arun': 78, 'Divya': 92, 'Karthik': 85}
```

TEXT

Copy

```
{ KEY : VALUE for ITEM in ITERABLE }
```

Filtering a Dictionary Comprehension

PYTHON

Copy

```
marks = {"Arun": 78, "Divya": 45, "Karthik": 85, "Priya": 40}  
passed = {name: mark for name, mark in marks.items() if mark >= 50}  
print(passed)
```

Output:

TEXT

Copy

```
{'Arun': 78, 'Karthik': 85}
```

Transforming Keys or Values

PYTHON

Copy

```
marks = {"Arun": 78, "Divya": 92, "Karthik": 85}  
result_status = {name: ("Pass" if mark >= 50 else "Fail") for name, mark in marks.items()}  
print(result_status)
```

Output:

TEXT

Copy

```
{'Arun': 'Pass', 'Divya': 'Pass', 'Karthik': 'Pass'}
```

Swapping Keys and Values

PYTHON

Copy

```
roll_numbers = {"Arun": 101, "Divya": 102, "Karthik": 103}
swapped = {roll: name for name, roll in roll_numbers.items()}
print(swapped)
```

Output:

TEXT

Copy

```
{101: 'Arun', 102: 'Divya', 103: 'Karthik'}
```

Comprehension Types Compared

Type	Brackets	Builds	Duplicates
List	[]	list	Kept
Set	{ }	set	Removed
Dictionary	{ k: v }	dict	Keys overwrite

TEXT

Copy

```
[ ]      → LIST
{item}   → SET
{k: v}   → DICTIONARY
```

Nested Structures

Why Nest Data Structures

A **nested data structure** is a data structure that contains other data structures inside it — for example, a list of tuples, a list of dictionaries, or a dictionary of lists. Real applications almost always store data this way, because a single student, order, or record naturally has several related fields.

List of Tuples

PYTHON

Copy

```
students = [("Arun", 78), ("Divya", 92), ("Karthik", 85)]
print(students[0])
print(students[0][0])
print(students[0][1])
```

Output:

TEXT

Copy

```
('Arun', 78)
Arun
78
```

List of Dictionaries

Each student is a dictionary; all students together form a list — this is the most common way real systems represent a table of records.

PYTHON

Copy

```
students = [
    {"name": "Arun", "marks": 78, "status": "Pass"},
    {"name": "Divya", "marks": 92, "status": "Pass"},
    {"name": "Karthik", "marks": 40, "status": "Fail"},
]

for student in students:
    print(student["name"], "-", student["marks"], "-", student["status"])
```

Output:

TEXT

Copy

```
Arun - 78 - Pass
Divya - 92 - Pass
Karthik - 40 - Fail
```

TEXT

Copy

```
[ {...}, {...}, {...} ] → LIST OF DICTIONARIES → TABLE OF RECORDS
```

Dictionary of Lists

All marks for each subject, grouped under one dictionary:

PYTHON

Copy

```
subject_marks = {
    "Maths": [78, 92, 85],
    "Science": [60, 74, 88],
}

print(subject_marks["Maths"])
print(subject_marks["Maths"][0])
```

Output:

TEXT

Copy

```
[78, 92, 85]
78
```

Dictionary of Dictionaries

Each student's full record, keyed by name:

PYTHON

Copy

```
students = {  
    "Arun": {"marks": 78, "status": "Pass"},  
    "Divya": {"marks": 92, "status": "Pass"},  
}  
  
print(students["Arun"])  
print(students["Arun"]["marks"])
```

Output:

TEXT

Copy

```
{'marks': 78, 'status': 'Pass'}  
78
```

TEXT

Copy

```
DICT { NAME : DICT { FIELD : VALUE } }
```

Accessing Nested Values

Each level is accessed with one more `[]` step, moving from the outer structure to the inner value.

PYTHON

Copy

```
college = {  
    "students": [  
        {"name": "Arun", "marks": [78, 92, 85]},  
        {"name": "Divya", "marks": [60, 74, 88]},  
    ]  
}  
  
print(college["students"][0]["name"])  
print(college["students"][0]["marks"][1])
```

Output:

TEXT

Copy

```
Arun
92
```

TEXT

Copy

```
college["students"][0]["marks"][1]
  ↑      ↑      ↑      ↑
  dict   list  dict   list
```

Modifying Nested Values

Navigating to the exact nested position also lets a value be updated in place.

PYTHON

Copy

```
college = {
    "students": [
        {"name": "Arun", "marks": [78, 92, 85]},
    ]
}

college["students"][0]["marks"][0] = 80
print(college["students"][0]["marks"])
```

Output:

TEXT

Copy

```
[80, 92, 85]
```

Shallow Copy vs Deep Copy of Nested Data

`Copy()` on a dictionary or list only copies the **outer** structure — any lists or dictionaries nested inside it are still shared between the original and the copy.

PYTHON

Copy

```
college = {
    "students": [
        {"name": "Arun", "marks": [78, 92, 85]},
    ]
}

college_copy = college.copy()
college_copy["students"][0]["marks"][0] = 100

print(college["students"][0]["marks"])
```

Output:

TEXT

Copy

```
[100, 92, 85]
```

Even though `college_copy` is a separate dictionary, its inner student dictionary and marks list are the *same* objects as in `college` — this is called a **shallow copy**. To copy every nested level independently, use `copy.deepcopy()`:

PYTHON

Copy

```
college = {
import copy

college_deep = copy.deepcopy(college)
college_deep["students"][0]["marks"][0] = 999

print(college["students"][0]["marks"])
```

Output:

TEXT

Copy

```
[100, 92, 85]
```

TEXT

Copy

```
copy()          → copies only the OUTER structure    → SHALLOW COPY
copy.deepcopy() → copies EVERY nested level          → DEEP COPY
```

Real-World Example: Building a Result Report

Combining comprehensions with nested structures is common in real projects — for example, generating a pass/fail report from a list of student records.

PYTHON[Copy](#)

```
students = [  
    {"name": "Arun", "marks": 78},  
    {"name": "Divya", "marks": 45},  
    {"name": "Karthik", "marks": 85},  
]  
  
report = {student["name"]: ("Pass" if student["marks"] >= 50 else "Fail") for student in students}  
print(report)
```

Output:

TEXT[Copy](#)

```
{'Arun': 'Pass', 'Divya': 'Fail', 'Karthik': 'Pass'}
```

TEXT[Copy](#)

```
LIST OF DICTS → DICT COMPREHENSION → NAME : RESULT
```

Common Fresher Mistakes



X Mistake	Problem	✓ Correct Understanding
Unpacking with the wrong number of names	Raises <code>ValueError: too many/not enough values to unpack</code>	The name count must match the value count
Forgetting <code>*</code> collects the "rest" into a list , not a tuple	Code that assumes a tuple result behaves unexpectedly	<code>*rest</code> always produces a list
Trying to swap values with a temporary variable out of habit	Adds an unnecessary extra variable and line	Python can swap directly: <code>a, b = b, a</code>
Writing a comprehension that is hard to read instead of a normal loop	Reduces readability and makes debugging harder	Prefer a loop when logic has more than one condition or nesting level
Confusing <code>[]</code> , <code>{item}</code> , and <code>{k: v}</code> comprehension syntax	Builds the wrong collection type by accident	<code>[]</code> → list, <code>{item}</code> → set, <code>{k: v}</code> → dictionary
Assuming a set comprehension preserves order or duplicates	Code relying on order or duplicate counts breaks silently	Sets remove duplicates and do not guarantee order
Forgetting keys overwrite in a dictionary comprehension	Earlier values for a repeated key are silently lost	A repeated key keeps only the last value seen
Accessing a nested value with too few or wrong <code>[]</code> steps	Raises <code>KeyError</code> , <code>IndexError</code> , or <code>TypeError</code>	Each nesting level needs its own <code>[]</code> step
Modifying a nested structure and expecting the original data unaffected	Unexpected side effects appear because the reference is shared	Nested lists/dicts are shared references; changes are visible everywhere
Using deeply nested comprehensions "to look		

Placement Quick Points

TEXT

[Copy](#)

PACKING → Multiple values combined into one object (tuple)
UNPACKING → Splitting an iterable into separate names
*rest → Collects remaining values into a LIST
SWAP → `a, b = b, a` (no temporary variable needed)
LIST COMP → `[expr for item in iterable]`
FILTER COMP → `[expr for item in iterable if condition]`
TRANSFORM → `[a if cond else b for item in iterable]`
SET COMP → `{expr for item in iterable}` → removes duplicates
DICT COMP → `{key: value for item in iterable}`
NESTED DATA → structure containing another structure (list of dicts, dict of lists, etc.)
ACCESS → one `[]` per nesting level

Interview Questions

Q1. What is packing in Python?

Packing is combining multiple comma-separated values into a single object, usually a tuple.

Q2. What is unpacking in Python?

Unpacking is splitting an iterable into separate names, one value per name.

Q3. What happens if the number of names does not match the number of values while unpacking?

Python raises a `ValueError`.

Q4. Can unpacking be used with lists and strings, not just tuples?

Yes. Unpacking works with any iterable.

Q5. How does Python swap two values without a temporary variable?

Using packing and unpacking together: `a, b = b, a`.

Q6. What does `*` do during unpacking?

It collects the remaining values into a list; it can appear at the start, middle, or end.

Q7. What type of object does `*rest` produce?

A list.

Q8. What is the underscore `_` used for during unpacking?

It is a convention to indicate that a value from unpacking is intentionally ignored.

Q9. What is a list comprehension?

A concise way to build a new list from an iterable in a single line, in the form

`[expression for item in iterable]`.

Q10. How is filtering added to a list comprehension?

By adding `if condition` at the end of the comprehension.

Q11. How is a transforming `if...else` written in a comprehension, and how is it different from a filtering `if`?

It is written before the `for` clause, as `[a if condition else b for item in iterable]`.

It keeps every item and changes its value, while a filtering `if` at the end removes items instead.

Q12. What is a nested list comprehension?

A comprehension that contains another `for` clause, commonly used to flatten a list of lists.

Q13. When should a normal loop be preferred over a comprehension?

When the logic involves more than one condition or more than one level of nesting, since readability matters more than compressing lines.

Q14. What is a set comprehension and how is its syntax different from a list comprehension?

A set comprehension builds a set using `{expression for item in iterable}`; unlike a list comprehension, duplicate values are automatically removed.

Q15. What is a dictionary comprehension?

A concise way to build a dictionary using `{key: value for item in iterable}`.

Q16. What happens when a dictionary comprehension produces a repeated key?

The last value seen for that key is kept; earlier values are overwritten.

Q17. What is a nested data structure?

A data structure that contains other data structures inside it, such as a list of dictionaries or a dictionary of lists.

Q18. Give an example of a common nested structure used to represent multiple records.

A list of dictionaries, where each dictionary represents one record (for example, one student).

Q19. How many `[]` steps are needed to access a value nested three levels deep?

Three — one step per nesting level.

Practice

Use a College Result Management System.

1. Pack a student's name, marks, and status into a tuple in one line.
2. Unpack that tuple into three separate names.
3. Try unpacking into the wrong number of names and observe the error.
4. Unpack a string into individual character names.
5. Assign the same value `0` to three names in one line.
6. Assign three different values to three names in one line.
7. Swap two mark values without using a temporary variable.
8. Use `*rest` to collect all marks except the first from a list.

9. Use `*rest` to collect all marks except the last from a list.
10. Loop over a list of `(name, marks)` tuples using unpacking.
11. Use `_` to ignore the status field while unpacking a student record.
12. Write a list comprehension that adds 5 bonus marks to every value in a list.
13. Write a list comprehension that keeps only marks greater than or equal to 50.
14. Write a list comprehension that labels each mark as `"Pass"` or `"Fail"`.
15. Flatten a list of subject-wise mark lists using a nested list comprehension.
16. Build a small nested list pairing student names with subject names.
17. Write a set comprehension that collects unique grades from a list.
18. Write a set comprehension that collects unique subjects from a list of records.
19. Write a dictionary comprehension mapping student names to marks using `zip()`.
20. Write a dictionary comprehension that keeps only students who passed.
21. Write a dictionary comprehension that swaps keys and values of a roll-number dictionary.
22. Create a list of dictionaries, each representing one student's record.
23. Create a dictionary of lists, grouping marks by subject.
24. Create a dictionary of dictionaries, keyed by student name.
25. Access a specific student's marks from a nested `list of dicts inside a dict`.
26. Modify one nested mark value in place and print the updated structure.
27. Build a pass/fail report dictionary from a list of student-record dictionaries using a dictionary comprehension.

Predict the output:

For set outputs, the order may vary.

PYTHON

Copy

```
name, marks = "Arun", 78
print(name, marks)
```

PYTHON

Copy

```
a, *b = [78, 92, 85, 60]
print(a)
print(b)
```

PYTHON

Copy

```
marks = [78, 45, 92, 30]
result = [m for m in marks if m >= 50]
print(result)
```

PYTHON

Copy

```
grades = ["A", "B", "A", "C"]
print({g for g in grades})
```

PYTHON

Copy

```
students = {"Arun": 78, "Divya": 92}
print({v: k for k, v in students.items()})
```

Find the mistake:

PYTHON

Copy

```
name, marks, status = ("Arun", 78)
```

PYTHON

Copy

```
students = [{"name": "Arun", "marks": [78, 92]}]
print(students["name"])
```

PYTHON

Copy

```
marks = {"Arun": 78, "Arun": 92}
print(marks)
```

Final task: Build a small College Result Management System that stores several students as a list of dictionaries (`name`, `marks`), then use comprehensions to (a) produce a list of only the passing students' names, (b) produce a set of unique pass/fail statuses, and (c) produce a dictionary mapping each student's name to their result status.

Output:

TEXT

Copy

```
PACK / UNPACK → MOVE VALUES IN AND OUT OF NAMES
COMPREHENSION → BUILD A NEW COLLECTION IN ONE READABLE LINE
NESTED DATA → STRUCTURE INSIDE STRUCTURE, ONE [] PER LEVEL
```

End of Level 4



Control Flow

Level: 5 | Version: v1.0 | Author: Meptrasoft

Control flow decides **which** code runs and **how many times**. This module uses a **Student Placement Eligibility System** for conditionals and an **E-commerce Order Processing System** for loops.

Conditional Statements

A **conditional statement** is used to make decisions in a program. It allows Python to execute different code based on a condition.

PYTHON

Copy

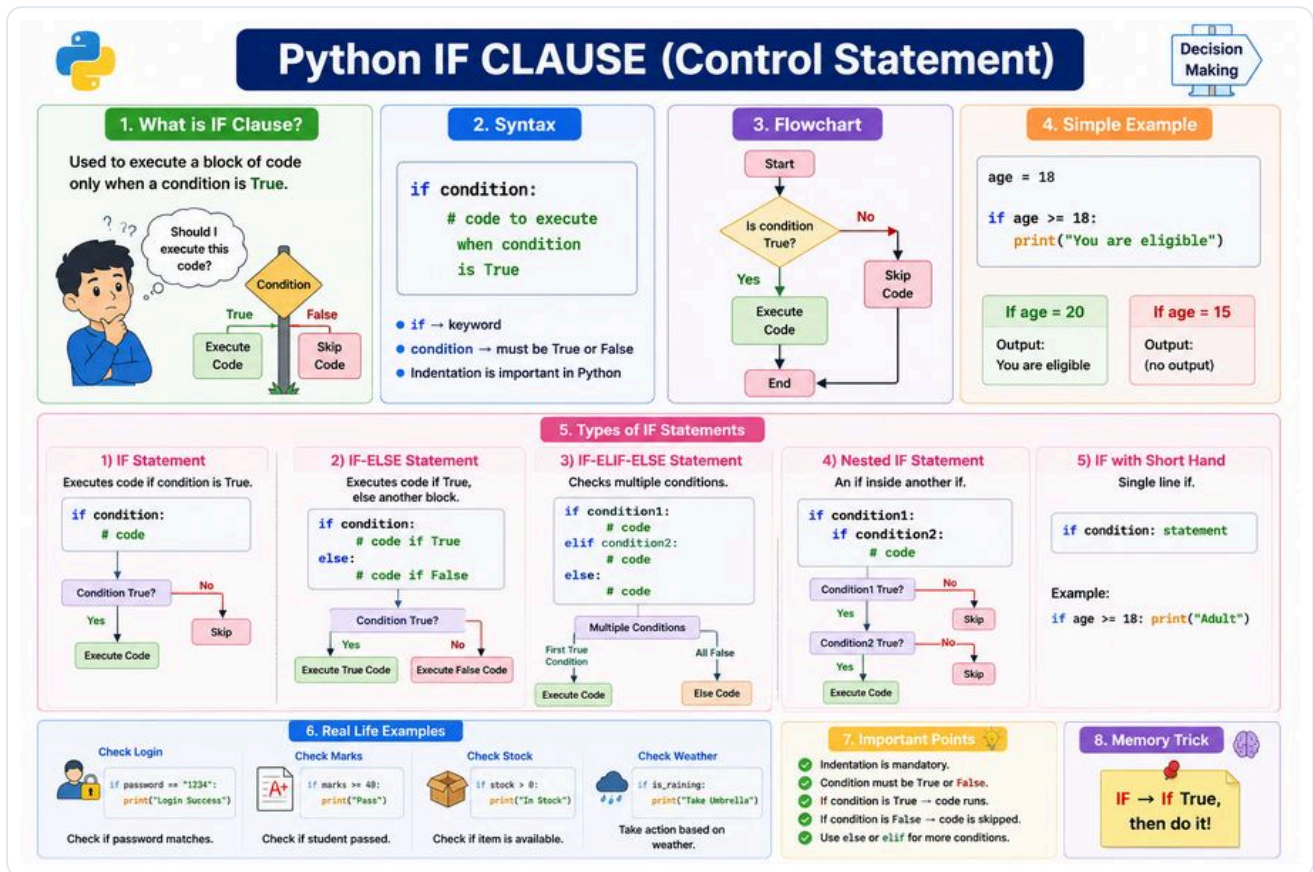
```
print(cgpa = 8.5)
print(cgpa)
if cgpa >= 7.0:
    print("Eligible for placement")
```

Output:

TEXT

Copy

```
8.5
Eligible for placement
```



Why Do We Need Conditional Statements?

Real applications need to make decisions: Is a student eligible for placement? Is a user allowed to log in? Is a payment successful? Is a product in stock?

PYTHON

Copy

```
print(cgpa = 8.5)
```

The program needs to decide:

Output:

TEXT

Copy

```
Is CGPA >= 7.0?
↓
Yes or No?
```


Types of Conditional Statements

Statement	Purpose
<code>if</code>	Execute code when a condition is true
<code>elif</code>	Check another condition
<code>else</code>	Execute when previous conditions are false
Nested Conditions	Condition inside another condition
Conditional Expression	Write a simple decision in one line

We will use this data throughout the conditionals topic.

PYTHON

Copy

```
cgpa = 8.5
backlogs = 0
skills = ["Python", "SQL"]
is_placed = False
```

Truthy and Falsy Values

Before we use conditions, it helps to know that **every value in Python is either "truthy" or "falsy"** when checked in a Boolean context (like an `if`). A **falsy** value acts like `False`, and a **truthy** value acts like `True`.

Common **falsy** values include:

PYTHON

Copy

```
False      # the Boolean False
0          # integer zero
0.0        # float zero
""         # empty string
[]         # empty list
{}         # empty dictionary
()         # empty tuple
set()      # empty set
None       # the None object
```

Most other common values are **truthy** — non-zero numbers, non-empty strings, non-empty lists, and so on.

PYTHON

Copy

```
if []:
    print("runs")  # skipped: empty list is falsy

if [1]:
    print("runs")  # runs: non-empty list is truthy
```

You can check any value with `bool(x)`:

PYTHON

Copy

```
print(bool([]))    # False
print(bool([1]))   # True
print(bool(0))     # False
print(bool("hi"))  # True
```

if Statement

The `if` statement executes a block of code **only when a condition is truthy**.

PYTHON

Copy

```
if condition:
    statement
```

Example:

PYTHON

Copy

```
cgpa = 8.5
if cgpa >= 7.0:
    print("Eligible for placement")
```

Output:

TEXT

Copy

Eligible for placement

How Does `if` Work?

TEXT

Copy

```
cgpa >= 7.0
  ↓
Is it True?
 /   \
Yes   No
  ↓   ↓
Execute code  Skip code
```

For `cgpa = 8.5`, the condition `cgpa >= 7.0` is `8.5 >= 7.0 → True`, so the `print` runs.

What Happens When the Condition is False?

PYTHON

Copy

```
cgpa = 6.5
if cgpa >= 7.0:
    print("Eligible for placement")
```

No output, because `6.5 >= 7.0 → False` and the `if` block is skipped.

Indentation is Important

Python uses **indentation** to identify the code inside a conditional block.

Correct:

PYTHON

Copy

```
cgpa = 6.5
if cgpa >= 7.0:
    print("Eligible")
```

Wrong (causes `IndentationError`):

PYTHON

Copy

```
if cgpa >= 7.0:
print("Eligible")
```

Output:

TEXT

Copy

```
Condition ends with :
      ↓
Indented code belongs to the condition
```

Python commonly uses **4 spaces** for indentation.

`else` Statement

The `else` statement executes when the `if` condition is false.

PYTHON

Copy

```
if condition:
    statement
else:
    statement
```

Placement example:

Output:

TEXT

Copy

```
Not eligible for placement
```

if vs **else**

Statement	Meaning
if	Run when condition is truthy
else	Run when the if / elif chain reaches it with no earlier branch taken

else does not have its own condition.

Wrong:

PYTHON

Copy

```
else cgpa < 7.0:  
    print("Not Eligible")
```

elif Statement**elif** means **else if**. It is used when we need to check **multiple conditions in order**.

PYTHON

Copy

```
if condition1:  
    statement  
elif condition2:  
    statement  
else:  
    statement
```

Placement example — CGPA category:

PYTHON

Copy

```
cgpa = 8.5
if cgpa >= 9.0:
    print("Excellent")
elif cgpa >= 8.0:
    print("Very Good")
elif cgpa >= 7.0:
    print("Good")
else:
    print("Needs Improvement")
```

Output:

TEXT

Copy

Very Good

First True Condition Wins

Python checks conditions **from top to bottom**. Once a condition is true, the remaining `elif`/`else` blocks are skipped.

PYTHON

Copy

```
cgpa = 9.5
if cgpa >= 7.0:
    print("Good")
elif cgpa >= 8.0:
    print("Very Good")
elif cgpa >= 9.0:
    print("Excellent")
```

Output:

TEXT

Copy

Good

Because `9.5 >= 7.0 → True`, Python prints `"Good"` and stops. When checking ranges, arrange conditions from the most specific/highest downward.

TEXT

Copy

```
if / elif checks from TOP to BOTTOM
First matching condition wins
```

if VS elif VS else

Statement	Has Condition?	Purpose
if	Yes	Check the first condition
elif	Yes	Check another condition if earlier branches failed
else	No	Run when no previous condition matched

TEXT

Copy

```
if → CHECK FIRST
elif → CHECK ANOTHER
else → OTHERWISE
```

Multiple if vs if-elif

Multiple independent if statements are each checked separately.

PYTHON

Copy

```
cgpa = 9.5
if cgpa >= 7.0:
    print("Eligible")
if cgpa >= 8.0:
    print("Priority Candidate")
if cgpa >= 9.0:
    print("Top Candidate")
```

Output:

TEXT

Copy

Eligible
Priority Candidate
Top Candidate

An `if-elif` chain stops after the first match.

PYTHON

Copy

```
cgpa = 9.5
if cgpa >= 9.0:
    print("Top Candidate")
elif cgpa >= 8.0:
    print("Priority Candidate")
elif cgpa >= 7.0:
    print("Eligible")
```

Output:

TEXT

Copy

Top Candidate

Multiple <code>if</code>	<code>if-elif</code>
Conditions are independent	Conditions belong to one decision chain
Multiple blocks can execute	Only one branch executes
Every <code>if</code> is checked	Stops after first matching branch
Use for multiple independent checks	Use for mutually exclusive choices

TEXT

Copy

Need MULTIPLE independent checks? → Use multiple `if`
Need ONE result from many choices? → Use `if-elif`

Combining Conditions with Logical Operators

Conditions are often combined using `and`, `or`, and `not`.

Using `and` — a student must satisfy all requirements:

PYTHON

Copy

```
cgpa = 8.5
backlogs = 0
if cgpa >= 7.0 and backlogs == 0:
    print("Eligible for placement")
```

Using `or` — accept Python or Java developers:

PYTHON

Copy

```
skills = ["Python", "SQL"]
if "Python" in skills or "Java" in skills:
    print("Required programming skill available")
```

Using `not` — check whether a student is not already placed:

PYTHON

Copy

```
is_placed = False
if not is_placed:
    print("Available for placement")
```

Complete placement condition:

PYTHON

Copy

```
cgpa = 8.5
backlogs = 0
skills = ["Python", "SQL"]
is_placed = False
if (
    cgpa >= 7.0
    and backlogs == 0
    and "Python" in skills
    and not is_placed
):
    print("Eligible for placement")
else:
    print("Not eligible for placement")
```

Output:

TEXT

Copy

```
Eligible for placement
```

Nested Conditions

A nested condition is an `if` statement inside another conditional block.

PYTHON

Copy

```
if condition1:
    if condition2:
        statement
```

Placement example:

PYTHON

Copy

```
cgpa = 8.5
backlogs = 1
if cgpa >= 7.0:
    if backlogs == 0:
        print("Eligible for placement")
    else:
        print("Not eligible due to backlogs")
else:
    print("Not eligible due to CGPA")
```

Output:

TEXT

Copy

```
Not eligible due to backlogs
```

The inner condition is checked only when the outer condition allows entry into that block. Use nested conditions when the **second decision depends on the first**.

PYTHON

Copy

```
is_logged_in = True
is_admin = True
if is_logged_in:
    if is_admin:
        print("Show Admin Dashboard")
```

Nested **if** vs **and**

Use **and** when conditions are one simple combined decision:

PYTHON

Copy

```
age = 21
backlogs = 0
cgpa = 8.5
if cgpa >= 7.0 and backlogs == 0:
    print("Eligible")
```

Use nested conditions when the next check or action logically depends on the previous branch, especially for different messages at each stage:

PYTHON

Copy

```
age = 21
backlogs = 0
cgpa = 8.5
if cgpa >= 7.0:
    print("CGPA requirement passed")
    if backlogs == 0:
        print("Eligible")
    else:
        print("Backlog requirement failed")
else:
    print("CGPA requirement failed")
```

Avoid unnecessary deep nesting.

Output:

TEXT

Copy

```
CGPA requirement passed
Eligible
```

Conditional Expressions

A conditional expression (ternary operator) writes a simple `if-else` decision in one line.

PYTHON

Copy

```
value_if_true if condition else value_if_false
```

Normal `if-else`:

PYTHON

Copy

```
cgpa = 8.5
if cgpa >= 7.0:
    status = "Eligible"
else:
    status = "Not Eligible"
print(status)
```

Conditional expression:

PYTHON

Copy

```
cgpa = 8.5
status = "Eligible" if cgpa >= 7.0 else "Not Eligible"
print(status)
```

Output:

TEXT

Copy

Eligible

Read it as:

TEXT

Copy

TRUE VALUE if CONDITION else FALSE VALUE

Real-time examples:

PYTHON

Copy

```
is_placed = True
status = "Placed" if is_placed else "Not Placed"

number = 10
result = "Even" if number % 2 == 0 else "Odd"
```

Use it for simple two-result decisions. Avoid it for complex logic:

PYTHON

Copy

```
cgpa = 8.5
status = "Excellent" if cgpa >= 9 else "Good" if cgpa >= 7 else "Poor"
```

Prefer normal `if-elif-else`:

PYTHON

Copy

```
cgpa = 8.5
if cgpa >= 9:
    status = "Excellent"
elif cgpa >= 7:
    status = "Good"
else:
    status = "Poor"
print(status)
```

Output:

TEXT

Copy

Good

Complete Placement Eligibility System

PYTHON

Copy

```
cgpa = 8.5
backlogs = 0
skills = ["Python", "SQL"]
is_placed = False
if cgpa >= 7.0:
    print("CGPA requirement passed")
    if backlogs == 0:
        print("Backlog requirement passed")
        if "Python" in skills:
            print("Skill requirement passed")
            if not is_placed:
                print("Eligible for placement")
            else:
                print("Already placed")
        else:
            print("Required skill missing")
    else:
        print("Backlog requirement failed")
else:
    print("CGPA requirement failed")
```

Cleaner version when only the final result matters:

PYTHON

Copy

```
age = 21
backlogs = 0
cgpa = 8.5
eligible = (
    cgpa >= 7.0
    and backlogs == 0
    and "Python" in skills
    and not is_placed
)
if eligible:
    print("Eligible for placement")
else:
    print("Not eligible for placement")
```

Output:

TEXT

Copy

Need to know WHY the student failed? → Step-by-step conditions can help
Need only final eligibility? → Use a clear combined condition

Common Fresher Mistakes — Conditions

X Mistake	Problem	✓ Correct Understanding
Missing <code>:</code>	Invalid syntax	Add <code>:</code> after the condition
No indentation	Python cannot identify the block	Indent the block
Using <code>=</code> in a condition	<code>=</code> is assignment	Use <code>==</code> for equality
Wrong <code>elif</code> order	Earlier broad condition matches first	Order ranges carefully
Thinking every <code>if</code> is part of one chain	Separate <code>if</code> statements are independent	Use <code>elif</code> for one decision chain
Adding a condition after <code>else</code>	<code>else</code> has no condition	Use <code>elif</code>
Deep unnecessary nesting	Code becomes difficult to read	Combine simple conditions
Using ternary for complex logic	Reduces readability	Use normal <code>if-elif-else</code>
Forgetting first true branch wins	Later <code>elif</code> branches are skipped	Check top-to-bottom order

Placement Quick Points — Conditions

- Conditional statements are used for decision-making.

- `if` checks a condition; `elif` checks another if earlier branches did not execute; `else` handles the rest.
- Conditions are evaluated using truthiness.
- Python uses indentation to define code blocks.
- `if` and `elif` conditions are checked from top to bottom; the first matching branch executes.
- Multiple independent `if` statements can execute multiple blocks.
- Use multiple `if` for independent checks; use `if-elif` for mutually exclusive choices.
- Nested conditions place one decision inside another; use them for dependent step-by-step decisions.
- Use logical operators to combine simple conditions.
- A conditional expression writes a simple `if-else` in one expression; avoid complex nesting of it.

Interview Questions — Conditions

Q1. What is a conditional statement?

A conditional statement is used to make decisions based on conditions.

Q2. What is the purpose of `if`?

It executes code when a condition is truthy.

Q3. What is `elif`?

It checks another condition when previous branches in the same chain did not execute.

Q4. Does `else` have a condition?

No.

Q5. What happens after an `if` or `elif` condition matches?

That branch executes, and the remaining branches in the same chain are skipped.

Q6. What is the difference between multiple `if` statements and `if-elif`?

Multiple `if` statements are independent and multiple blocks may execute. In an `if-elif` chain, only the first matching branch executes.

Q7. Why is indentation important in Python?

Indentation defines code blocks.

Q8. What is a nested condition?

A conditional statement inside another conditional block.

Q9. When should nested conditions be used?

When a later decision logically depends on an earlier branch or needs step-by-step handling.

Q10. What is a conditional expression?

A one-line expression used for a simple `if-else` decision: `value_if_true if condition else value_if_false`.

Q11. What is the difference between `=` and `==`?

`=` assigns a value. `==` compares values.

Q12. Why is the order of `elif` conditions important?

Because conditions are checked from top to bottom and the first matching branch executes.

Practice — Conditions

Use:

PYTHON

Copy

```
cgpa = 8.5
backlogs = 0
skills = ["Python", "SQL"]
is_placed = False
```

1. Check whether the student's CGPA is at least 7.0.
2. Print "Eligible" or "Not Eligible" using if-else.
3. Create CGPA categories using if-elif-else.
4. Check CGPA and backlog requirements using and.
5. Check whether the student knows "Python" or "Java".
6. Check whether the student is not already placed.
7. Create a nested condition for CGPA and backlogs.
8. Rewrite the nested condition using and.
9. Create a conditional expression for "Placed" or "Not Placed".
10. Create a conditional expression to check even or odd.
11. Explain why only one branch executes:

PYTHON

Copy

```
cgpa = 9.5
if cgpa >= 9.0:
    print("Excellent")
elif cgpa >= 8.0:
    print("Very Good")
elif cgpa >= 7.0:
    print("Good")
```

1. Predict the output:

PYTHON

Copy

```
cgpa = 9.5
if cgpa >= 7.0:
    print("Eligible")
if cgpa >= 8.0:
    print("Priority")
if cgpa >= 9.0:
    print("Top Candidate")
```

1. Find the mistake:

PYTHON

Copy

```
if cgpa >= 7.0
    print("Eligible")
```

1. Find the mistake:

PYTHON

Copy

```
if cgpa = 7.0:
    print("Eligible")
```

1. Explain the difference between two separate `if` statements and an `if/elif` chain.

Loops

A loop executes a block of code repeatedly.

Without a loop:

PYTHON

Copy

```
print("Process Order 1")
print("Process Order 2")
print("Process Order 3")
```

With a loop:

PYTHON

Copy

```
orders = ["ORD101", "ORD102", "ORD103"]
for order in orders:
    print("Processing:", order)
```

Why Do We Need Loops?

Real applications process large amounts of data — thousands of orders, emails to many users, file records, API responses. Imagine 10,000 orders; we cannot write `process(order1)` 10,000 times. Instead:

PYTHON

Copy

```
orders = ["ORD101", "ORD102", "ORD103"]
for order in orders:
    process(order)
```

Types of Loops in Python

Loop	Purpose
<code>for</code>	Process items
<code>while</code>	Repeat based on a condition

Output:

TEXT

Copy

```
for → PROCESS ITEMS
while → REPEAT BASED ON CONDITION
```

Throughout the loops topic, we use an **E-commerce Order Processing System**.

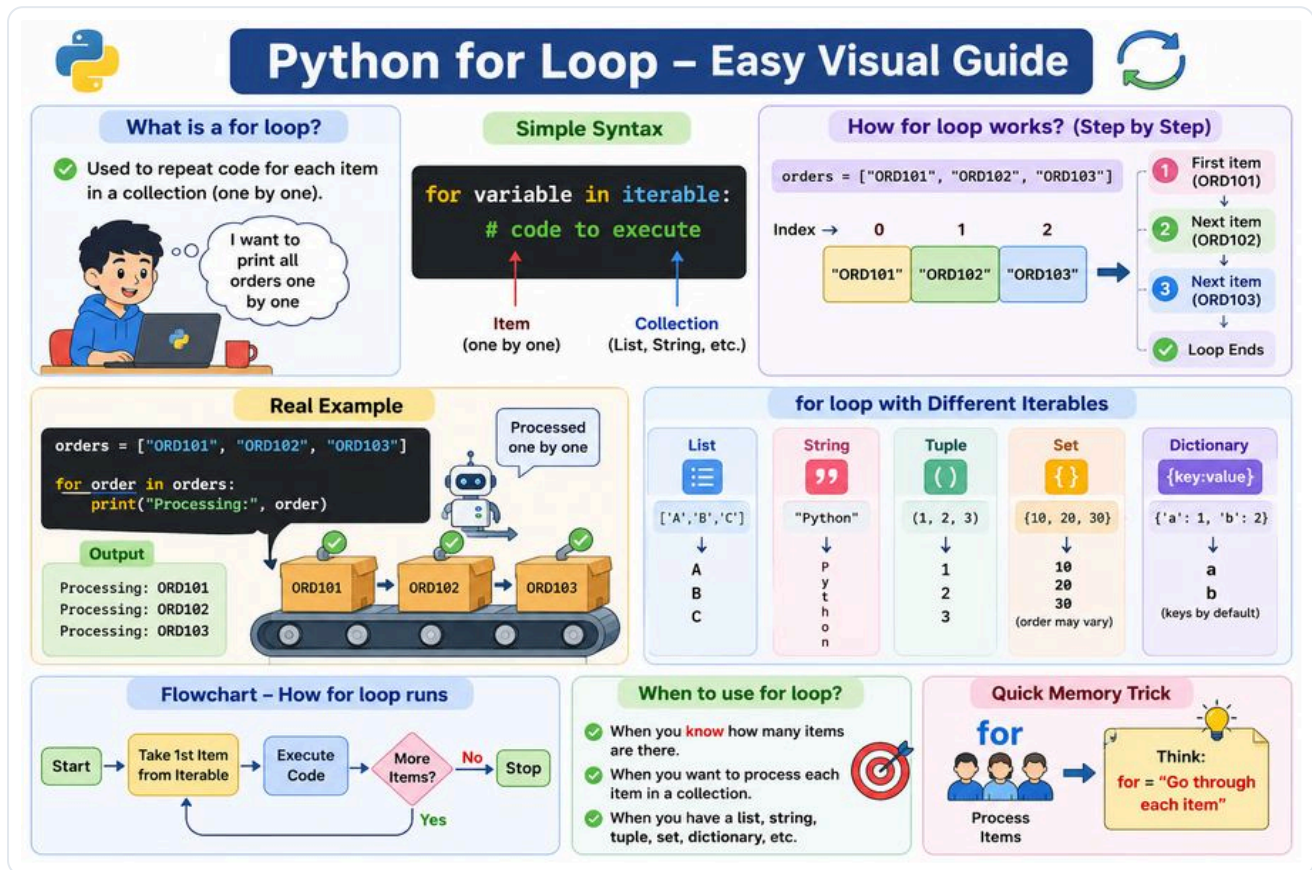
PYTHON

Copy

```
orders = ["ORD101", "ORD102", "ORD103"]
```

for Loop

A **for** loop processes items from an iterable one by one.



PYTHON

Copy

```
for variable in iterable:
    statement
```

Example — process orders:

PYTHON

Copy

```
orders = ["ORD101", "ORD102", "ORD103"]
for order in orders:
    print("Processing:", order)
```

Output:

TEXT

Copy

```
Processing: ORD101
Processing: ORD102
Processing: ORD103
```

How Does a `for` Loop Work?

TEXT

Copy

```
orders = ["ORD101", "ORD102", "ORD103"]
1st Iteration → order = "ORD101"
2nd Iteration → order = "ORD102"
3rd Iteration → order = "ORD103"
                ↓
                Loop Ends
```

Here `orders` is the iterable and `order` is the **loop variable**, which stores the current item during each iteration. One complete execution of the loop block is called an **iteration**.

`for` Loop with Different Iterables

A `for` loop works with strings, lists, tuples, sets, dictionaries, and `range()`.

PYTHON

Copy

```
order_id = "ORD101"
for character in order_id:
    print(character)
```

PYTHON

Copy

```
statuses = ("Pending", "Paid", "Shipped")
for status in statuses:
    print(status)
```

PYTHON

Copy

```
categories = {"Laptop", "Mobile", "Books"}
for category in categories:
    print(category)
```

The iteration order of a set is not guaranteed.

Dictionary iteration processes **keys** by default:

PYTHON

Copy

```
order = {"id": "ORD101", "amount": 1500, "status": "Paid"}
for key in order:
    print(key)

for value in order.values():
    print(value)

for key, value in order.items():
    print(key, value)
```

Output:

TEXT

Copy

```
id
amount
status
ORD101
1500
Paid
id ORD101
amount 1500
status Paid
```

while Loop

A **while** loop repeatedly executes code while a condition is truthy.



Python WHILE LOOP



Repeats a block of code as long as a condition is True

1. WHAT IS WHILE LOOP?

Executes code repeatedly while the condition is **True**.



I will keep working until condition is **False**!

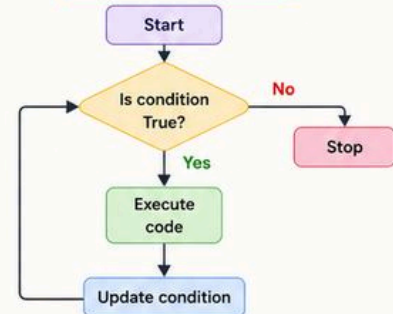
2. SYNTAX

```
while condition:
    # code to execute
```

Condition
(Must be True to run)

Code Block
Executes while condition is True

3. HOW IT WORKS?



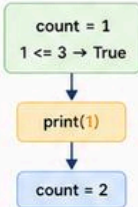
4. EXAMPLE – COUNT FROM 1 TO 3

CODE

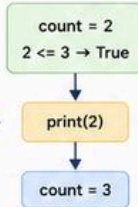
```
count = 1
while count <= 3:
    print(count)
    count += 1
```

STEP BY STEP EXECUTION

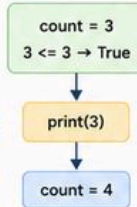
Iteration 1



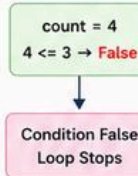
Iteration 2



Iteration 3



Iteration 4



OUTPUT

```
1
2
3
```

5. IMPORTANT POINTS

- ✓ The condition is checked **BEFORE** executing the code.
- ✓ If condition is False at the beginning, the code will not run even once.
- ✓ Make sure to update the condition inside the loop, otherwise it will become an infinite loop.



Always change the condition inside the loop!

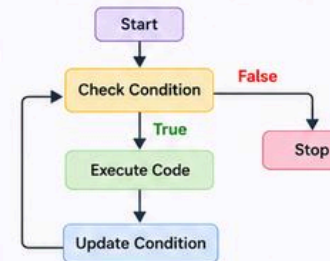
6. REAL LIFE EXAMPLE – LOGIN ATTEMPTS



Allow user maximum 3 login attempts

```
attempt = 1
while attempt <= 3:
    password = input("Enter password: ")
    if password == "1234":
        print("Login Successful!")
        break
    else:
        print("Try again")
        attempt += 1
if attempt > 3:
    print("Access Denied")
```

7. FLOWCHART SUMMARY



MEMORY TRICK:

WHILE = Will Help In Loop Execution

Check → Execute → Update → Repeat

PYTHON

```
while condition:
    statement
```

Copy

Real-time example — payment retry (max three attempts):

PYTHON

Copy

```
attempt = 1
while attempt <= 3:
    print("Payment Attempt:", attempt)
    attempt += 1
```

Output:

TEXT

Copy

```
Payment Attempt: 1
Payment Attempt: 2
Payment Attempt: 3
```

TEXT

Copy

```
attempt = 1 → True → Execute
attempt = 2 → True → Execute
attempt = 3 → True → Execute
attempt = 4 → False → Stop
```

Infinite Loop

A loop that never stops is an **infinite loop**.

Good question. This is something every online code runner needs to handle.

If a user clicks **Run** and the code has an infinite loop, the program will never finish unless you stop it.

Example

PYTHON

Copy

```
while True:
    print("Hello")
```

This keeps printing forever.

Or:

PYTHON

Copy

```
i = 1
while i <= 10:
    print(i)
    # Forgot: i += 1
```

Since `i` never changes, the loop never ends.

What happens if you don't prevent it?

- ✗ Browser becomes slow or freezes.
- ✗ CPU usage goes very high.
- ✗ Memory keeps increasing.
- ✗ The Run button never finishes.
- ✗ User has to refresh the page.

How to prevent it?

1. Execution timeout (Best practice) ★

Stop the program automatically after a few seconds (e.g., 3–5 seconds).

TEXT

Copy

```
Running...
⌚ Execution timed out after 5 seconds.
Possible infinite loop detected.
```

This is the most common solution.

2. Stop button

Provide a **Stop** button so users can manually interrupt execution.

TEXT

Copy

```
[ Run ] [ Stop ]
```

3. Limit output

If the code prints too many lines, stop after a limit.

Example:

TEXT

Copy

```
Maximum output limit (1000 lines) reached.  
Execution stopped.
```

This prevents huge outputs from consuming memory.

4. Detect common infinite loops (Optional)

Before running, warn about obvious cases like:

PYTHON

Copy

```
while True:
```

Or

PYTHON

Copy

```
while i < 10:  
    # i never changes
```

This won't catch every case, but it can warn users.

Best setup for your Digital Notes

For your Pyodide-based Run button, we recommend:

- ✓ 5-second execution timeout.
- ✓ Stop button.
- ✓ Maximum output limit (1000 lines or 50 KB).

"Execution stopped because the program exceeded the time limit. Check for an infinite loop."

This combination gives users a good experience while protecting the browser from hanging or consuming excessive resources.

PYTHON

Copy

```
attempt = 1
while attempt <= 3:
    print("Retrying Payment")
```

`attempt` never changes, so `1 <= 3` is always true. When using `while`, make sure the loop state changes:

PYTHON

Copy

```
attempt += 1
```

Realistic retry with an exit:

PYTHON

Copy

```
attempt = 1
max_attempts = 3
payment_success = False
while attempt <= max_attempts:
    print("Trying payment:", attempt)
    if payment_success:
        print("Payment Successful")
        break
    attempt += 1
```

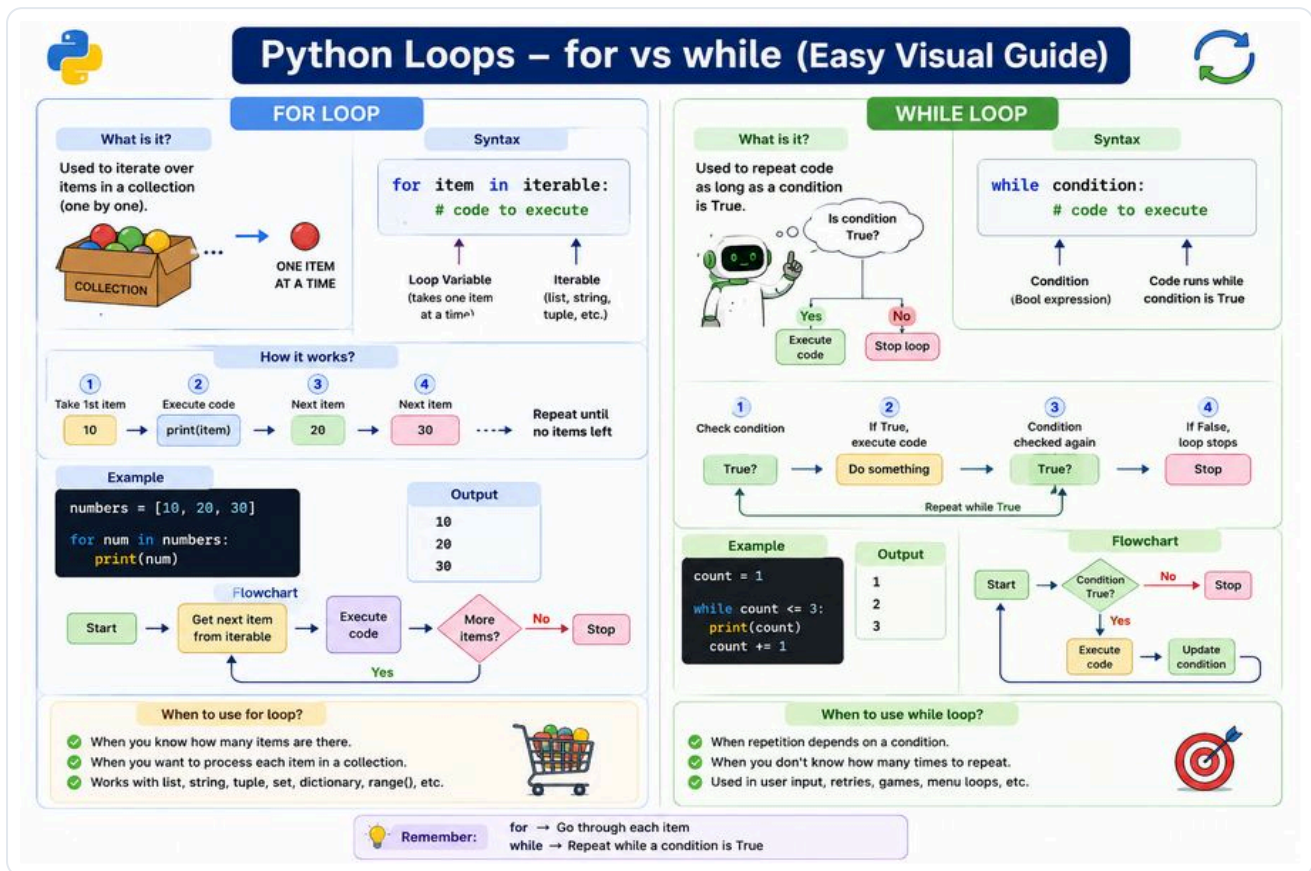
Output:

TEXT

Copy

Need to REPEAT BASED ON CONDITION? → while

for VS while



for	while
Processes items	Repeats based on condition
Works with iterables	Works with Boolean conditions
Stops when iterable ends	Stops when condition becomes falsy
Common for collections	Common for retries and repeated checks

TEXT

Copy

PROCESS ITEMS → for
REPEAT BASED ON CONDITION → while

range()

`range()` generates a sequence of integers, commonly used with a `for` loop.

PYTHON

Copy

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

Output:

TEXT

Copy

```
0  
1  
2  
3  
4
```

TEXT

Copy

STOP VALUE IS EXCLUDED

`range(start, stop)`:

PYTHON

Copy

```
for number in range(101, 105):  
    print(number)    # 101 102 103 104
```

`range(start, stop, step)`:

PYTHON

Copy

```
for number in range(0, 10, 2):  
    print(number)    # 0 2 4 6 8
```

Output:

TEXT

Copy

```
0
2
4
6
8
```

Real-time — generate order IDs:

PYTHON

Copy

```
for number in range(101, 106):
    print(f"ORD{number}")
```

Reverse with a negative step:

PYTHON

Copy

```
for seconds in range(5, 0, -1):
    print(seconds)    # 5 4 3 2 1
```

The step direction must match the movement. `range(5, 1, 1)` produces **no values** because the step moves upward while the stop value is lower.

Syntax	Meaning
<code>range(stop)</code>	<code>0</code> to <code>stop - 1</code>
<code>range(start, stop)</code>	<code>start</code> to <code>stop - 1</code>
<code>range(start, stop, step)</code>	Generates values using a step

Output:

TEXT

Copy

```
5
4
3
2
1
```


Loop Control Statements

Some statements change the normal execution of a loop. `pass` is included here because it is commonly used as a placeholder inside loop or conditional blocks.

Statement	Purpose
<code>break</code>	Stop the loop
<code>continue</code>	Skip current iteration
<code>pass</code>	Do nothing

`break`

`break` immediately stops the nearest enclosing loop.

Real-time example — fraud detection:

PYTHON

Copy

```
orders = ["ORD101", "ORD102", "FRAUD", "ORD104"]
for order in orders:
    if order == "FRAUD":
        print("Fraud Detected")
        break
    print("Processing:", order)
```

Output:

TEXT

Copy

```
Processing: ORD101
Processing: ORD102
Fraud Detected
```

`ORD104` is not processed.

TEXT

Copy

`break → STOP THE LOOP`

Common uses: stop when fraud is detected, after payment success, when a search result is found, after valid input, or to exit a menu.

continue

Continue skips the rest of the **current iteration** and moves to the next.

Real-time example — skip cancelled orders:

PYTHON

Copy

```
orders = [
    ("ORD101", "Paid"),
    ("ORD102", "Cancelled"),
    ("ORD103", "Paid")
]
for order_id, status in orders:
    if status == "Cancelled":
        continue
    print("Processing:", order_id)
```

Output:

TEXT

Copy

```
Processing: ORD101
Processing: ORD103
```

The loop does not stop; only the current iteration is skipped.

TEXT

Copy

`continue → SKIP CURRENT ITERATION`

break VS continue

break	continue
Stops the entire loop	Skips one iteration
Loop ends	Loop continues
No next iteration	Moves to next iteration

TEXT

Copy

```
break    → STOP
continue → SKIP
```

pass

`pass` is a statement that **does nothing**. It is used as a placeholder when Python requires a statement but the logic is not implemented yet.

PYTHON

Copy

```
if True:
    pass
```

Real-time — future refund feature:

PYTHON

Copy

```
order_status = "Refunded"
if order_status == "Refunded":
    pass
```

`pass` inside a function:

PYTHON

Copy

```
def process_refund():
    print(pass)
```

Important fresher correction — `pass` does **not** skip; it does nothing.

Output:

TEXT

Copy

```
continue → SKIP CURRENT ITERATION
pass      → DO NOTHING
```

`pass` does not stop the loop, skip the iteration, or return a value.

`break` VS `continue` VS `pass`

Statement	Meaning
<code>break</code>	Stop the loop
<code>continue</code>	Skip current iteration
<code>pass</code>	Do nothing

TEXT

Copy

```
break      → STOP
continue   → SKIP
pass       → DO NOTHING
```

Pattern Matching

`match-case`

`match-case` is Python's **structural pattern matching** feature, introduced in Python 3.10. For beginners, it can be understood as switch-style routing.

PYTHON

Copy

```
match value:
    case pattern1:
        statement
    case pattern2:
        statement
    case _:
        statement
```

Real-time example — order status processing:

PYTHON

Copy

```
order_status = "Shipped"
match order_status:
    case "Pending":
        print("Waiting for Payment")
    case "Paid":
        print("Prepare the Order")
    case "Shipped":
        print("Order is on the Way")
    case "Delivered":
        print("Order Completed")
    case "Cancelled":
        print("Order Cancelled")
    case _:
        print("Unknown Order Status")
```

Output:

TEXT

Copy

Order is on the Way

`_` is a **wildcard pattern** that matches the remaining cases (acts like a default in simple examples).

`if-elif` VS `match-case`

if-elif	match-case
Flexible conditions	Pattern matching
Range comparisons	Known patterns
Complex Boolean logic	Structural matching
<code>amount > 5000</code>	<code>case "Paid"</code>

TEXT

Copy

COMPLEX BOOLEAN / RANGE CONDITIONS → if-elif
PATTERN-BASED MATCHING → match-case

`match-case` is More Than `switch`

Do not permanently remember `match-case = switch`. Correct understanding: `match-case` → Structural Pattern Matching.

PYTHON

Copy

```
order = ("Paid", 5000)
match order:
    case ("Paid", amount):
        print("Paid Amount:", amount)
    case ("Cancelled", _):
        print("Order Cancelled")
```

Python matches the **structure of the tuple**. Beginner view: switch-style routing. Interview view: structural pattern matching.

Complete E-commerce Order Processing Example

PYTHON

Copy

```
orders = [
    ("ORD101", "Paid"),
    ("ORD102", "Cancelled"),
    ("ORD103", "Shipped"),
    ("ORD104", "Fraud"),
    ("ORD105", "Delivered")
]
for order_id, status in orders:
    if status == "Cancelled":
        print(order_id, "Skipped")
        continue
    if status == "Fraud":
        print("Fraud Detected:", order_id)
        break
    match status:
        case "Paid":
            print(order_id, "→ Prepare Order")
        case "Shipped":
            print(order_id, "→ Track Delivery")
        case "Delivered":
            print(order_id, "→ Order Completed")
        case _:
            print(order_id, "→ Unknown Status")
```

Output:

TEXT

Copy

```
ORD101 → Prepare Order
ORD102 Skipped
ORD103 → Track Delivery
Fraud Detected: ORD104
```

ORD105 is not processed because **break** stops the loop at **ORD104**. This example combines **for**, **continue**, **break**, and **match-case**.

Most Important Loop Decision

TEXT

[Copy](#)

Do I need to PROCESS ITEMS? → for
Do I need to REPEAT BASED ON A CONDITION? → while

PYTHON

[Copy](#)

```
for order in orders:  
    process(order)           # process orders → for  
  
while attempt <= max_attempts:  
    retry_payment()          # retry payment → while
```


Common Fresher Mistakes — Loops

X Mistake	✓ Correct Understanding
<code>while</code> condition never changes	Can create an infinite loop
Thinking <code>range(5)</code> includes <code>5</code>	Stop value is excluded
Confusing <code>break</code> and <code>continue</code>	<code>break</code> stops, <code>continue</code> skips
Thinking <code>pass</code> skips	<code>pass</code> does nothing
Using <code>for</code> only with lists	<code>for</code> works with iterables
Expecting dictionary values by default	Dictionary iteration gives keys
Wrong <code>range()</code> step direction	May produce no values
Using <code>while</code> for simple collection processing	Prefer <code>for</code>
Calling <code>match-case</code> only a switch	It supports structural patterns
Forgetting Python version	<code>match-case</code> requires Python 3.10+

Placement Quick Revision — Loops

Output:

TEXT

Copy

Loop → Repeats a block of code
Iteration → One execution of a loop block
for → Process items
while → Repeat based on condition
range() → Generate integer sequence
break → Stop the loop
continue → Skip current iteration
pass → Do nothing
match-case → Structural pattern matching

Interview Questions — Loops

Q1. What is a loop?

A loop repeatedly executes a block of code.

Q2. What is an iteration?

One execution of a loop block is called an iteration.

Q3. What is the difference between `for` and `while`?

`for` processes items from an iterable. `while` repeats based on a condition.

Q4. When should you use `for`?

When processing items from an iterable.

Q5. When should you use `while`?

When repetition depends on a condition.

Q6. What is an infinite loop?

A loop that never reaches its stopping condition.

Q7. What does `range(5)` generate?

Integers from `0` to `4`.

Q8. Is the stop value included in `range()` ?

No.

Q9. What is the difference between `break` and `continue` ?

`break` stops the loop. `continue` skips the current iteration.

Q10. What does `pass` do?

It does nothing and is commonly used as a placeholder.

Q11. Does `pass` skip an iteration?

No.

Q12. What does dictionary iteration process by default?

Keys.

Q13. How do you iterate through dictionary keys and values?

`for key, value in data.items(): print(key, value)`.

Q14. When was `match-case` introduced?

Python 3.10.

Q15. What is `match-case` ?

Python's structural pattern matching feature.

Q16. What does `case _` mean?

It is a wildcard pattern that matches remaining cases.

Practice — Loops

Use:

PYTHON

Copy

```
orders = [  
    ("ORD101", "Paid"),  
    ("ORD102", "Cancelled"),  
    ("ORD103", "Shipped"),  
    ("ORD104", "Fraud"),  
    ("ORD105", "Delivered")  
]
```

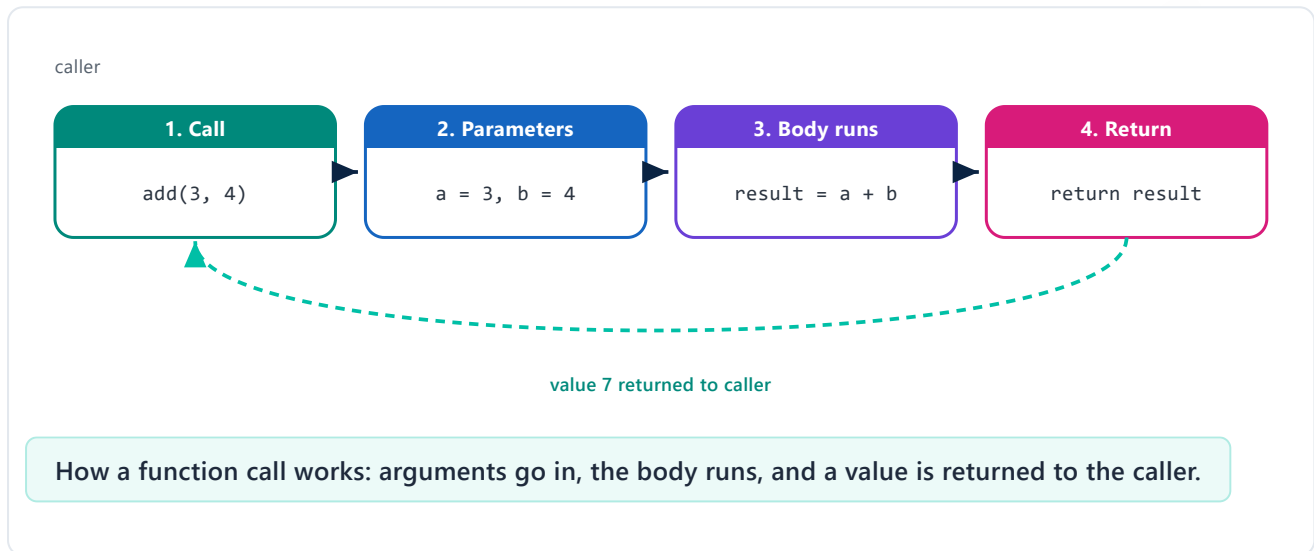
1. Print every order using a `for` loop.
2. Generate `ORD101` to `ORD105` using `range()`.
3. Create a payment retry system with three attempts using `while`.
4. Skip cancelled orders using `continue`.
5. Stop processing when fraud is detected using `break`.
6. Create an empty refund function using `pass`.
7. Process order statuses using `match-case`.
8. Explain `for` vs `while`.
9. Explain `break` vs `continue`.
10. Explain `pass` vs `continue`.
11. Predict the output of `range(5)`.
12. Predict the output of `range(1, 10, 2)`.
13. Predict the output of `range(5, 0, -1)`.
14. Find the infinite loop problem in a payment retry program.
15. Create a complete order-processing system that skips cancelled orders, stops on fraud, processes paid orders, tracks shipped orders, and completes delivered orders.

End of Level 5

Functions

Level: 6 | Version: v1.0 | Author: Meptrasoftware

A **function** is a reusable block of code designed to perform a specific task. This module uses a **Food Delivery Order System** as the running example.



PYTHON

Copy

```

def calculate_total():
    price = 200
    quantity = 2
    total = price * quantity
    print(total)

calculate_total()
  
```

Output:

TEXT

Copy

400

Defining & Calling

A function lets us write logic once and reuse it many times.

Without a function, the same calculation is repeated:

PYTHON

Copy

```
price = 200
quantity = 2
total = price * quantity
print(total)

price = 150
quantity = 3
total = price * quantity
print(total)
```

Repeated logic creates duplicate code, more maintenance, more bugs, and harder-to-read programs. Using a function:

PYTHON

Copy

```
def calculate_total(price, quantity):
    print(return price * quantity)

print(calculate_total(200, 2))
print(calculate_total(150, 3))
print(calculate_total(300, 2))
```

Output:

TEXT

Copy

```
400
450
600
```

Why Functions Matter in Real Projects

Functions are used to calculate order totals, apply discounts, validate input, process payments, send emails, fetch database records, call APIs, generate reports, and update order status. Functions organize these tasks into reusable units.

A function commonly follows this flow:

TEXT

Copy

INPUT → FUNCTION → PROCESS → OUTPUT

PYTHON

Copy

```
def calculate_total(price, quantity):  
    return price * quantity  
  
print(calculate_total(200, 2))
```

Output:

TEXT

Copy

400

Defining a Function

Creating a function is called **defining** it. Python uses the `def` keyword.

PYTHON

Copy

```
def function_name():  
    statement
```

Example:

PYTHON

Copy

```
def show_order():  
    print("Order Received")
```

Output:

TEXT

Copy

def	→ keyword used to define a function
show_order	→ function name
()	→ parameters are written here
:	→ starts the function block
indented	→ function body

Defining Does Not Execute

PYTHON

Copy

```
def show_order():  
    print("Order Received")
```

Output: **no output**. The function was only **defined**, not executed.

TEXT

Copy

def → DEFINE / CREATE FUNCTION (not execute)

Calling a Function

Executing a function is called **calling** it — the function name followed by parentheses.

PYTHON

Copy

```
def show_order():  
    print("Order Received")  
  
show_order()
```

Output:

TEXT

Copy

Order Received

How calling works:

PYTHON

Copy

```
def show_order():
    print("Order Received")

print("Start")
show_order()
print("End")
```

Output:

TEXT

Copy

```
Start
Order Received
End
```

TEXT

Copy

Program → show_order() → Execute Function Body → Return to Calling Point

Defining vs Calling

Defining	Calling
Creates a function	Executes a function
Uses <code>def</code>	Uses function name with <code>()</code>
Writes function logic	Runs function logic

TEXT

Copy

```
def → DEFINE      () → CALL
DEFINE ≠ EXECUTE  CALL → EXECUTE
```

Parameters and Arguments

PYTHON

Copy

```
def calculate_total(price, quantity):
    return price * quantity

print(calculate_total(200, 2))
```

A **parameter** is a name written in the function definition (`price`, `quantity`). An **argument** is the actual value passed during a call (`200`, `2`).

Output:

TEXT

Copy

400

Parameter	Argument
Name in function definition	Actual value passed to function
Receives data	Sends data
<code>price</code>	<code>200</code>
<code>quantity</code>	<code>2</code>

TEXT

Copy

PARAMETER → RECEIVES VALUE
ARGUMENT → SENDS VALUE

Inside the call `calculate_total(200, 2)`: `price = 200`, `quantity = 2`, so `200 × 2 = 400`.

Argument Types

Python supports different ways to pass arguments and define parameters.

Feature	Purpose
Positional Arguments	Match values by position
Keyword Arguments	Match values by parameter name
Default Parameter Values	Use a predefined value when omitted
<code>*args</code>	Accept extra positional arguments
<code>**kwargs</code>	Accept extra keyword arguments

Positional Arguments

Positional arguments are matched based on their **position**.

PYTHON

Copy

```
def place_order(food, quantity):  
    print("Food:", food)  
    print("Quantity:", quantity)  
  
place_order("Burger", 2)
```

Output:

TEXT

Copy

```
Food: Burger  
Quantity: 2
```

Python matches the 1st argument to the 1st parameter, and so on. If order is wrong, values are still matched by position:

PYTHON

Copy

```
def place_order(food, quantity):
    place_order(2, "Burger")    # food ← 2, quantity ← "Burger"
```

Output:

TEXT

Copy

POSITIONAL ARGUMENT → POSITION MATTERS

Keyword Arguments

Keyword arguments pass values using parameter names, so order can change.

PYTHON

Copy

```
def place_order(food, quantity):
    print("Food:", food)
    print("Quantity:", quantity)

place_order(food="Burger", quantity=2)
place_order(quantity=2, food="Burger")
```

Both print the same result.

Positional	Keyword
Matched by position	Matched by parameter name
Position matters	Keyword order does not matter
Shorter	More explicit

Output:

TEXT

Copy

```
Food: Burger
Quantity: 2
Food: Burger
Quantity: 2
```

Default Arguments

A parameter can have a predefined default value, used when the caller omits it.

PYTHON

Copy

```
def calculate_bill(price, delivery_fee=40):
    return price + delivery_fee

print(calculate_bill(200))      # 240 (delivery_fee defaults to 40)
print(calculate_bill(200, 60)) # 260 (default overridden)
```

Default values are useful for delivery fees, tax rates, retry counts, user roles, page sizes, and timeouts.

For beginners, use simple immutable default values such as numbers, strings, `True`, `False`, or `None`. Avoid mutable default values such as `[]` or `{}`; they can keep changes between function calls.

Parameter Order Rule

Parameters without default values must come before parameters with defaults.

Correct:

PYTHON

Copy

```
def calculate_bill(price, delivery_fee=40):
    pass
```

Wrong (causes `SyntaxError`):

PYTHON

Copy

```
def calculate_bill(delivery_fee=40, price):  
    pass
```

Output:

TEXT

Copy

REQUIRED PARAMETER → FIRST
DEFAULT PARAMETER → AFTER

Variable-Length Positional Arguments — `*args`

`*args` allows a function to accept a variable number of positional arguments, collected into a **tuple**.

PYTHON

Copy

```
def calculate_order(*prices):  
    print(prices)  
  
calculate_order(200, 150, 300)    # (200, 150, 300)
```

Sum multiple prices:

PYTHON

Copy

```
def calculate_order(*prices):  
    return sum(prices)  
  
print(calculate_order(200))  
print(calculate_order(200, 150))  
print(calculate_order(200, 150, 300))
```

The same function accepts different numbers of arguments — 1, 5, 10, or 100 prices. The name `args` is only a convention; the `*` creates the behavior.

Output:

TEXT

Copy

```
200
350
650
```

Variable-Length Keyword Arguments — `**kwargs`

`**kwargs` allows a function to accept a variable number of keyword arguments, collected into a **dictionary**.

PYTHON

Copy

```
def create_order(**details):
    print(details)

create_order(customer="Arun", food="Burger", quantity=2)
```

Output:

TEXT

Copy

```
{'customer': 'Arun', 'food': 'Burger', 'quantity': 2}
```

Access values and iterate:

PYTHON

Copy

```
def create_order(**details):
    for key, value in details.items():
        print(key, ":", value)

create_order(
    customer="Arun",
    food="Burger",
    quantity=2,
    payment="UPI",
    coupon="SAVE50"
)
```

`**kwargs` is useful for user profiles, API parameters, configuration settings, database filters, product details, and order metadata.

Output:

TEXT

Copy

```
customer : Arun
food : Burger
quantity : 2
payment : UPI
coupon : SAVE50
```

args** VS *kwargs**

*args	**kwargs
Extra positional arguments	Extra keyword arguments
Collected into a tuple	Collected into a dictionary
Values	Key-value pairs
<code>function(10, 20)</code>	<code>function(name="Arun")</code>
*	**

TEXT

Copy

```
*args    → VALUES    → TUPLE
**kwargs → KEY=VALUE  → DICTIONARY
```

Scope & Return

Return Values

The `return` statement sends a result from a function back to the caller.

PYTHON

Copy

```
def calculate_total(price, quantity):
    return price * quantity

total = calculate_total(200, 2)
print(total)    # 400
```

The returned value can be stored, reused, compared, or passed to another function:

PYTHON

Copy

```
def calculate_total(price, quantity):
    return price * quantity

total = calculate_total(200, 2)
final_bill = total + 40
print(final_bill)
```

Output:

TEXT

Copy

440

`print()` vs `return`

`print()` only displays a value; `return` sends the value back to the caller.

<code>print()</code>	<code>return</code>
Displays a value	Gives a value back
Used to show output	Used in program logic
Mainly for display or debugging	Result can be stored
Does not return the displayed value	Sends a result to the caller

TEXT

Copy

print → SHOW VALUE
return → GIVE VALUE BACK

return Stops the Function

PYTHON

Copy

```
def calculate_total():  
    return 400  
    print("Completed")  # unreachable
```

Code after an unconditional **return** in the same execution path is unreachable.

Output:

TEXT

Copy

return → FUNCTION CALL ENDS

Function Without Explicit **return**

PYTHON

Copy

```
def show_order():  
    print("Order Received")  
  
result = show_order()  
print(result)
```

Output:

TEXT

Copy

Order Received
None

With no explicit **return**, Python returns **None**.

Variable Scope

Scope defines where a name can be accessed or resolved. The two basic scopes are **local** and **global**.

A variable created inside a function is **local**:

PYTHON

Copy

```
def calculate_total():
    total = 400
    print(total)
```

Accessing `total` outside raises `NameError`, because it was created inside the function (local scope).

A variable created at module level is **global**:

PYTHON

Copy

```
delivery_fee = 40

def show_fee():
    print(delivery_fee)

show_fee() # 40
```

Local	Global
Created inside a function	Created at module level
Belongs to local scope	Belongs to global/module scope
Common for function data	Common for module-level names

Output:

TEXT

Copy

40

Name Resolution — LEGB

Python resolves names using the **LEGB rule**: Local → Enclosing → Global → Built-in.

PYTHON

Copy

```
price = 200

def show_price():
    price = 300
    print(price)

show_price()  # 300
```

Python finds the local `price` first.

Output:

TEXT

Copy

300

`global` Keyword

The `global` keyword allows a function to assign to a module-level variable.

PYTHON

Copy

```
order_count = 0

def create_order():
    global order_count
    order_count += 1

create_order()
print(order_count)  # 1
```

Without `global`, assigning to `order_count` inside the function raises `UnboundLocalError`, because Python treats an assigned name as local unless declared `global`.

Do not overuse global state — it creates hidden dependencies and hard-to-debug programs. Prefer parameters and return values:

PYTHON

Copy

```
def increase_order_count(order_count):  
    return order_count + 1  
  
order_count = 0  
order_count = increase_order_count(order_count)  
print(order_count)
```

Output:

TEXT

Copy

1

Functional Tools

Lambda and Anonymous Functions

A lambda expression creates a small function using a single expression.

PYTHON

Copy

```
lambda parameters: expression
```

Example:

PYTHON

Copy

```
calculate_total = lambda price, quantity: price * quantity  
print(calculate_total(200, 2))    # 400
```

Normal function vs lambda — both compute the same result:

PYTHON

Copy

```
def calculate_total(price, quantity):  
    return price * quantity  
  
calculate_total = lambda price, quantity: price * quantity
```

An **anonymous function** is used without a normal `def`-based name and can be passed directly to another function:

PYTHON

Copy

```
prices = [100, 200, 300]  
result = map(lambda price: price * 1.05, prices)
```

Use lambda for small functions, simple transformations, sorting keys, `map()`, `filter()`, and short callbacks. Avoid it for complex logic:

PYTHON

Copy

```
process = lambda x: "High" if x > 1000 else "Medium" if x > 500 else "Low"
```

A normal function is clearer:

PYTHON

Copy

```
def classify_price(price):  
    if price > 1000:  
        return "High"  
    if price > 500:  
        return "Medium"  
    print(return "Low")
```

Output:

TEXT

Copy

```
SMALL ONE-EXPRESSION FUNCTION → lambda  
COMPLEX LOGIC → def
```

Recursion

Recursion happens when a function calls itself. It needs a stopping condition.

PYTHON

Copy

```
def process_countdown(count):  
    if count == 0:  
        print("Order Processing Started")  
        return  
    print(count)  
    process_countdown(count - 1)  
  
process_countdown(3)
```

Output:

TEXT

Copy

```
3  
2  
1  
Order Processing Started
```

The condition that stops recursion is the **base case**; the part that calls the function again is the **recursive case**.

PYTHON

Copy

```
if count == 0:                # base case → STOP  
    return  
process_countdown(count - 1)  # recursive case → CALL AGAIN
```

Recursion without a base case eventually raises `RecursionError`:

PYTHON

Copy

```
def process():  
    process()
```

Factorial using recursion:

PYTHON

Copy

```
def factorial(number):  
    if number <= 1:  
        return 1  
    return number * factorial(number - 1)  
  
print(factorial(5))    # 120
```

Recursion is useful for naturally recursive problems — tree traversal, folder traversal, graph algorithms, divide-and-conquer, and recursive data structures.

Output:

TEXT

Copy

120

Built-in Functions for Data Processing

Python provides several built-in functions for processing iterable data.

TEXT

Copy

map	→ TRANSFORM
filter	→ SELECT
zip	→ COMBINE
enumerate	→ COUNTER + ITEM

map()

map() applies a function to every item in an iterable.

PYTHON

Copy

```
prices = [100, 200, 300]  
updated_prices = list(map(lambda price: price * 1.05, prices))  
print(updated_prices)    # [105.0, 210.0, 315.0]
```


PYTHON

Copy

```
map(function, iterable)
```

In Python 3, `map()` returns a **map iterator**, not a list — convert with `list()`:

PYTHON

Copy

```
prices = ["100", "200", "300"]
prices = list(map(int, prices))
print(prices)    # [100, 200, 300]
```

Uses: update every price, convert strings to lowercase, convert string numbers to integers, transform API values, normalize data.

Output:

TEXT

Copy

```
[100, 200, 300]
```

`filter()`

`filter()` keeps items that satisfy a condition.

PYTHON

Copy

```
prices = [100, 500, 50, 800, 200]
selected_prices = list(filter(lambda price: price >= 200, prices))
print(selected_prices)    # [500, 800, 200]
```

PYTHON

Copy

```
numbers = [10, 15, 20, 25]
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(even_numbers)
```

Output:

TEXT

Copy

```
[10, 20]
```

Like `map()`, `filter()` returns a **filter iterator** in Python 3 — convert with `list()`. Uses: select available products, active users, passed students, valid records, expensive products.

Output:

TEXT

Copy

```
filter() → KEEP MATCHING ITEMS
```

`map()` VS `filter()`

<code>map()</code>	<code>filter()</code>
Transforms items	Selects items
Applies logic to every item	Keeps matching items
Usually changes values	Usually reduces items

TEXT

Copy

```
map    → CHANGE / TRANSFORM
filter → CHECK / SELECT
```

`zip()`

`zip()` combines items from multiple iterables by position.

PYTHON

Copy

```
foods = ["Burger", "Pizza", "Pasta"]
prices = [200, 300, 250]

menu = zip(foods, prices)
print(list(menu))
# [('Burger', 200), ('Pizza', 300), ('Pasta', 250)]

for food, price in zip(foods, prices):
    print(food, "₹", price)
```

When lengths differ, normal `zip()` stops at the shortest iterable:

PYTHON

Copy

```
foods = ["Burger", "Pizza", "Pasta"]
prices = [200, 300]
print(list(zip(foods, prices)))
# [('Burger', 200), ('Pizza', 300)] ← "Pasta" excluded
```

Output:

TEXT

Copy

```
[('Burger', 200), ('Pizza', 300)]
```

Modern Python supports `strict=True`, which raises `ValueError` when lengths differ. This is available in Python 3.10 and later, and is useful when unequal lengths indicate a data problem:

PYTHON

Copy

```
colors = ["red", "blue", "green"]
prices = [100, 200, 350]
foods = ["Burger", "Pizza", "Pasta"]
foods = ["Burger", "Pizza", "Pasta"]
menu = zip(foods, prices, strict=True) # ValueError if Lengths differ
```

`enumerate()`

`enumerate()` provides a counter and an item while iterating.

PYTHON

Copy

```
foods = ["Burger", "Pizza", "Pasta"]
for number, food in enumerate(foods):
    print(number, food)
# 0 Burger / 1 Pizza / 2 Pasta
```

The counter starts at `0` by default; use `start=1` for user-facing lists:

PYTHON

Copy

```
foods = ["Burger", "Pizza", "Pasta"]
for number, food in enumerate(foods, start=1):
    print(f"{number}. {food}")
# 1. Burger / 2. Pizza / 3. Pasta
```

When both counter/index and item are needed, `enumerate()` is clearer than `range(len())`.

Output:

TEXT

Copy

```
1. Burger
2. Pizza
3. Pasta
```

Processing Functions Comparison

Function	Main Question
<code>map()</code>	How do I transform every item?
<code>filter()</code>	How do I keep matching items?
<code>zip()</code>	How do I combine iterables by position?
<code>enumerate()</code>	How do I get a counter and item?

TEXT

Copy

```
map      → TRANSFORM
filter   → SELECT
zip      → COMBINE
enumerate → COUNTER + ITEM
```

Built-in vs User-defined

Built-in vs User-Defined Functions

Built-in functions are already provided by Python:

PYTHON

Copy

```
print(); len(); sum(); max(); min(); type(); input()
range(); map(); filter(); zip(); enumerate()
```

PYTHON

Copy

```
prices = [100, 200, 300]
print(sum(prices))    # 600 - sum() is built-in
```

Functions created by programmers are user-defined:

PYTHON

Copy

```
def calculate_total(price, quantity):
    return price * quantity
```

Built-in Function	User-Defined Function
Provided by Python	Created by programmer
Ready to use	Must be defined
<code>print()</code>	<code>calculate_total()</code>
<code>len()</code>	<code>process_order()</code>
<code>sum()</code>	<code>apply_discount()</code>

Output:

TEXT

Copy

PYTHON PROVIDED IT → BUILT-IN FUNCTION
 PROGRAMMER CREATED IT → USER-DEFINED FUNCTION

Complete Food Delivery Billing System

PYTHON

Copy

```
def calculate_food_total(*prices):  
    return sum(prices)  
  
def apply_discount(total, discount=10):  
    discount_amount = total * discount / 100  
    return total - discount_amount  
  
def add_delivery_fee(total, delivery_fee=40):  
    return total + delivery_fee  
  
def show_order(**details):  
    print("\nOrder Details")  
    for key, value in details.items():  
        print(key, ":", value)  
  
food_total = calculate_food_total(200, 300, 150)  
discounted_total = apply_discount(food_total)  
final_bill = add_delivery_fee(discounted_total)  
  
show_order(  
    customer="Arun",  
    payment="UPI",  
    status="Confirmed"  
)  
  
print("Food Total:", food_total)  
print("After Discount:", discounted_total)  
print("Final Bill:", final_bill)
```

Output:

TEXT

Copy

```
Order Details  
customer : Arun  
payment : UPI  
status : Confirmed  
Food Total: 650  
After Discount: 585.0  
Final Bill: 625.0
```

Flow:

TEXT

Copy

```
200, 300, 150 → calculate_food_total() → 650
               → apply_discount()      → 585
               → add_delivery_fee()     → 625 (FINAL BILL)
```

Each function performs one clear responsibility.

Function Design in Real Projects

Beginners often create one large function responsible for many tasks:

PYTHON

Copy

```
def process_order():
    # calculate food total
    # apply discount
    # calculate tax
    # add delivery fee
    # process payment
    # update database
    # send email
print(pass)
```

Output:

TEXT

Copy

```
ONE LARGE FUNCTION → TOO MANY RESPONSIBILITIES
→ HARDER TO UNDERSTAND / TEST / DEBUG
```

Cleaner design — one function, one clear job:

PYTHON

Copy

```
def calculate_total():  
    pass  
  
def apply_discount():  
    pass  
  
def process_payment():  
    pass  
  
def update_order():  
    pass  
  
def send_notification():  
    print(pass)
```

Small, focused functions are easier to understand, test, debug, reuse, and maintain. The goal is a **clear responsibility** — do not split functions mechanically into tiny pieces that provide no design value.

Output:

TEXT

Copy

ONE FUNCTION → ONE CLEAR JOB

Common Fresher Mistakes



✗ Mistake	✓ Correct Understanding
Defining a function but not calling it	Call the function using <code>()</code>
Thinking definition executes the function	Definition creates; calling executes
Confusing parameters and arguments	Parameter receives; argument sends
Wrong positional argument order	Position matters
Default parameter before required parameter	Required parameters come first
Using <code>[]</code> or <code>{}</code> as a default value	Use <code>None</code> and create the collection inside the function
Thinking <code>*args</code> is a list	<code>*args</code> is collected into a tuple
Thinking <code>**kwargs</code> is a tuple	<code>**kwargs</code> is collected into a dictionary
Using <code>print()</code> instead of <code>return</code> for reusable results	Use <code>return</code>
Writing code after unconditional <code>return</code>	<code>return</code> ends that function path
Accessing a local variable outside its scope	Understand local scope
Overusing <code>global</code>	Prefer parameters and return values
Recursion without a base case	Add a stopping condition
Thinking <code>map()</code> selects items	<code>map()</code> transforms
Thinking <code>filter()</code> transforms items	<code>filter()</code> selects
Thinking <code>map()</code> directly returns a list	It returns a map iterator
Thinking <code>filter()</code> directly returns a list	It returns a filter iterator

Placement Quick Revision

TEXT

[Copy](#)

FUNCTION → Reusable block of code
def → Define a function
() → Call a function
PARAMETER → Receives a value
ARGUMENT → Sends a value
return → Gives a value back
print → Shows a value
POSITIONAL → Position matters
KEYWORD → Parameter name
DEFAULT → Fallback value
*args → Extra positional arguments → Tuple
**kwargs → Extra keyword arguments → Dictionary
LOCAL → Function scope
GLOBAL → Module scope
LEGB → Local → Enclosing → Global → Built-in
global → Assign to module-level variable
lambda → Small one-expression function
RECURSION → Function calls itself
BASE CASE → Stops recursion
map → Transform
filter → Select
zip → Combine by position
enumerate → Counter + item

Interview Questions

Q1. What is a function?

A function is a reusable block of code designed to perform a specific task.

Q2. What does `def` do?

`def` is used to define a function.

Q3. What is function calling?

Executing a function using its name and parentheses.

Q4. What is the difference between defining and calling a function?

Defining creates the function. Calling executes the function.

Q5. What is a parameter?

A parameter is a name in a function definition that receives a value.

Q6. What is an argument?

An argument is an actual value passed during a function call.

Q7. What is the difference between a parameter and an argument?

A parameter receives data in the function definition. An argument sends an actual value during the call.

Q8. What does `return` do?

`return` sends a result from a function back to the caller.

Q9. What is the difference between `print()` and `return`?

`print()` displays a value. `return` gives a value back to the caller.

Q10. What does a function return without an explicit `return`?

`None`.

Q11. Does `return` stop a function?

Yes. It ends the current function call when execution reaches it.

Q12. What are positional arguments?

Arguments matched to parameters based on position.

Q13. What are keyword arguments?

Arguments passed using parameter names.

Q14.What is a default parameter value?

A predefined value used when the caller omits the corresponding argument.

Q15.What is `*args` ?

It collects extra positional arguments into a tuple.

Q16.Is the name `args` mandatory?

No. The `*` creates variable positional argument behavior; `args` is a naming convention.

Q17.What is `**kwargs` ?

It collects extra keyword arguments into a dictionary.

Q18.What is the difference between `*args` and `**kwargs` ?

`*args` collects extra positional arguments into a tuple. `**kwargs` collects extra keyword arguments into a dictionary.

Q19.What is scope?

Scope defines where a name can be accessed or resolved.

Q20.What is a local variable?

A variable created inside a function and belonging to its local scope.

Q21.What is the LEGB rule?

Local, Enclosing, Global, Built-in.

Q22.What does `global` do?

It allows assignment to a module-level variable from inside a function.

Q23.What is a lambda function?

A small function created using a lambda expression containing one expression.

Q24. What is an anonymous function?

A function used without a normal `def`-based name.

Q25. What is recursion?

Recursion occurs when a function calls itself.

Q26. What is a base case?

The condition that stops recursion.

Q27. What is a recursive case?

The part that calls the function again.

Q28. What does `map()` do?

It applies a function to every item in an iterable.

Q29. Does `map()` directly return a list in Python 3?

No. It returns a map iterator.

Q30. What does `filter()` do?

It keeps items that satisfy a condition.

Q31. What is the difference between `map()` and `filter()`?

`map()` transforms items. `filter()` selects items.

Q32. What does `zip()` do?

It combines items from multiple iterables by position.

Q33. What happens when normal `zip()` receives iterables of different lengths?

It stops when the shortest iterable is exhausted.

Q34. What does `enumerate()` do?

It provides a counter and item while iterating.

Q35. What is the difference between built-in and user-defined functions?

Built-in functions are provided by Python. User-defined functions are created by programmers.

Practice

Use a Food Delivery Order System.

1. Create a function named `show_order()`.
2. Define and call `show_order()`.
3. Create `calculate_total(price, quantity)`.
4. Identify `price` and `quantity` as parameters.
5. Pass `200` and `2` as arguments.
6. Return the calculated total.
7. Store the returned total in a variable.
8. Explain `print()` vs `return`.
9. Call a function using positional arguments.
10. Call a function using keyword arguments.
11. Explain why positional order matters.
12. Add a default delivery fee of `40`.
13. Override the default delivery fee.
14. Create a function accepting multiple prices using `*args`.
15. Calculate the sum of all `*args` values.
16. Check the type of `args`.
17. Create an order-details function using `**kwargs`.
18. Display every `**kwargs` key and value.
19. Check the type of `kwargs`.
20. Explain `*args` vs `**kwargs`.
21. Create a local variable inside a function.
22. Try accessing the local variable outside the function.
23. Explain the LEGB rule.
24. Update a module-level order count using `global`.
25. Rewrite the order counter using parameters and `return`.
26. Create a lambda to add 5% tax.

27. Explain when lambda should be avoided.
28. Create a recursive order countdown.
29. Identify the base case.
30. Identify the recursive case.
31. Create a recursive factorial function.
32. Explain why recursion without a base case is dangerous.
33. Add 5% tax to every price using `map()`.
34. Convert string prices to integers using `map()`.
35. Select prices greater than or equal to ₹200 using `filter()`.
36. Explain `map()` vs `filter()`.
37. Combine food names and prices using `zip()`.
38. Test `zip()` with unequal iterable lengths.
39. Display menu numbers using `enumerate()`.
40. Start `enumerate()` from `1`.
41. Rewrite a `range(len())` loop using `enumerate()`.
42. Explain built-in vs user-defined functions.

Predict the output:

PYTHON

Copy

```
def calculate(price, quantity):
    return price * quantity
print(calculate(200, 2))
```

PYTHON

Copy

```
def show():
    print("Order")
result = show()
print(result)
```

PYTHON

Copy

```
def show(*args):
    print(type(args))
show(10, 20, 30)
```

PYTHON

Copy

```
def show(**kwargs):
    print(type(kwargs))
show(name="Arun")
```

PYTHON

Copy

```
def calculate():  
    return 400  
    print("Completed")  
print(calculate())
```

Find the mistake:

PYTHON

Copy

```
def calculate(delivery_fee=40, price):  
    return price + delivery_fee
```

PYTHON

Copy

```
def process():  
    process()
```

PYTHON

Copy

```
def calculate_total():  
    total = 400  
calculate_total()  
print(total)
```

Final task: Create a complete Food Delivery Billing System using separate functions for food total, discount, delivery fee, order details, and final bill. Use parameters, arguments, `return`, default values, `*args`, and `**kwargs`.

Output:

TEXT

Copy

ONE FUNCTION → ONE CLEAR JOB

End of Level 6

Modules & File Handling

Level: 7 | Version: v1.0 | Author: Meptrasoft

Modules & Packages

A **module** is a `.py` file containing reusable code. A **package** is usually a folder of modules. Together they let us split a project into organized, reusable files.

import Statement

`import` loads a module. Access its members with `module.name`.

PYTHON

Copy

```
import math
print(math.sqrt(16))    # 4.0
```

from...import

Import specific names directly (no module prefix needed).

PYTHON

Copy

```
from math import sqrt, pi
print(sqrt(16))    # 4.0
print(pi)          # 3.14159...
```

Aliasing Modules

Give a module a shorter name using `as`.

PYTHON

Copy

```
import datetime as dt
print(dt.date.today())
```

Python Standard Library

Modules that ship with Python — no installation needed. Common ones:

Module	Purpose
<code>math</code>	Math functions
<code>random</code>	Random numbers
<code>datetime</code>	Dates and times
<code>os</code>	Operating-system paths
<code>json</code>	JSON encode/decode
<code>csv</code>	Read/write CSV files

pip and Installing Packages

`pip` installs third-party packages from PyPI (not in the standard library).

BASH

Copy

```
pip install requests
```

PYTHON

Copy

```
import requests
```

Creating Your Own Module

Any `.py` file you write is a module. Put reusable functions in one file, then import it from another file in the same folder.

`calculator.py` — your module:

PYTHON

Copy

```
# calculator.py
def add(a, b):
    return a + b

def subtract(a, b):
    return a - b
```

`main.py` — uses the module:

PYTHON

Copy

```
# main.py
import calculator

print(calculator.add(2, 3))      # 5
print(calculator.subtract(5, 1)) # 4
```

Import specific functions with `from`:

PYTHON

Copy

```
from calculator import add, subtract

print(add(2, 3))      # 5 (no calculator. prefix)
```

Python finds `calculator` because it sits in the same folder as `main.py`.

The `if __name__ == "__main__":` Guard

When Python runs a file directly, it sets that file's `__name__` to `"__main__"`. When the file is imported, `__name__` is the module's name (e.g. `"calculator"`) instead. This guard runs code only when the file is run directly, not when it is imported.

PYTHON

Copy

```
# calculator.py
def add(a, b):
    return a + b

if __name__ == "__main__":
    # runs only when you do: python calculator.py
    print("Testing:", add(2, 3))
```

Now `import calculator` gives you the `add` function **without** running the test line. Always wrap a module's test/demo code in this guard so importing it does not trigger unwanted output.

Packages

A **package** is usually a folder of modules. For beginner projects, add an `__init__.py` file inside the folder. It can be empty, and it clearly marks the folder as a package.

TEXT

Copy

```
mathtools/
  __init__.py
  calculator.py
```

Import a module from the package using dot notation:

PYTHON

Copy

```
import mathtools.calculator
print(mathtools.calculator.add(2, 3))

from mathtools.calculator import add
print(add(2, 3))
```

File Handling

Opening Files and File Modes

Use `open(path, mode)` to open a file. The mode decides what you can do.

Mode	Meaning
"r"	Read (default); error if file is missing
"w"	Write; creates/overwrites the file
"a"	Append; adds to the end
"x"	Create; error if file already exists
"b"	Binary (combine, e.g. "rb")

Reading Files

When working with text files, pass `encoding="utf-8"`. Without it, Python uses the operating system's default encoding (on Windows this is often not UTF-8), so accented letters, emojis, and other non-English characters can read as garbage or raise a `UnicodeDecodeError`. Being explicit keeps files portable across machines.

PYTHON

Copy

```
with open("data.txt", "r", encoding="utf-8") as f:  
    content = f.read()      # entire file as one string
```

`readline()` reads one line; `readlines()` returns a list of lines. You can also loop directly:

PYTHON

Copy

```
with open("data.txt", "r", encoding="utf-8") as f:
    for line in f:
        print(line.strip())
```

Writing Files

"w" overwrites existing content.

PYTHON

Copy

```
with open("data.txt", "w", encoding="utf-8") as f:
    f.write("Hello\n")
```

Appending to Files

"a" keeps existing content and adds to the end.

PYTHON

Copy

```
with open("data.txt", "a", encoding="utf-8") as f:
    f.write("New line\n")
```

with Statement

With opens a file and closes it automatically — the recommended way.

PYTHON

Copy

```
with open("data.txt", "r", encoding="utf-8") as f:
    content = f.read()
# file is closed automatically here
```

No manual `close()` needed, even if an error occurs.

CSV Files

Use the `csv` module for comma-separated data.

PYTHON

Copy

```
import csv

with open("students.csv", "r", encoding="utf-8", newline="") as f:
    reader = csv.reader(f)
    for row in reader:
        print(row)    # each row is a list

with open("out.csv", "w", encoding="utf-8", newline="") as f:
    writer = csv.writer(f)
    writer.writerow(["Name", "CGPA"])
    writer.writerow(["Arun", 8.5])
```

JSON Files

Use the `json` module to convert between JSON text and Python objects.

PYTHON

Copy

```
import json

data = {"name": "Arun", "cgpa": 8.5}

with open("student.json", "w", encoding="utf-8") as f:
    json.dump(data, f)    # write dict → JSON

with open("student.json", "r", encoding="utf-8") as f:
    loaded = json.load(f)    # read JSON → dict
```

`json.dumps()` / `json.loads()` work with strings instead of files.

Working with File Paths

Use `os.path` (or `pathlib`) for safe, cross-platform paths — do not hardcode `\` or `/`.

PYTHON

Copy

```
import os
path = os.path.join("data", "students.csv") # "data/students.csv" or "data\\students.csv"
print(os.path.exists(path))
```

Common Fresher Mistakes

X Mistake	Problem	✓ Correct Understanding
Naming your file <code>math.py</code> , <code>random.py</code> , etc.	Shadows the standard-library module	Give modules unique names
Forgetting the <code>if __name__ == "__main__":</code> guard	Test code runs on import	Wrap demo/test code in the guard
Folder missing <code>__init__.py</code>	Package setup may confuse beginners	Add an <code>__init__.py</code> file for clarity
Opening a file without <code>encoding="utf-8"</code>	Garbled text / <code>UnicodeDecodeError</code>	Always pass <code>encoding="utf-8"</code>
Using <code>open()</code> without <code>close()</code> or <code>with</code>	File may stay open / data not saved	Use <code>with open(...)</code>
Expecting <code>"w"</code> mode to keep old content	<code>"w"</code> overwrites the whole file	Use <code>"a"</code> to append

Quick Points

- A module is a `.py` file; a package is usually a folder of modules.
- `import module` → use `module.name`; `from module import name` → use `name` directly.
- `as` creates an alias.
- Standard library ships with Python; `pip` installs external packages.

- File modes: `r` read, `w` overwrite, `a` append, `x` create.
- `with open(...)` closes the file automatically — prefer it.
- Use `csv` for CSV, `json` for JSON, `os.path`/`pathlib` for paths.

Interview Questions

Q1. What is the difference between a module and a package?

A module is a single `.py` file. A package is usually a folder containing modules.

Q2. What is the difference between `import module` and `from module import name`?

`import module` requires `module.name`. `from module import name` uses `name` directly.

Q3. Why use the `with` statement for files?

It closes the file automatically, even if an error occurs.

Q4. What is the difference between `"w"` and `"a"` modes?

`"w"` overwrites the file. `"a"` appends to the end.

Q5. What is `pip`?

A tool that installs third-party packages from PyPI.

Q6. What does `if __name__ == "__main__":` do?

It runs the code inside it only when the file is run directly, not when it is imported.

Q7. What is the beginner-friendly way to make a folder a package?

Add an `__init__.py` file to the folder.

Q8. Why pass `encoding="utf-8"` when opening a text file?

It reads/writes text consistently across systems and avoids errors with non-English characters.

Practice

1. Create a module `calculator.py` with `add` and `subtract` functions.
2. In another file, `import calculator` and call `calculator.add(2, 3)`.
3. Import just `add` using `from calculator import add`.
4. Give `calculator` an alias with `import calculator as calc` and use it.
5. Add an `if __name__ == "__main__":` block to `calculator.py` that prints a test result, then confirm it does **not** run when imported.
6. Create a folder `mathtools/` with an `__init__.py` and move `calculator.py` into it; import it as `from mathtools.calculator import add`.
7. Open a file with `encoding="utf-8"` and write a line containing an accented character.
8. Read a file line by line using a `for` loop.
9. Append a new line to an existing file using `"a"` mode.
10. Rewrite a manual `open()` / `close()` example using `with`.
11. Write a dictionary to a JSON file and read it back with the `json` module.
12. Build a cross-platform path with `os.path.join` and check if it exists.

End of Level 7

Error Handling

Level: 8 | Version: v1.0 | Author: Meptrasoft

Programs receive bad input, missing files, and failed network calls. Error handling lets a program **respond to problems instead of crashing**. This module uses a **Payment Processing System** as the running example.

Exceptions Basics

An **exception** is an error that occurs **while a program is running**. When an exception occurs and is not handled, the program stops and shows an error message called a **traceback**.

PYTHON

Copy

```
amount = 100  
result = amount / 0
```

Output:

TEXT

Copy

```
ZeroDivisionError: division by zero
```

The program stops normal execution at that line. The lines after it do not run.

Error vs Exception

TEXT

Copy

```
Syntax Error → Code is written incorrectly (caught before running)  
Exception    → Code is valid but fails while running
```

Syntax error (program does not even start):

PYTHON

Copy

```
if amount > 0
    print("Valid")
```

Exception (program starts, then fails):

PYTHON

Copy

```
amount = int("abc") # ValueError at runtime
```

Common Exceptions

Exception	When It Happens
<code>ZeroDivisionError</code>	Dividing by zero
<code>ValueError</code>	Right type, wrong value (<code>int("abc")</code>)
<code>TypeError</code>	Wrong type used in an operation
<code>IndexError</code>	List index out of range
<code>KeyError</code>	Dictionary key does not exist
<code>FileNotFoundError</code>	Opening a file that does not exist
<code>NameError</code>	Using a variable that was never defined
<code>AttributeError</code>	Using a method/attribute an object lacks

PYTHON

Copy

```
numbers = [10, 20, 30]
print(numbers[5])           # IndexError

student = {"name": "Arun"}
print(student["cgpa"])      # KeyError
```

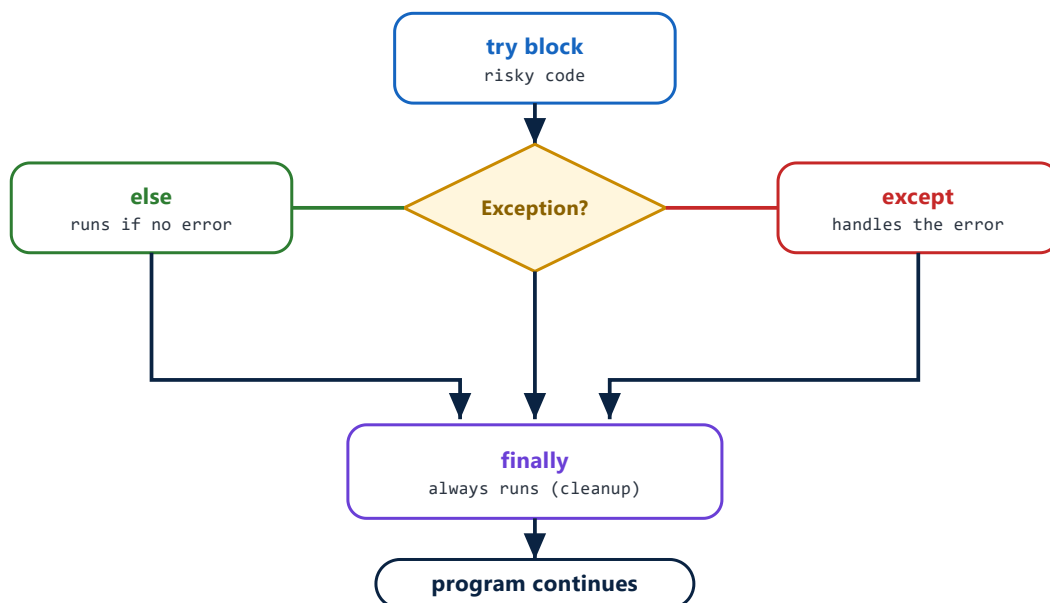
Output:

TEXT

Copy

EXCEPTION → RUNTIME ERROR → PROGRAM STOPS (unless handled)

try / except Blocks



Control flow through try / except / else / finally. finally always runs, whether or not an exception occurred.

try

PYTHON

Copy

```
try:
    amount = int(input("Enter amount: "))
    print("Amount:", amount)
```

If the risky line fails, Python stops running the `try` block and jumps to `except` instead of crashing.

`except`

The `except` block runs **only** when an exception occurs in the `try` block.

PYTHON

Copy

```
try:
    amount = int("abc")
except ValueError:
    print("Invalid amount entered")
```

Output:

TEXT

Copy

```
Invalid amount entered
```

The program continues normally after handling the error.

Catching Specific Exceptions

Handle different errors differently by naming each one:

PYTHON

Copy

```
amount = int("abc")
try:
    result = amount / divisor
except ZeroDivisionError:
    print("Cannot divide by zero")
except ValueError:
    print("Invalid value")
```


Prefer specific exceptions over a bare `except:`, which hides real bugs.

Accessing the Error Message

Use `as` to capture the exception object.

PYTHON

Copy

```
try:
    amount = int("abc")
except ValueError as e:
    print("Error:", e)
```

Output:

TEXT

Copy

```
Error: invalid literal for int() with base 10: 'abc'
```

`else`

The `else` block runs only when no exception occurs in the `try` block.

PYTHON

Copy

```
try:
    amount = int("500")
except ValueError:
    print("Invalid amount")
else:
    print("Payment of", amount, "accepted")
```

Output:

TEXT

Copy

```
Payment of 500 accepted
```

finally

The `finally` block runs **always** — whether an exception occurred or not. Use it for cleanup such as closing files, closing connections, or ending a transaction.

PYTHON

Copy

```
connection_open = False

try:
    connection_open = True
    print("Processing payment")
except RuntimeError:
    print("Payment failed")
finally:
    if connection_open:
        print("Closing connection")
```

`finally` runs even if `try` succeeds, even if `except` runs, and even if the function returns from inside `try`.

Full Structure

PYTHON

Copy

```
try:
    # risky code
except SomeError:
    # runs if that error occurs
else:
    # runs if no error occurs
finally:
    # runs always
```

Block	Runs When
<code>try</code>	Always (contains the risky code)
<code>except</code>	An exception occurs
<code>else</code>	No exception occurs
<code>finally</code>	Always (cleanup)

Raising & Custom

Raising Exceptions

Use `raise` to trigger an exception yourself — useful for rejecting invalid data.

PYTHON

Copy

```
def process_payment(amount):  
    if amount <= 0:  
        raise ValueError("Amount must be positive")  
    print("Processing:", amount)  
  
process_payment(-50)
```

Output:

TEXT

Copy

```
ValueError: Amount must be positive
```

`raise` stops the function immediately, exactly like a built-in exception. The caller can catch it:

PYTHON

Copy

```
def process_payment(amount):  
    try:  
        process_payment(-50)  
    except ValueError as e:  
        print("Rejected:", e)
```

Output:

TEXT

Copy

raise → TRIGGER AN EXCEPTION ON PURPOSE

Custom Exceptions

Create your own exception type by inheriting from `Exception`. This makes errors meaningful and specific to your application.

PYTHON

Copy

```
class PaymentError(Exception):  
    pass  
  
def process_payment(amount, balance):  
    if amount > balance:  
        raise PaymentError("Insufficient balance")  
    print("Payment successful")  
  
try:  
    process_payment(1000, 500)  
except PaymentError as e:  
    print("Payment failed:", e)
```

Output:

TEXT

Copy

Payment failed: Insufficient balance

A custom exception reads clearly and can be caught separately from built-in errors.

TEXT

Copy

```
class MyError(Exception): pass → APP-SPECIFIC ERROR TYPE
```

assert Statement

`assert` checks that a condition is true. If it is false, Python raises an `AssertionError`. It is used to catch **programming mistakes** during development and testing.

PYTHON

Copy

```
def apply_discount(price, discount):  
    assert discount >= 0, "Discount cannot be negative"  
    return price - discount  
  
apply_discount(100, -10)
```

Output:

TEXT

Copy

```
AssertionError: Discount cannot be negative
```

TEXT

Copy

```
assert condition, "message"  
condition True → continue  
condition False → AssertionError
```

Do not use `assert` for normal user-input validation — assertions can be disabled when Python runs optimized (`python -O`). Use `raise` for real validation, `assert` for internal sanity checks.

Complete Payment Processing Example

This combines every concept — validate with `raise`, a custom exception, and `try` / `except` / `else` / `finally` in one flow.

PYTHON

Copy

```
class PaymentError(Exception):
    pass

def process_payment(amount, balance):
    if amount <= 0:
        raise ValueError("Amount must be positive")
    if amount > balance:
        raise PaymentError("Insufficient balance")
    return balance - amount

def pay(amount, balance):
    try:
        new_balance = process_payment(amount, balance)
    except ValueError as e:
        print("Invalid payment:", e)
    except PaymentError as e:
        print("Payment failed:", e)
    else:
        print("Payment successful. New balance:", new_balance)
    finally:
        print("Transaction closed\n")

pay(500, 1000)    # success
pay(-50, 1000)   # invalid amount
pay(2000, 1000)  # insufficient balance
```

Output:

TEXT

Copy

```
Payment successful. New balance: 500
Transaction closed

Invalid payment: Amount must be positive
Transaction closed

Payment failed: Insufficient balance
Transaction closed
```

Flow for each call:

TEXT

Copy

```
process_payment()
↓
amount <= 0?    → raise ValueError → except ValueError
amount > balance? → raise PaymentError → except PaymentError
otherwise      → return balance   → else block
↓
finally always runs
```

Each error path is handled cleanly, and `finally` closes the transaction every time — success or failure.

Common Fresher Mistakes

X Mistake	Problem	✓ Correct Understanding
Using bare <code>except:</code>	Hides real bugs	Catch specific exceptions
Confusing syntax error and exception	Different stages	Syntax = before run, exception = at run
Expecting <code>else</code> to run after an error	<code>else</code> runs only on success	Use <code>except</code> for the error path
Forgetting <code>finally</code> always runs	Cleanup skipped	Put cleanup in <code>finally</code>
Using <code>assert</code> for user validation	Assertions can be disabled	Use <code>raise</code> for real validation
Catching an exception but ignoring it (<code>pass</code>)	Errors silently swallowed	Log or handle it meaningfully

Placement Quick Points

- An exception is a runtime error; unhandled, it stops normal program execution.

- A syntax error is caught before running; an exception occurs while running.
- `try` holds risky code; `except` handles the error.
- Catch specific exceptions, not a bare `except:`.
- `else` runs only when no exception occurs.
- `finally` always runs — used for cleanup.
- `raise` triggers an exception on purpose.
- Custom exceptions inherit from `Exception`.
- `assert` checks internal assumptions and raises `AssertionError` on failure.
- Use `raise` for validation; use `assert` for development-time checks.

Interview Questions

Q1. What is an exception?

An error that occurs while a program is running.

Q2. What is the difference between a syntax error and an exception?

A syntax error is detected before the program runs. An exception occurs during execution.

Q3. What is the purpose of `try` and `except`?

`try` holds code that might fail; `except` handles the error if it occurs.

Q4. When does the `else` block run?

Only when no exception occurs in the `try` block.

Q5. When does the `finally` block run?

Always — whether or not an exception occurred.

Q6. What does `raise` do?

It triggers an exception intentionally.

Q7. How do you create a custom exception?

Define a class that inherits from `Exception`.

Q8. What does `assert` do?

It checks a condition and raises `AssertionError` if the condition is false.

Q9. What is the difference between `assert` and `raise`?

`assert` is for development-time checks and can be disabled. `raise` is for real error handling and always runs.

Q10. Why avoid a bare `except:`?

It catches every error, including bugs, and hides real problems.

Practice

1. Trigger and read a `ZeroDivisionError`.
2. Handle a `ValueError` from `int("abc")` using `try/except`.
3. Catch two different exceptions with separate `except` blocks.
4. Print the error message using `except ... as e`.
5. Add an `else` block that runs on success.
6. Add a `finally` block for cleanup.
7. Raise a `ValueError` for a negative payment amount.
8. Catch the raised `ValueError` in the caller.
9. Create a `PaymentError` custom exception and raise it for insufficient balance.
10. Use `assert` to check that a discount is not negative.
11. Explain the difference between `assert` and `raise`.
12. Predict the output:

PYTHON

Copy

```
try:
    print(10 / 2)
except ZeroDivisionError:
    print("Error")
else:
    print("No error")
finally:
    print("Done")
```

1. Find the mistake:

PYTHON

Copy

```
try:
    amount = int("abc")
except:
    pass
```

End of Level 8

Object-Oriented Programming

Level: 9 | Version: v1.0 | Author: Meptrasoft

Object-Oriented Programming (OOP) organizes a program around **objects** — self-contained units that bundle data (attributes) and behavior (methods). This module builds OOP from the ground up: classes and objects, the four pillars (encapsulation, inheritance, polymorphism, abstraction), and inner classes — with real-world examples, comparison tables, interview questions, and practice throughout.

Classes and Objects

Introduction to OOP

Definition & Core Concepts

Object-Oriented Programming (OOP) is a programming paradigm that organizes a program using objects. An object represents a real-world entity and contains data (attributes) and behavior (methods). OOP helps make programs more organized, reusable, and easier to maintain.

Example:

PYTHON

Copy

```
class Dog:
    def bark(self):
        print("Woof! Woof!")
dog1 = Dog()
dog1.bark()
```

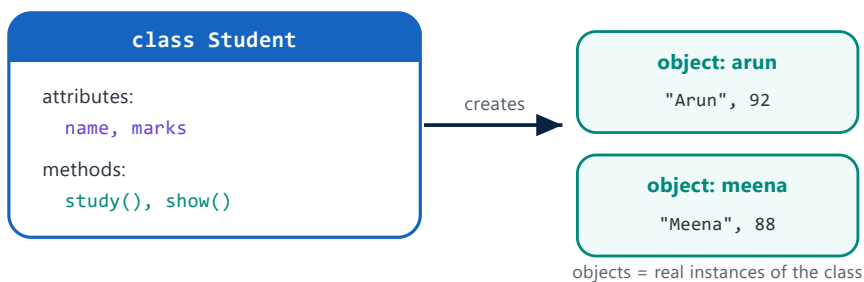
Output:

TEXT

Copy

Woof! Woof!

- `Dog` is a class.
- `dog1` is an object.
- `bark()` is a method.
- The object `dog1` calls the `bark()` method.

Class = blueprint**Inheritance: child reuses parent**

A class is a blueprint; objects are real instances built from it. Inheritance lets a child class reuse a parent's attributes and methods.

Why Do We Use Object-Oriented Programming?

OOP helps us organize related data and functions into a single unit called an object. This makes programs easier to read, manage, reuse, and expand.

Without OOP:

PYTHON

Copy

```
student1_name = "Arun"
student1_age = 21
student2_name = "Priya"
student2_age = 22
print(student1_name)
print(student2_name)
```

When the number of students increases, managing separate variables becomes difficult.

With OOP:

PYTHON

Copy

```
class Student:
    def __init__(self, name, age):
        self.name = name
        self.age = age
student1 = Student("Arun", 21)
student2 = Student("Priya", 22)
print(student1.name)
print(student2.name)
```

Each student's information is stored inside its own object, making the code clean and organized.

Procedural Programming vs Object-Oriented Programming

Procedural Programming	Object-Oriented Programming
Focuses on functions	Focuses on objects
Data and functions are separate	Data and methods are grouped together
Suitable for small programs	Suitable for small and large programs
More code duplication	Better code reuse
Harder to maintain large projects	Easier to maintain and extend

Procedural Programming Example:

PYTHON

Copy

```
def student_details(name, age):  
    print(name)  
    print(age)  
student_details("Arun", 21)
```

Object-Oriented Programming Example:

PYTHON

Copy

```
class Student:  
    def __init__(self, name, age):  
        self.name = name  
        self.age = age  
    def display(self):  
        print(self.name)  
        print(self.age)  
student = Student("Arun", 21)  
student.display()
```

Basic Structure of an OOP Program

Every OOP program follows a simple structure:

PYTHON

Copy

```
class Student:  
    def display(self):  
        print("Hello")  
student = Student()  
student.display()
```

Structure Explanation:

Part	Description
<code>class</code>	Defines a blueprint for creating objects
<code>Student</code>	Name of the class
<code>display()</code>	A method that defines an action
<code>student</code>	An object created from the class
<code>student.display()</code>	Calls the object's method

Real-World Examples of OOP

Real-World Entity	Class	Objects
Car	<code>Car</code>	BMW, Audi, Tesla
Student	<code>Student</code>	Arun, Priya
Mobile Phone	<code>Mobile</code>	iPhone, Samsung
Employee	<code>Employee</code>	John, David
Bank Account	<code>BankAccount</code>	Account1, Account2

Advantages of OOP

Advantage	Description
Code Reusability	Reuse existing classes instead of writing the same code again
Better Organization	Related data and methods stay together
Easy Maintenance	Programs are easier to update and debug
Scalability	Easy to add new features as the program grows
Real-World Modeling	Objects closely represent real-world entities

Placement Quick Points

- OOP stands for Object-Oriented Programming.
- OOP organizes a program using objects.
- Objects contain attributes (data) and methods (behavior).
- A class is a blueprint used to create objects.
- OOP improves code organization, reusability, and maintainability.
- Python supports Object-Oriented Programming along with procedural programming.
- The four main pillars of OOP are:
 - Inheritance
 - Encapsulation
 - Polymorphism
 - Abstraction

Interview Questions

Q1. What is Object-Oriented Programming (OOP)?

Object-Oriented Programming (OOP) is a programming paradigm that organizes programs using objects, where each object contains data and the functions that operate on that data.

Q2. Why is OOP used?

OOP is used to make programs more organized, reusable, maintainable, and easier to expand.

Q3. What is the difference between Procedural Programming and OOP?

The differences are:

Procedural Programming	Object-Oriented Programming
Function-based	Object-based
Data and functions are separate	Data and methods are grouped together
Best for small programs	Better for large applications

Q4. What are the four pillars of OOP?

The four pillars are:

- Inheritance
- Encapsulation
- Polymorphism
- Abstraction

Q5. Does Python support Object-Oriented Programming?

Yes. Python fully supports Object-Oriented Programming and also supports procedural and functional programming.

Q6. Give a real-world example of OOP.

A `Car` is an example of OOP.

- Attributes: Brand, Color, Speed
- Methods: Start, Stop, Accelerate

Practice

1. Write a class named `Car` with a method `start()` that prints `"Car Started"`.
2. Create an object of the `Car` class and call the `start()` method.
3. List three real-world objects and write two attributes and two methods for each.
4. Write one advantage of Object-Oriented Programming over Procedural Programming.
5. Identify the class and the object in the following code:

PYTHON

Copy

```
class Student:
    pass
student1 = Student()
```

6. Explain in your own words why OOP is useful for large applications.

Class Fundamentals**Definition & Purpose**

A class is a blueprint or template used to create objects. It defines the attributes (data) and methods (functions) that all objects created from the class will have.

A class itself does not store individual data. Instead, it provides the structure from which objects are created.

Example:

PYTHON

Copy

```
class Student:  
    pass
```

- `class` is the keyword used to define a class.
- `Student` is the name of the class.
- `pass` is a placeholder that allows us to create an empty class.

At this stage, the class only defines a blueprint. No object has been created yet.

⚠ IMPORTANT

Key Takeaway: Class vs Object Data

A `class` defines the blueprint and behavior. Individual object data, such as a student's name or marks, is stored only after an **object** is created using `class` `ClassName: object_name = ClassName()`.

Why Do We Use a Class?

A class allows us to create multiple objects with the same structure instead of writing duplicate code.

Without a Class:

PYTHON

Copy

```
student1_name = "Arun"  
student1_age = 21  
student2_name = "Priya"  
student2_age = 22  
print(student1_name)  
print(student2_name)
```

Managing many students like this becomes difficult.

With a Class:

PYTHON

Copy

```
class Student:
    pass
student1 = Student()
student2 = Student()
print(student1)
print(student2)
```

Each object is created from the same class, making the program more organized.

How to Create a Class?

Use the `class` keyword followed by the class name.

Syntax:

PYTHON

Copy

```
class ClassName:
    pass
```

Example:

PYTHON

Copy

```
class Car:
    pass
class Employee:
    pass
class Mobile:
    pass
```

A good practice is to write class names using PascalCase.

Naming Convention for Classes

Python recommends using PascalCase for class names.

X Incorrect	✓ Correct
student	Student
bank_account	BankAccount
employee_details	EmployeeDetails
mobile_phone	MobilePhone

Example:

PYTHON

Copy

```
class BankAccount:
    pass
```

Creating Multiple Objects from One Class

A single class can be used to create many objects.

PYTHON

Copy

```
class Student:
    pass
student1 = Student()
student2 = Student()
student3 = Student()
print(student1)
print(student2)
print(student3)
```

Each object is different, even though they are created from the same class.

Class vs Object

Class	Object
Blueprint or template	Instance created from the class
Defines the structure	Holds actual data
Created using <code>class</code> keyword	Created by calling the class
One class can create many objects	Each object is independent

Example:

TEXT

Copy

```
class Car:
    pass
car1 = Car()
print(car2 = Car())
print(car2)
print(car2)
```

Output:

PYTHON

Copy

```
<Car object at 0x000002B6AB273F90>
<Car object at 0x000002B6AB273F90>
```

Empty Class

Sometimes we create a class first and implement it later.

PYTHON

Copy

```
class Student:
    pass
```

The `pass` keyword tells Python to do nothing for now.

Checking the Type of an Object

Every object belongs to a class.

TEXT

Copy

```
class Student:
    pass
student1 = Student()
print(type(student1))
```

Output:

PYTHON

Copy

```
<class 'Student'>
```

This shows that `student1` is an object of the `Student` class.

Placement Quick Points

- A class is a blueprint used to create objects.
- Classes are created using the `class` keyword.
- Python follows PascalCase naming for class names.
- A class can create multiple objects.
- A class defines the structure of objects.
- `pass` is used to create an empty class.
- Objects created from the same class are independent of each other.

Interview Questions

Q1. What is a class?

A class is a blueprint or template used to create objects. It defines the attributes and methods that objects will have.

Q2. Which keyword is used to create a class?

The `class` keyword.

Q3. What is the purpose of the `pass` keyword in a class?

The `pass` keyword creates an empty class without causing a syntax error.

Q4. Can one class create multiple objects?

Yes. A single class can be used to create any number of objects.

Q5. What naming convention is recommended for Python class names?

PascalCase.

PYTHON

Copy

```
class StudentDetails:  
    pass
```

Q6. What is the difference between a class and an object?

A class is a blueprint, while an object is an instance created from that blueprint.

Practice

1. Create an empty class named `Student`.
2. Create an empty class named `Car`.
3. Create three objects from a class named `Employee`.
4. Write the correct class names using PascalCase:
5. `bank_account`
6. `student_details`
7. `mobile_phone`
8. Identify the class and objects in the following code:

TEXT

Copy

```
class Animal:  
    pass  
dog = Animal()  
cat = Animal()
```

9. Write a program to create an empty class named `Book` and create two objects from it.

Objects

What is an Object?

An object is an instance of a class. It is created from a class and represents a real-world entity. An object can store its own data (attributes) and perform actions (methods) defined by the class.

Example:

PYTHON

Copy

```
class Student:
    pass
student1 = Student()
```

- `Student` is a class.
- `student1` is an object created from the `Student` class.
- The object is an instance of the class.

Why Do We Use Objects?

Objects allow us to create multiple independent instances from the same class. Each object can represent a different real-world entity.

Example:

PYTHON

Copy

```
class Student:
    pass
student1 = Student()
student2 = Student()
student3 = Student()
print(student1)
print(student2)
print(student3)
```

Output:

TEXT

Copy

```
<Student object at 0x...>  
<Student object at 0x...>  
<Student object at 0x...>
```

Although all three objects are created from the same class, each object is different.

How to Create an Object?

An object is created by calling the class name followed by parentheses `()`.

Syntax:

PYTHON

Copy

```
object_name = ClassName()
```

Example:

PYTHON

Copy

```
class Car:  
    pass  
car1 = Car()  
print(car2 = Car())
```

Output:

TEXT

Copy

```
Car → Class  
car1 → Object  
car2 → Object
```

Multiple Objects from One Class

One class can create any number of objects.

PYTHON

Copy

```
class Employee:
    pass
employee1 = Employee()
employee2 = Employee()
employee3 = Employee()
```

Every object is created using the same blueprint.

Objects are Independent

Each object is stored separately in memory.

TEXT

Copy

```
class Student:
    pass
student1 = Student()
student2 = Student()
print(student1 == student2)
```

Output:

PYTHON

Copy

```
False
```

Even though both objects belong to the same class, they are different objects.

Checking the Type of an Object

Use the `type()` function to check the class of an object.

TEXT

Copy

```
class Student:
    pass
student1 = Student()
print(type(student1))
```

Output:

PYTHON

Copy

```
<class 'Student'>
```

This confirms that `student1` is an object of the `Student` class.

Checking the Identity of an Object

Use the `id()` function to view the unique identity value of an object during the program run.

PYTHON

Copy

```
class Student:
    pass
student1 = Student()
student2 = Student()
print(id(student1))
print(id(student2))
```

Output:

TEXT

Copy

```
2983578767632
2983578767952
```

The values may be different on every computer or every run, but each object has its own unique identity.

One Class, Many Objects

PYTHON

Copy

```
class Dog:
    pass
dog1 = Dog()
dog2 = Dog()
dog3 = Dog()
```

Class	Objects
Dog	dog1
Dog	dog2
Dog	dog3

A single class can create multiple objects.

Real-World Examples

Class	Objects
Student	Arun, Priya, Rahul
Car	BMW, Audi, Tesla
Mobile	iPhone, Samsung
Employee	John, David
Book	Python Book, Java Book

Placement Quick Points

- An object is an instance of a class.
- Objects are created by calling the class name using `()`.
- A single class can create multiple objects.
- Each object is independent and has its own identity.
- Use `type()` to check an object's class.
- Use `id()` to check an object's unique identity value.

Interview Questions

Q1. What is an object?

An object is an instance of a class. It represents a real-world entity and is created from a class.

Q2. How do you create an object in Python?

By calling the class name with parentheses.

PYTHON

Copy

```
student = Student()
```

Q3. Can one class create multiple objects?

Yes. A single class can create any number of objects.

Q4. Which function is used to check an object's class?

The `type()` function.

PYTHON

Copy

```
print(type(student))
```

Q5. Which function is used to check an object's identity?

The `id()` function.

PYTHON

Copy

```
print(id(student))
```

Q6. Are two objects created from the same class equal?

No. They are separate instances with different identities.

Practice

1. Create a class named `Car` and create two objects from it.
2. Create three objects from a class named `Employee`.
3. Use the `type()` function to check the type of an object.
4. Use the `id()` function to print the identity of two different objects.
5. Identify the class and objects in the following code:

PYTHON

Copy

```
class Animal:
    pass
dog = Animal()
cat = Animal()
```

6. What will be the output of the following code?

TEXT

Copy

```
class Student:
    pass
student1 = Student()
student2 = Student()
print(student1 == student2)
```

Explain why the output is produced.

`__init__()` and `self`

What is `__init__()`?

`__init__()` is a special method (constructor) in Python that is automatically called whenever an object is created. It is mainly used to initialize the object's attributes with initial values.

Example:

PYTHON

Copy

```
class Student:
    def __init__(self):
        print("Object Created")
student1 = Student()
```

Output:

TEXT

Copy

Object Created

- `__init__()` is called automatically when `Student()` is executed.
- We do not call `__init__()` directly.
- It runs once for every new object that is created.

Why Do We Use `__init__()`?

Without `__init__()`, we have to assign values to each object manually.

Without `__init__()`:

PYTHON

Copy

```
class Student:
    pass
student1 = Student()
student1.name = "Arun"
student1.age = 21
print(student1.name)
print(student1.age)
```

With `__init__()`:

TEXT

Copy

```
class Student:
    def __init__(self, name, age):
        self.name = name
        self.age = age
student1 = Student("Arun", 21)
print(student1.name)
print(student1.age)
```

Using `__init__()` makes object creation easier and ensures every object is initialized properly.

How to Create a Constructor?

A constructor is created using the special method `__init__()`.

Syntax:

TEXT

Copy

```
class ClassName:
    def __init__(self):
        pass
```

Example:

PYTHON

Copy

```
class Car:
    def __init__(self):
        print("Car Object Created")
car1 = Car()
```

Constructor Without Parameters

A constructor can simply perform an action when an object is created.

TEXT

Copy

```
class Student:
    def __init__(self):
        print("Welcome!")
student1 = Student()
student2 = Student()
```

Output:

PYTHON

Copy

```
Welcome!
Welcome!
```

The constructor runs once for each object.

Constructor With Parameters

The constructor can receive values while creating an object.

PYTHON

Copy

```
class Student:
    def __init__(self, name, age):
        self.name = name
        self.age = age
student1 = Student("Arun", 21)
print(student1.name)
print(student1.age)
```

Output:

PYTHON

Copy

```
Arun
21
```

What is `self`?

`self` is a reference to the current object. It allows an object to access its own attributes and methods.

TEXT

Copy

```
class Student:
    def __init__(self, name):
        self.name = name
class Student:
class Student:
    pass
student1 = Student()
student2 = Student()
student1 = Student("Arun")
print(student1.name)
```

Output:

PYTHON

Copy

```
Arun
```

When `student1` is created:

PYTHON

Copy

```
student1 -----> Student Object
                  |
                  self.name
                  |
                  "Arun"
```

`self.name` stores the value inside the current object.Why Do We Use `self`?`self` helps Python identify which object's data should be used.

Example:

TEXT

Copy

```
class Student:
    def __init__(self, name):
        self.name = name
student1 = Student("Arun")
student2 = Student("Priya")
print(student1.name)
print(student2.name)
```

Output:

PYTHON

Copy

```
Arun
Priya
```

Each object stores its own value because of `self`.

How Does `self` Work?

When we create an object:

TEXT

Copy

```
student1 = Student("Arun")
```

Python internally works like this:

PYTHON

Copy

```
class Student:
Student.__init__(student1, "Arun")
```

- `self` refers to `student1`.
- Python passes the object automatically.
- We do not pass `self` manually.

Constructor vs Normal Method

Constructor (<code>__init__()</code>)	Normal Method
Called automatically	Called manually
Runs when an object is created	Runs only when invoked
Used to initialize objects	Used to perform actions
Defined using <code>__init__()</code>	Can have any valid method name

Example:

PYTHON

Copy

```
class Student:
    def __init__(self):
        print("Constructor")
    def display(self):
        print("Normal Method")
student = Student()
student.display()
```

Output:

PYTHON

Copy

```
Constructor
Normal Method
```

`self` vs Object Name

TEXT

Copy

```
class Student:
    def __init__(self, name):
        self.name = name
student = Student("Arun")
print(student.name)
```

Inside the Class	Outside the Class
<code>self.name</code>	<code>student.name</code>
Refers to the current object	Refers to a specific object

Placement Quick Points

- `__init__()` is a special method called automatically when an object is created.
- `__init__()` is also known as the constructor.
- `self` refers to the current object.
- `self` is passed automatically by Python.
- `self` is used to access an object's attributes and methods.
- Every object has its own copy of data stored using `self`.
- `__init__()` is mainly used to initialize object attributes.

Interview Questions

Q1. What is `__init__()`?

`__init__()` is a special method (constructor) that is automatically called when an object is created.

Q2. Why is `__init__()` used?

It is used to initialize the attributes of an object when it is created.

Q3. What is `self` in Python?

`self` is a reference to the current object. It allows an object to access its own attributes and methods.

Q4. Do we need to pass `self` while creating an object?

No. Python passes `self` automatically.

Q5. What is the difference between `__init__()` and a normal method?

The differences are:

<code>__init__()</code>	Normal Method
Called automatically	Called manually
Initializes the object	Performs actions

Q6. Can a constructor have parameters?

Yes. A constructor can accept parameters to initialize object attributes.

PYTHON

Copy

```
class Student:
    def __init__(self, name):
        self.name = name
```

Practice

1. Create a class `Student` with a constructor that prints `"Student Created"`.
2. Create a class `Car` with a constructor that accepts brand and stores it using `self`.
3. Create two `Student` objects with different names and print their names.
4. Explain the purpose of `self` in your own words.
5. What will happen if you remove `self` from the `__init__()` method?
6. Identify the constructor and the object in the following code:

TEXT

Copy

```
class Employee:
    def __init__(self, name):
        self.name = name
employee = Employee("Rahul")
```

Attributes

What are Attributes?

Attributes are variables that belong to an object or a class. They are used to store the data or properties of an object.

For example, a student has attributes like name, age, and marks.

Example:

PYTHON

Copy

```
class Student:
    def __init__(self, name, age):
        self.name = name
        self.age = age
student1 = Student("Arun", 21)
print(student1.name)
print(student1.age)
```

Output:

PYTHON

Copy

```
Arun
21
```

- `name` and `age` are attributes.
- `self.name` stores the student's name.
- `self.age` stores the student's age.

Why Do We Use Attributes?

Attributes allow every object to store its own data.

Example:

TEXT

Copy

```
class Student:
    def __init__(self, name, age):
        self.name = name
        self.age = age
student1 = Student("Arun", 21)
student2 = Student("Priya", 22)
print(student1.name)
print(student2.name)
```

Output:

PYTHON

Copy

```
Arun
Priya
```

Although both objects belong to the same class, each object stores different values.

How to Create Attributes?

Attributes are usually created inside the `__init__()` method using `self`.

Syntax:

TEXT

Copy

```
self.attribute_name = value
```

Example:

PYTHON

Copy

```
class Car:
    def __init__(self, brand, color):
        self.brand = brand
        self.color = color
car1 = Car("BMW", "Black")
print(car1.brand)
print(car1.color)
```

Accessing Attributes

Use the dot (`.`) operator to access an object's attributes.

Example:

PYTHON

Copy

```
class Student:
    def __init__(self, name):
        self.name = name
student1 = Student("Arun")
print(student1.name)
```

Output:

TEXT

Copy

Arun

Modifying Attributes

An attribute's value can be changed after the object is created.

Example:

PYTHON

Copy

```
class Student:
    def __init__(self, name):
        self.name = name
student1 = Student("Arun")
student1.name = "Rahul"
print(student1.name)
```

Output:

PYTHON

Copy

Rahul

Deleting Attributes

Use the `del` keyword to remove an attribute from an object.

TEXT

Copy

```
class Student:
    def __init__(self, name):
        self.name = name
student1 = Student("Arun")
del student1.name
```

After deleting the attribute, trying to access it will result in an error.

Instance Attributes

An instance attribute belongs to an individual object. Every object has its own copy.

Example:

PYTHON

Copy

```
class Student:
    def __init__(self, name):
        self.name = name
student1 = Student("Arun")
student2 = Student("Priya")
print(student1.name)
print(student2.name)
```

Output:

PYTHON

Copy

```
Arun
Priya
```

Each object stores its own value for `name`.

Class Attributes

A class attribute belongs to the class itself and is shared by all objects.

Example:

PYTHON

Copy

```
class Student:
    school = "ABC School"
student1 = Student()
student2 = Student()
print(student1.school)
print(student2.school)
```

Output:

TEXT

Copy

```
ABC School
ABC School
```

Both objects access the same class attribute.

Instance Attributes vs Class Attributes

Instance Attribute	Class Attribute
Belongs to an object	Belongs to the class
Created using <code>self</code>	Created inside the class, outside methods
Each object has its own copy	Shared by all objects
Can have different values	Same value for all objects (unless modified)

Example:

PYTHON

Copy

```
class Student:
    school = "ABC School"
    def __init__(self, name):
        self.name = name
student1 = Student("Arun")
student2 = Student("Priya")
print(student1.name)
print(student2.name)
print(student1.school)
print(student2.school)
```

Placement Quick Points

- Attributes are variables that store data for a class or an object.
- Attributes are usually created inside the `__init__()` method using `self`.
- Use the dot (`.`) operator to access attributes.
- Attribute values can be modified after object creation.
- Use the `del` keyword to delete an attribute.
- Instance attributes belong to individual objects.
- Class attributes are shared by all objects of a class.

Interview Questions

Q1. What are attributes in Python?

Attributes are variables that belong to a class or an object and store data.

Q2. Where are instance attributes usually created?

Inside the `__init__()` method using `self`.

Q3. How do you access an attribute?

Using the dot (`.`) operator.

TEXT

Copy

```
print(student.name)
```

Q4. What is the difference between an instance attribute and a class attribute?

The differences are:

Instance Attribute	Class Attribute
Belongs to an object	Belongs to the class
Unique for each object	Shared by all objects

Q5. Can an attribute be modified after object creation?

Yes.

PYTHON

Copy

```
student.name = "Rahul"
```

Q6. Which keyword is used to delete an attribute?

The `del` keyword.

TEXT

Copy

```
del student.name
```

Practice

1. Create a `Student` class with attributes `name` and `age`.
2. Create two student objects with different values and print their attributes.
3. Create a `Car` class with attributes `brand` and `color`.
4. Modify the `color` attribute of a car object after it is created.
5. Create a class attribute named `country` with the value `"India"` and access it using two different objects.
6. Identify the instance attribute and the class attribute in the following code:

PYTHON

Copy

```
class Employee:
    company = "ABC Technologies"
    def __init__(self, name):
        self.name = name
```

Methods

What are Methods?

A method is a function defined inside a class. Methods define the behavior or actions that an object can perform.

For example, a car can start, stop, and accelerate. These actions are represented as methods.

Example:

TEXT

Copy

```
class Car:
    def start(self):
        print("Car Started")
car1 = Car()
car1.start()
```

Output:

PYTHON

Copy

```
Car Started
```

- `start()` is a method.
- It is defined inside the `Car` class.
- `car1.start()` calls the method.

Why Do We Use Methods?

Methods allow objects to perform actions instead of only storing data.

Without Methods:

TEXT

Copy

```
class Student:
    def __init__(self, name):
        self.name = name
student1 = Student("Arun")
print(student1.name)
print("Welcome", student1.name)
```

With Methods:

PYTHON

Copy

```
class Student:
    def __init__(self, name):
        self.name = name
    def greet(self):
        print("Welcome", self.name)
student1 = Student("Arun")
student1.greet()
```

Output:

PYTHON

Copy

```
Welcome Arun
```

The greeting logic is now part of the class, making the code cleaner and reusable.

How to Create a Method?

Methods are created inside a class using the `def` keyword.

Syntax:

TEXT

Copy

```
class ClassName:
    def method_name(self):
        pass
```


Example:

PYTHON

Copy

```
class Dog:
    def bark(self):
        print("Woof! Woof!")
dog1 = Dog()
dog1.bark()
```

Calling a Method

A method is called using the dot (.) operator.

Example:

TEXT

Copy

```
class Employee:
    def work(self):
        print("Employee is working")
employee1 = Employee()
employee1.work()
```

Output:

PYTHON

Copy

```
Employee is working
```

Methods with Parameters

A method can receive additional values through parameters.

Example:

TEXT

Copy

```
class Calculator:
    def add(self, a, b):
        print(a + b)
calc = Calculator()
calc.add(10, 20)
```

Output:

PYTHON

Copy

30

- `self` refers to the current object.
- `a` and `b` are parameters passed to the method.

Methods Returning Values

A method can return a value using the `return` statement.

Example:

PYTHON

Copy

```
class Calculator:
    def add(self, a, b):
        return a + b
calc = Calculator()
result = calc.add(10, 20)
print(result)
```

Output:

PYTHON

Copy

30

Accessing Attributes Inside Methods

Methods can access an object's attributes using `self`.

Example:

PYTHON

Copy

```
class Student:
    def __init__(self, name):
        self.name = name
    def display(self):
        print("Student Name:", self.name)
student1 = Student("Arun")
student1.display()
```

Output:

PYTHON

Copy

```
Student Name: Arun
```

Calling One Method from Another Method

A method can call another method of the same object using `self`.

Example:

PYTHON

Copy

```
class Student:
    def greet(self):
        print("Welcome!")
    def display(self):
        self.greet()
        print("Learning Python")
student1 = Student()
student1.display()
```

Output:

TEXT

Copy

```
Welcome!
Learning Python
```

Methods vs Functions

Method	Function
Defined inside a class	Defined outside a class
Called using an object	Called directly
Has <code>self</code> as the first parameter	Does not require <code>self</code>
Represents an object's behavior	Performs a general task

Example:

PYTHON

Copy

```
def greet():
    print("Hello")
class Student:
    def greet(self):
        print("Hello Student")
greet()
student = Student()
student.greet()
```

Readable Objects with `__str__`

By default, printing an object shows something like `<__main__.Student object at 0x...>`, which is not helpful. Defining a `__str__()` method lets you control what `print()` displays.

Example:

PYTHON

Copy

```
class Student:
    def __init__(self, name, marks):
        self.name = name
        self.marks = marks
    def __str__(self):
        return f"Student: {self.name}, Marks: {self.marks}"

student = Student("Arun", 90)
print(student)
```

Output:

PYTHON

Copy

```
Student: Arun, Marks: 90
```

- `__str__()` must return a string.
- `print(student)` now shows a meaningful message instead of the memory address.

Python also provides `__repr__()`, which gives the developer-facing form of an object (used in the interactive shell and when printing a list of objects). If `__str__()` is not defined, `print()` falls back to `__repr__()`.

PYTHON

Copy

```
class Student:
    def __init__(self, name):
        self.name = name
    def __repr__(self):
        return f"Student('{self.name}')"

student = Student("Arun")
print(student)
```

Output:

TEXT

Copy

```
Student('Arun')
```

Placement Quick Points

- A method is a function defined inside a class.
- Methods define the behavior of an object.
- Methods are created using the `def` keyword.
- The first parameter of an instance method is `self`.
- Methods are called using the dot (`.`) operator.
- Methods can accept parameters.
- Methods can return values using `return`.
- Methods can access object attributes using `self`.

Interview Questions

Q1. What is a method?

A method is a function defined inside a class that represents the behavior of an object.

Q2. How do you call a method?

Using the dot (.) operator.

TEXT

Copy

```
student.display()
```

Q3. What is the first parameter of an instance method?

self

Q4. Can methods accept parameters?

Yes.

PYTHON

Copy

```
class Calculator:  
    def add(self, a, b):  
        return a + b
```

Q5. Can methods return values?

Yes. A method can return values using the `return` statement.

Q6. What is the difference between a method and a function?

The differences are:

Method	Function
Inside a class	Outside a class
Called using an object	Called directly
Uses <code>self</code>	Does not use <code>self</code>

Practice

- Create a `Car` class with a method `start()` that prints `"Car Started"`.
- Create a `Student` class with a method `display()` that prints the student's name.
- Create a `Calculator` class with a method `multiply(a, b)` that returns the product.
- Write a class `Dog` with two methods: `bark()` and `sleep()`.
- Create a class `Employee` with a method `work()`. Create two objects and call the method for both.
- Identify the methods in the following code:

TEXT

Copy

```
class Mobile:
    def call(self):
        print("Calling...")
    def message(self):
        print("Sending Message")
```

Inheritance

Inheritance Basics

What is Inheritance?

Inheritance is an Object-Oriented Programming (OOP) feature that allows one class (child class) to inherit the attributes and methods of another class (parent class). It helps in reusing existing code and creating a relationship between classes.

Example:

TEXT

Copy

```
class Animal:
    def eat(self):
        print("Animal is eating")
class Dog(Animal):
    pass
dog = Dog()
dog.eat()
```

Output:

PYTHON

Copy

```
Animal is eating
```

- `Animal` is the parent class.
- `Dog` is the child class.
- `Dog` inherits the `eat()` method from `Animal`.
- Therefore, `dog.eat()` works even though `eat()` is not defined inside the `Dog` class.

Why Do We Use Inheritance?

Inheritance allows us to reuse code instead of writing the same methods again in multiple classes.

Without Inheritance:

TEXT

Copy

```
class Dog:
    def eat(self):
        print("Animal is eating")
class Cat:
    def eat(self):
        print("Animal is eating")
```

The `eat()` method is duplicated in both classes.

With Inheritance:

TEXT

Copy

```
class Animal:
    def eat(self):
        print("Animal is eating")
class Dog(Animal):
    pass
class Cat(Animal):
    pass
dog = Dog()
cat = Cat()
dog.eat()
cat.eat()
```

Now both classes reuse the same `eat()` method from the parent class.

How to Create Inheritance?

A child class is created by passing the parent class name inside parentheses.

Syntax:

PYTHON

Copy

```
class Parent:
    pass
class Child(Parent):
    pass
```

Example:

TEXT

Copy

```
class Vehicle:
    def start(self):
        print("Vehicle Started")
class Car(Vehicle):
    pass
car = Car()
car.start()
```

Parent Class

A parent class (also called a base class or superclass) is the class whose attributes and methods are inherited by another class.

Example:

TEXT

Copy

```
class Animal:
    def sleep(self):
        print("Sleeping...")
```

`Animal` is the parent class.

Child Class

A child class (also called a derived class or subclass) inherits the attributes and methods of the parent class.

Example:

PYTHON

Copy

```
class Animal:
    def sleep(self):
        print("Sleeping...")
class Dog(Animal):
    pass
dog = Dog()
dog.sleep()
```

Output:

TEXT

Copy

Sleeping...

`Dog` inherits the `sleep()` method from `Animal`.

Placement Quick Points

- Inheritance allows one class to inherit another class's attributes and methods.
- The class being inherited is called the parent class.
- The class that inherits is called the child class.
- Inheritance helps in code reusability.
- A child class can directly use the methods of its parent class.
- Inheritance creates an "is-a" relationship between classes.

Interview Questions

Q1. What is inheritance?

Inheritance is an OOP feature where one class acquires the attributes and methods of another class.

Q2. What is a parent class?

A parent class is the class whose members are inherited by another class.

Q3. What is a child class?

A child class is the class that inherits the members of a parent class.

Q4. Why is inheritance used?

Inheritance is used to reuse existing code, reduce duplication, and establish relationships between classes.

Q5. How do you create inheritance in Python?

By passing the parent class name inside parentheses.

PYTHON

Copy

```
class Child(Parent):  
    pass
```

Practice

1. Create a parent class `Animal` with a method `eat()`. Create a child class `Dog` and call the inherited method.
2. Create a parent class `Vehicle` with a method `start()`. Create a child class `Car` and call the method.
3. Create a parent class `Person` with a method `walk()`. Create a child class `Employee` and call the inherited method.
4. Create a parent class `Shape` with a method `draw()`. Create a child class `Circle` and call the inherited method.
5. Identify the parent class and the child class in the following code:

TEXT

Copy

```
class Animal:  
    def eat(self):  
        print("Eating")  
class Dog(Animal):  
    pass
```

6. Explain how inheritance helps reduce code duplication using your own example.

Parent Class and Child Class

For the definitions of a parent class and a child class, see **Inheritance Basics** above. This section focuses on how parent and child classes work together in practice.

How to Create a Parent and Child Class?

Create a normal class as the parent and pass its name inside parentheses while creating the child class.

Syntax:

PYTHON

Copy

```
class Parent:
    pass
class Child(Parent):
    pass
```

Example:

TEXT

Copy

```
class Vehicle:
    def start(self):
        print("Vehicle Started")
class Car(Vehicle):
    pass
car = Car()
car.start()
```

Parent Class Can Have Multiple Child Classes

One parent class can be inherited by multiple child classes.

Example:

PYTHON

Copy

```
class Animal:
    def eat(self):
        print("Animal is eating")
class Dog(Animal):
    pass
class Cat(Animal):
    pass
class Cow(Animal):
    pass
Dog().eat()
Cat().eat()
Cow().eat()
```

All three child classes inherit the same method from the parent class.

Child Class Can Have Its Own Methods

A child class can use the parent's methods and also define its own methods.

Example:

TEXT

Copy

```
class Animal:
    def eat(self):
        print("Animal is eating")
class Dog(Animal):
    def bark(self):
        print("Dog is barking")
dog = Dog()
dog.eat()
dog.bark()
```

Output:

PYTHON

Copy

```
Animal is eating
Dog is barking
```

- `eat()` comes from the parent class.
- `bark()` belongs to the child class.

Parent Class vs Child Class

Parent Class	Child Class
Provides common attributes and methods	Inherits attributes and methods from the parent
Also called Base Class or Superclass	Also called Derived Class or Subclass
Cannot inherit automatically from the child	Can use the parent's public members
Can have multiple child classes	Can add its own methods and attributes

Real-World Examples

Parent Class	Child Class
Animal	Dog, Cat, Cow
Vehicle	Car, Bike, Bus
Person	Student, Employee
Shape	Circle, Rectangle
Appliance	WashingMachine, Refrigerator

Placement Quick Points

- A parent class provides common attributes and methods.
- A child class inherits from a parent class.
- A child class automatically gets access to the parent's public methods and attributes.
- One parent class can have multiple child classes.
- A child class can define its own methods in addition to inherited methods.
- Parent-child relationships help reduce code duplication.

Interview Questions

Q1. What is a parent class?

A parent class is a class whose attributes and methods are inherited by another class.

Q2. What is a child class?

A child class is a class that inherits the attributes and methods of a parent class.

Q3. Can a parent class have multiple child classes?

Yes. A single parent class can have multiple child classes.

Q4. Can a child class have its own methods?

Yes. A child class can define new methods in addition to the inherited methods.

Q5. What are the other names for a parent class and a child class?

The other names are:

Parent Class	Child Class
Base Class	Derived Class
Superclass	Subclass

Q6. Does a child class automatically inherit the parent's public methods?

Yes. A child class automatically inherits and can use the public methods and attributes of its parent class.

Practice

- Create a parent class `Vehicle` with a method `start()`. Create a child class `Car` and call the inherited method.
- Create a parent class `Animal` with a method `eat()`. Create child classes `Dog` and `Cat`, and call the inherited method from both objects.

3. Create a parent class `Person` and a child class `Student`. Add a new method `study()` to the child class.
4. Create a parent class `Shape` with a method `draw()`. Create child classes `Circle` and `Rectangle`.
5. Identify the parent class and the child class in the following code:

TEXT

Copy

```
class Vehicle:
    def start(self):
        print("Vehicle Started")
class Bike(Vehicle):
    pass
```

6. Write one advantage of using parent and child classes instead of writing the same code in multiple classes.

Types of Inheritance

What are the Types of Inheritance?

Python supports different types of inheritance based on how classes are related to each other. These inheritance types help organize code and promote code reuse.

There are four commonly used types of inheritance in Python:

- Single Inheritance
- Multiple Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance

Why Do We Have Different Types of Inheritance?

Different applications require different relationships between classes. Python provides multiple inheritance types to model these relationships effectively.

Example:

A school management system may have:

- `Student` inherits from `Person` (Single Inheritance)
- `TeachingAssistant` inherits from both `Student` and `Employee` (Multiple Inheritance)
- `Manager` inherits from `Employee`, which inherits from `Person` (Multilevel Inheritance)

Different situations require different inheritance structures.

Single Inheritance

In Single Inheritance, one child class inherits from one parent class.

Diagram:

PYTHON

Copy

```
Animal
|
▼
Dog
```

Example:

TEXT

Copy

```
class Animal:
    def eat(self):
        print("Animal is eating")
class Dog(Animal):
    pass
dog = Dog()
dog.eat()
```

Output:

PYTHON

Copy

```
Animal is eating
```

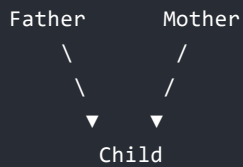
Multiple Inheritance

In Multiple Inheritance, one child class inherits from more than one parent class.

Diagram:

PYTHON

Copy



Example:

TEXT

Copy

```
class Father:
    def drive(self):
        print("Can Drive")
class Mother:
    def cook(self):
        print("Can Cook")
class Child(Father, Mother):
    pass
child = Child()
child.drive()
child.cook()
```

Output:

PYTHON

Copy

```
Can Drive
Can Cook
```

Multilevel Inheritance

In Multilevel Inheritance, a child class inherits from another child class.

Diagram:

Example:

PYTHON

Copy

```
class Person:
    def walk(self):
        print("Walking")
class Employee(Person):
    pass
class Manager(Employee):
    pass
manager = Manager()
manager.walk()
```

Output:

TEXT

Copy

Walking

Hierarchical Inheritance

In Hierarchical Inheritance, multiple child classes inherit from the same parent class.

Diagram:

PYTHON

Copy

```
      Animal
     /  |  \
    ▼  ▼  ▼
  Dog Cat  Cow
```

Example:

PYTHON

Copy

```
class Animal:
    def eat(self):
        print("Eating")
class Dog(Animal):
    pass
class Cat(Animal):
    pass
class Cow(Animal):
    pass
Dog().eat()
Cat().eat()
Cow().eat()
```

Output:

TEXT

Copy

```
Eating
Eating
Eating
```

Comparison of Inheritance Types

Inheritance Type	Description
Single	One parent → One child
Multiple	Multiple parents → One child
Multilevel	Grandparent → Parent → Child
Hierarchical	One parent → Multiple child classes

Placement Quick Points

- Python commonly uses four types of inheritance.
- Single Inheritance → One parent, one child.
- Multiple Inheritance → Multiple parents, one child.
- Multilevel Inheritance → Inheritance across multiple levels.

- Hierarchical Inheritance → One parent, multiple child classes.
- Inheritance helps improve code reusability and program organization.

Interview Questions

Q1. How many commonly used types of inheritance are there in Python?

There are four commonly used types of inheritance:

- Single Inheritance
- Multiple Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance

Q2. What is Single Inheritance?

One child class inherits from one parent class.

Q3. What is Multiple Inheritance?

One child class inherits from more than one parent class.

Q4. What is Multilevel Inheritance?

A child class inherits from another child class, forming multiple inheritance levels.

Q5. What is Hierarchical Inheritance?

Multiple child classes inherit from the same parent class.

Practice

1. Create a Single Inheritance example using `Animal` and `Dog`.
2. Create a Multiple Inheritance example using `Father`, `Mother`, and `Child`.
3. Create a Multilevel Inheritance example using `Person`, `Employee`, and `Manager`.
4. Create a Hierarchical Inheritance example using `Vehicle`, `Car`, and `Bike`.
5. Identify the type of inheritance used in the following code:

PYTHON

Copy

```
class Person:
    pass
class Student(Person):
    pass
class Employee(Person):
    pass
```

Explain your answer.

Accessing Parent Members

What is Accessing Parent Members?

Accessing Parent Members means using the attributes and methods of a parent class inside a child class. Since the child class inherits from the parent class, it can directly access the parent's public members without rewriting them.

Example:

TEXT

Copy

```
class Animal:
    def eat(self):
        print("Animal is eating")
class Dog(Animal):
    pass
dog = Dog()
dog.eat()
```

Output:

PYTHON

Copy

```
Animal is eating
```

- `Animal` is the parent class.
- `Dog` is the child class.
- The `eat()` method belongs to the parent class.

- The child object `dog` can directly call `eat()`.

Why Do We Access Parent Members?

Accessing parent members allows us to reuse existing code instead of writing the same methods again in every child class. This is the same "without vs with inheritance" comparison shown in **Inheritance Basics** above: rather than each child class defining its own `eat()` method, the child classes share a single `eat()` method inherited from the parent.

Accessing Parent Methods

A child class can directly call any public method of its parent class.

Example:

TEXT

Copy

```
class Vehicle:
    def start(self):
        print("Vehicle Started")
class Car(Vehicle):
    pass
car = Car()
car.start()
```

Output:

PYTHON

Copy

```
Vehicle Started
```

Accessing Parent Attributes

A child class also inherits the parent's attributes.

Example:

PYTHON

Copy

```
class Person:
    def __init__(self):
        self.country = "India"
class Student(Person):
    pass
student = Student()
print(student.country)
```

Output:

TEXT

Copy

India

The `country` attribute is created in the parent class but accessed through the child object.

Accessing Parent Attributes and Methods Together

A child object can use both the parent's attributes and methods.

Example:

PYTHON

Copy

```
class Employee:
    def __init__(self):
        self.company = "ABC Technologies"
    def work(self):
        print("Employee is working")
class Manager(Employee):
    pass
manager = Manager()
print(manager.company)
manager.work()
```

Output:

PYTHON

Copy

ABC Technologies
Employee is working

Child Class Can Add Its Own Members

A child class can use the parent's members and also define its own methods.

Example:

PYTHON

Copy

```
class Animal:
    def eat(self):
        print("Animal is eating")
class Dog(Animal):
    def bark(self):
        print("Dog is barking")
dog = Dog()
dog.eat()
dog.bark()
```

Output:

PYTHON

Copy

```
Animal is eating
Dog is barking
```

Parent Members vs Child Members

Parent Members	Child Members
Defined in the parent class	Defined in the child class
Inherited by the child class	Belong only to the child class
Can be accessed using the child object	Can also be accessed using the child object

Example:

PYTHON

Copy

```
class Animal:
    def eat(self):
        print("Eating")
class Dog(Animal):
    def bark(self):
        print("Barking")
dog = Dog()
dog.eat()      # Parent method
dog.bark()     # Child method
```

Placement Quick Points

- A child class automatically inherits the public attributes and public methods of its parent class.
- Parent members can be accessed directly using the child object.
- Inheritance eliminates code duplication.
- A child class can use inherited members and define its own members.
- Parent members are available to every object of the child class.

Interview Questions

Q1. What does accessing parent members mean?

It means using the attributes and methods of a parent class through a child class.

Q2. Can a child class access the parent's methods?

Yes. A child class automatically inherits and can access the parent's public methods.

Q3. Can a child class access the parent's attributes?

Yes. Public attributes of the parent class are inherited by the child class.

Q4. Why do we access parent members?

To reuse existing code and avoid writing the same code multiple times.

Q5. Can a child class have its own methods?

Yes. A child class can inherit parent members and also define its own methods.

Practice

1. Create a parent class `Vehicle` with a method `start()`. Create a child class `Car` and call the inherited method.
2. Create a parent class `Person` with an attribute `country = "India"`. Create a child class `Student` and print the inherited attribute.
3. Create a parent class `Animal` with a method `eat()`. Create a child class `Dog` with an additional method `bark()`. Call both methods.
4. Create a parent class `Employee` with an attribute `company` and a method `work()`. Access both using a child class `Manager`.
5. Identify the parent member and the child member in the following code:

TEXT

Copy

```
class Animal:
    def eat(self):
        print("Eating")
class Dog(Animal):
    def bark(self):
        print("Barking")
```

Explain why accessing parent members is useful in object-oriented programming.

Encapsulation

Encapsulation Basics

What is Encapsulation?

Encapsulation is an Object-Oriented Programming (OOP) concept that bundles data (attributes) and the methods that operate on that data into a single unit (class). It also helps protect an object's data from being accessed or modified directly.

Example:

PYTHON

Copy

```
class Student:
    def __init__(self, name):
        self.name = name
    def display(self):
        print(self.name)
student = Student("Arun")
student.display()
```

Output:

TEXT

Copy

Arun

- `name` is the data (attribute).
- `display()` is the method.
- Both are grouped inside the `Student` class.
- This is the basic idea of encapsulation.

Why Do We Use Encapsulation?

Encapsulation keeps related data and methods together and helps prevent accidental modification of an object's data.

Without Encapsulation:

PYTHON

Copy

```
name = "Arun"
print(name)
name = 123
print(name)
```

The variable can be modified from anywhere.

With Encapsulation:

PYTHON

Copy

```
class Student:
    def __init__(self, name):
        self.name = name
    def display(self):
        print(self.name)
student = Student("Arun")
student.display()
```

Now the data belongs to the object and is managed through the class.

How Does Encapsulation Work?

Encapsulation combines attributes and methods inside a class.

Example:

TEXT

Copy

```
class BankAccount:
    def __init__(self, balance):
        self.balance = balance
    def show_balance(self):
        print("Balance:", self.balance)
account = BankAccount(5000)
account.show_balance()
```

Output:

PYTHON

Copy

```
Balance: 5000
```

Both the data (`balance`) and the method (`show_balance()`) belong to the same object.

Types of Members

Python commonly uses four member categories based on how they are shared or accessed.

Example:

TEXT

Copy

```
class Student:
    school = "ABC School"      # Class Member
    def __init__(self):
        self.name = "Arun"    # Public Member
        self._age = 21        # Protected Member
        self.__marks = 95     # Private Member
student = Student()
print(student.school)
print(student.name)
print(student._age)
# print(student.__marks)    # Gives an AttributeError
```

Member Type	Syntax	Access
Class Member	<code>school</code>	Shared by all objects
Public Member	<code>name</code>	Accessible from anywhere
Protected Member	<code>_age</code>	Intended for internal use (convention)
Private Member	<code>__marks</code>	Name-mangled; should not be accessed directly outside the class

Getter and Setter methods — used to read and update private attributes — are covered in full under **Getter and Setter Methods** later in this module.

Placement Quick Points

- Encapsulation combines data and methods into a single class.
- It helps organize and protect object data.
- Python has four commonly used member types:
 - Class Members
 - Public Members
 - Protected Members

- Private Members
- Getter methods are used to read private data.
- Setter methods are used to update private data.

Interview Questions

Q1. What is encapsulation?

Encapsulation is the process of combining data and methods into a single class while helping protect the object's data.

Q2. Why is encapsulation used?

It improves code organization and helps prevent unauthorized or accidental modification of data.

Q3. Name the four types of members in Python.

The four member types are:

- Class Members
- Public Members
- Protected Members
- Private Members

Q4. What is the purpose of Getter and Setter methods?

Getter methods are used to read private attributes, while Setter methods are used to update private attributes.

Q5. Which member should not be accessed directly outside the class using its original name?

A Private Member (`__member`).

Practice

1. Create a class `Student` with a class member `school` and a public member `name`.
2. Add a protected member `_age` and a private member `__marks` to the class.

3. Write a getter method to display the value of `__marks`.
4. Write a setter method to update `__marks`.
5. Create an object and use both the getter and setter methods.
6. Identify the class, public, protected, and private members in the following code:

PYTHON

Copy

```
class Employee:
    company = "ABC Technologies"
    def __init__(self):
        self.name = "Arun"
        self._salary = 50000
        self.__bonus = 10000
```

Public, Protected and Private Members

What are Public, Protected and Private Members?

In Python, members (attributes and methods) can have different levels of accessibility. Based on how they are accessed, they are classified into:

- Public Members
- Protected Members
- Private Members

These access levels help control how the data inside a class is used.

Why Do We Use Different Types of Members?

Different members provide different levels of access to an object's data.

- Public Members can be accessed from anywhere.
- Protected Members are intended to be used within the class and its child classes.
- Private Members use name mangling to discourage direct access outside the class.

This helps make programs more organized and reduces accidental misuse of data.

Public Members

A public member is accessible from anywhere inside or outside the class. By default, all members in Python are public.

Example:

TEXT

Copy

```
class Student:
    def __init__(self):
        self.name = "Arun"
student = Student()
print(student.name)
```

Output:

PYTHON

Copy

Arun

Protected Members

A protected member starts with a single underscore (`_`).

It is intended to be used within the class and its child classes. Although it can still be accessed outside the class, it is considered an internal member by convention.

Example:

PYTHON

Copy

```
class Student:
    def __init__(self):
        self._age = 21
student = Student()
print(student._age)
```

Output:

PYTHON

Copy

21

Note: A protected member can still be accessed outside the class. The single underscore is a naming convention, not an access restriction.

Private Members

A private member starts with a double underscore (`__`).

It is name-mangled by Python, so it cannot be accessed directly using its original name outside the class. It is used to discourage direct access to important data.

Example:

TEXT

Copy

```
class Student:
    def __init__(self):
        self.__marks = 95
student = Student()
print(student.__marks)
```

Output:

TEXT

Copy

```
AttributeError: 'Student' object has no attribute '__marks'
```

The private member should normally be accessed inside the class or through methods like getters and setters.

Comparison of Member Types

Member Type	Syntax	Accessible Outside Class?
Public	<code>name</code>	✓ Yes
Protected	<code>_name</code>	✓ Yes (by convention, avoid direct access)
Private	<code>__name</code>	✗ Not directly by its original name

Common Example

PYTHON

Copy

```
class Student:
    def __init__(self):
        self.name = "Arun"           # Public
        self._age = 21                # Protected
        self.__marks = 95             # Private
student = Student()
print(student.name)
print(student._age)
# print(student.__marks) # AttributeError
```

Placement Quick Points

- Python supports Public, Protected, and Private members.
- Public members are accessible from anywhere.
- Protected members use a single underscore (`_`).
- Private members use a double underscore (`__`).
- Private members cannot be accessed directly outside the class using their original name.
- Protected members are accessible outside the class but should be treated as internal by convention.

Interview Questions

Q1. What are the three types of members in Python?

The three types are:

- Public Members
- Protected Members
- Private Members

Q2. Which member is accessible from anywhere?

Public Members.

Q3. Which symbol is used for a protected member?

A single underscore (`_`).

Q4. Which symbol is used for a private member?

A double underscore (`__`).

Q5. Can a private member be accessed directly outside the class using its original name?

No. Python changes its name internally, so it should normally be accessed through a method.

Q6. What is the difference between protected and private members?

The difference is:

Protected	Private
Uses <code>_</code>	Uses <code>__</code>
Accessible outside the class (by convention)	Cannot be accessed directly outside the class
Intended for internal use	Used to hide important data

Practice

1. Create a class `Student` with a public member `name`.
2. Add a protected member `_age` to the class.
3. Add a private member `__marks` to the class.
4. Create an object and access the public and protected members.
5. Try to access the private member directly and observe the output.
6. Identify the public, protected, and private members in the following code:

TEXT

Copy

```
class Employee:
    def __init__(self):
        self.name = "Arun"
        self._salary = 50000
        self.__bonus = 10000
```

Private Methods

What are Private Methods?

A private method is a method whose name begins with double underscores (`__`). It is intended to be used only within the class and should not be called directly from outside the class using its original name.

Private methods are used to hide the internal implementation of a class.

Example:

PYTHON

Copy

```
class Student:
    def __welcome(self):
        print("Welcome to Python!")
    def greet(self):
        self.__welcome()
student = Student()
student.greet()
```

Output:

PYTHON

Copy

```
Welcome to Python!
```

- `__welcome()` is a private method.
- `greet()` is a public method.
- The private method is called from inside the class using `self`.

Why Do We Use Private Methods?

Private methods help hide the internal logic of a class and prevent them from being called accidentally from outside the class.

Without a Private Method:

PYTHON

Copy

```
class Student:
    def welcome(self):
        print("Welcome!")
student = Student()
student.welcome()
```

Here, anyone can call the `welcome()` method directly.

With a Private Method:

TEXT

Copy

```
class Student:
    def __welcome(self):
        print("Welcome!")
    def greet(self):
        self.__welcome()
student = Student()
student.greet()
```

Now, `__welcome()` is meant to be used inside the class.

How to Create a Private Method?

A private method is created by adding two underscores (`__`) before the method name.

Syntax:

PYTHON

Copy

```
class ClassName:
    def __method_name(self):
        pass
```

Example:

TEXT

Copy

```
class BankAccount:
    def __calculate_interest(self):
        print("Interest Calculated")
    def show_interest(self):
        self.__calculate_interest()
account = BankAccount()
account.show_interest()
```

Accessing a Private Method

A private method should be accessed through a public method inside the class.

Example:

PYTHON

Copy

```
class Student:
    def __display(self):
        print("Private Method")
    def show(self):
        self.__display()
student = Student()
student.show()
```

Output:

TEXT

Copy

Private Method

Calling a Private Method Directly

Trying to call a private method directly from outside the class using its original name results in an error.

Example:

PYTHON

Copy

```
class Student:
    def __display(self):
        print("Private Method")
student = Student()
student.__display()
```

Output:

TEXT

Copy

```
AttributeError: 'Student' object has no attribute '__display'
```

Public Method vs Private Method

Public Method	Private Method
<code>display()</code>	<code>__display()</code>
Can be called from anywhere	Should normally be called from inside the class
No underscore prefix	Double underscore (<code>__</code>) prefix

Example:

PYTHON

Copy

```
class Student:
    def display(self):
        print("Public Method")
    def __private(self):
        print("Private Method")
    def show(self):
        self.display()
        self.__private()
student = Student()
student.show()
```

Placement Quick Points

- A private method starts with double underscores (`__`).

- Private methods are intended for internal use within the class.
- They cannot be called directly from outside the class using their original names.
- Private methods are usually accessed through public methods.
- Private methods help hide the internal implementation of a class.

Interview Questions

Q1. What is a private method?

A private method is a method whose name begins with double underscores (`__`) and is intended to be used only within the class.

Q2. How do you create a private method in Python?

By adding double underscores (`__`) before the method name.

Q3. Can a private method be called directly outside the class?

No. Attempting to do so results in an `AttributeError`.

Q4. How can a private method be accessed?

It can be accessed through another public method inside the same class.

Q5. Why are private methods used?

Private methods hide the internal implementation of a class and prevent accidental access from outside the class.

Practice

1. Create a class `Student` with a private method `__welcome()`.
2. Create a public method `greet()` that calls `__welcome()`.
3. Create an object and call the public method.
4. Try calling the private method directly and observe the output.
5. Identify the public and private methods in the following code:

TEXT

Copy

```
class Employee:
    def work(self):
        print("Working")
    def __calculate_salary(self):
        print("Salary Calculated")
    def show_salary(self):
        self.__calculate_salary()
```

Getter and Setter Methods

What are Getter and Setter Methods?

Getter and Setter methods are special methods used to access and modify private attributes of a class.

A Getter method is used to read the value of a private attribute.

A Setter method is used to update the value of a private attribute.

These methods provide controlled access to private data.

Example:

PYTHON

Copy

```
class Student:
    def __init__(self):
        self.__marks = 90
    def get_marks(self):
        return self.__marks
    def set_marks(self, marks):
        self.__marks = marks
student = Student()
print(student.get_marks())
student.set_marks(95)
print(student.get_marks())
```

Output:

TEXT

Copy

90
95

- `__marks` is a private attribute.
- `get_marks()` reads the value of `__marks`.
- `set_marks()` updates the value of `__marks`.

Why Do We Use Getter and Setter Methods?

Since private attributes cannot be accessed directly, Getter and Setter methods provide a safe way to read and modify them.

Without Getter and Setter:

PYTHON

Copy

```
class Student:
    def __init__(self):
        self.__marks = 90
student = Student()
print(student.__marks)
```

Output:

TEXT

Copy

```
AttributeError: 'Student' object has no attribute '__marks'
```

With Getter and Setter:

PYTHON

Copy

```
class Student:
    def __init__(self):
        self.__marks = 90
    def get_marks(self):
        return self.__marks
    def set_marks(self, marks):
        self.__marks = marks
student = Student()
print(student.get_marks())
student.set_marks(100)
print(student.get_marks())
```

Output:

TEXT

Copy

```
90
100
```

How to Create Getter and Setter Methods?

Create two methods:

- A Getter method to return the private attribute.
- A Setter method to update the private attribute.

Syntax:

PYTHON

Copy

```
class ClassName:
    def get_attribute(self):
        return self.__attribute
    def set_attribute(self, value):
        self.__attribute = value
```

Example:

PYTHON

Copy

```
class Employee:
    def __init__(self):
        self.__salary = 50000
    def get_salary(self):
        return self.__salary
    def set_salary(self, salary):
        self.__salary = salary
employee = Employee()
print(employee.get_salary())
employee.set_salary(60000)
print(employee.get_salary())
```

Getter vs Setter

Getter Method	Setter Method
Reads a private attribute	Updates a private attribute
Uses <code>return</code>	Assigns a new value
Does not modify data	Modifies data

Example:

PYTHON

Copy

```
class Student:
    def __init__(self):
        self.__marks = 90
    def get_marks(self):
        return self.__marks
    def set_marks(self, marks):
        self.__marks = marks

s = Student()
print("Initial Marks:", s.get_marks())
s.set_marks(95)
print("Updated Marks:", s.get_marks())
```

Output:

TEXT

Copy

Initial Marks: 90

Updated Marks: 95

Placement Quick Points

- Getter and Setter methods are used with private attributes.
- A Getter method reads the value of a private attribute.
- A Setter method updates the value of a private attribute.
- They provide controlled access to private data.
- Getter methods usually use the `return` statement.
- Setter methods assign a new value to the private attribute.

Interview Questions

Q1. What is a Getter method?

A Getter method is used to read the value of a private attribute.

Q2. What is a Setter method?

A Setter method is used to update the value of a private attribute.

Q3. Why are Getter and Setter methods used?

They provide controlled access to private attributes without exposing them directly.

Q4. Which type of attribute is commonly used with Getter and Setter methods?

Private attributes (`__attribute`).

Q5. What is the difference between a Getter and a Setter?

The difference is:

Getter	Setter
Reads data	Updates data
Returns a value	Assigns a value

Practice

1. Create a class `Student` with a private attribute `__marks`.
2. Write a Getter method to return the value of `__marks`.
3. Write a Setter method to update the value of `__marks`.
4. Create an object and use both the Getter and Setter methods.
5. Identify the Getter and Setter methods in the following code:

PYTHON

Copy

```
class Employee:
    def __init__(self):
        self.__salary = 50000
    def get_salary(self):
        return self.__salary
    def set_salary(self, salary):
        self.__salary = salary
```

Polymorphism

Polymorphism Basics

What is Polymorphism?

Polymorphism is an Object-Oriented Programming (OOP) concept that allows the same method or operation to perform different actions depending on the object it is used with.

The word Polymorphism comes from two Greek words:

Output:

TEXT

Copy

```
Poly = Many
Morph = Forms
```

So, Polymorphism means "Many Forms."

Example:

PYTHON

Copy

```
class Dog:
    def sound(self):
        print("Dog barks")

class Cat:
    def sound(self):
        print("Cat meows")

dog = Dog()
cat = Cat()
dog.sound()
cat.sound()
```

Output:

TEXT

Copy

```
Dog barks
Cat meows
```

- Both classes have a method named `sound()`.
- The same method name performs different actions.
- This is called Polymorphism.

Why Do We Use Polymorphism?

Polymorphism allows us to use the same interface (method name) for different objects without changing the calling code.

Without Polymorphism:

PYTHON

Copy

```
class Dog:
    def dog_sound(self):
        print("Dog barks")

class Cat:
    def cat_sound(self):
        print("Cat meows")

dog = Dog()
cat = Cat()
dog.dog_sound()
cat.cat_sound()
```

Different method names must be remembered.

With Polymorphism:

PYTHON

Copy

```
class Dog:
    def sound(self):
        print("Dog barks")

class Cat:
    def sound(self):
        print("Cat meows")

dog = Dog()
cat = Cat()
dog.sound()
cat.sound()
```

Now every object uses the same method name `sound()`.

How Does Polymorphism Work?

Different classes define the same method name, but each class provides its own implementation.

Example:

PYTHON

Copy

```
class Bird:
    def move(self):
        print("Bird flies")

class Fish:
    def move(self):
        print("Fish swims")

bird = Bird()
fish = Fish()
bird.move()
fish.move()
```

Output:

PYTHON

Copy

```
Bird flies
Fish swims
```

Real-World Examples of Polymorphism

Object	Same Method	Different Behavior
Dog	<code>sound()</code>	Barks
Cat	<code>sound()</code>	Meows
Cow	<code>sound()</code>	Moos
Bird	<code>move()</code>	Flies
Fish	<code>move()</code>	Swims

Built-in Example of Polymorphism

Python's built-in functions also demonstrate polymorphism.

Example:

PYTHON

Copy

```
print(len("Python"))  
print(len([10, 20, 30]))
```

Output:

TEXT

Copy

```
6  
3
```

- The same `len()` function works with different data types.
- String → Number of characters
- List → Number of elements

Placement Quick Points

- Polymorphism means "Many Forms."
- The same method name can perform different actions.
- Different classes can define the same method differently.
- Polymorphism improves code flexibility and readability.
- Python supports polymorphism through methods, operators, and built-in functions like `len()`.

Interview Questions

Q1. What is polymorphism?

Polymorphism is an OOP concept where the same method or operation performs different actions depending on the object.

Q2. What does the word "Polymorphism" mean?

It means "Many Forms."

Q3. Why is polymorphism used?

It allows different objects to use the same method name while providing different implementations.

Q4. Give a real-world example of polymorphism.

Different animals respond differently to the same action `sound()`:

- Dog → Barks
- Cat → Meows
- Cow → Moos

Q5. Give an example of built-in polymorphism in Python.

The `len()` function works with different data types.

PYTHON

Copy

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

Practice

1. Create classes `Dog` and `Cat`, each with a `sound()` method.
2. Create classes `Bird` and `Fish`, each with a `move()` method.
3. Write a program where `Car` and `Bike` both have a `start()` method.
4. Use the `len()` function with a string and a list.
5. Identify the polymorphic behavior in the following code:

TEXT

Copy

```
class Circle:
    def draw(self):
        print("Drawing Circle")

class Rectangle:
    def draw(self):
        print("Drawing Rectangle")

circle = Circle()
rectangle = Rectangle()
circle.draw()
rectangle.draw()
```

Explain why this is an example of polymorphism.

Method Overloading and Method Overriding

What are Method Overloading and Method Overriding?

Method Overloading and Method Overriding are two important concepts of Polymorphism.

Method Overloading means creating multiple methods with the same name but different parameters in languages that support traditional overloading.

Method Overriding means a child class provides its own implementation of a method that already exists in the parent class.

Why Do We Use Method Overloading and Method Overriding?

These concepts allow us to write flexible and reusable code.

- Method Overloading allows the same method to work with different inputs.
- Method Overriding allows a child class to customize the behavior of a parent class.

Method Overloading

Method Overloading means creating multiple methods with the same name but different parameters.

Note: Python does not support traditional method overloading like Java or C++. If multiple methods have the same name, the last one overrides the previous ones.

Example:

PYTHON

Copy

```
class Calculator:
    def add(self, a, b):
        return a + b
    def add(self, a, b, c):
        return a + b + c

calc = Calculator()
print(calc.add(10, 20, 30))
```

Output:

PYTHON

Copy

```
60
```

Only the last `add()` method is available.

Achieving Similar Behavior in Python:

Python commonly uses default arguments to achieve similar behavior.

PYTHON

Copy

```
class Calculator:
    def add(self, a, b, c=0):
        return a + b + c

calc = Calculator()
print(calc.add(10, 20))
print(calc.add(10, 20, 30))
```

Output:

PYTHON

Copy

```
30
60
```

Method Overriding

Method Overriding occurs when a child class defines a method with the same name as the parent class. The child class method replaces the parent's implementation.

Example:

PYTHON

Copy

```
class Animal:
    def sound(self):
        print("Animal makes a sound")

class Dog(Animal):
    def sound(self):
        print("Dog barks")

dog = Dog()
dog.sound()
```

Output:

PYTHON

Copy

```
Dog barks
```

- `Animal` has a `sound()` method.
- `Dog` defines its own `sound()` method.
- The child class method overrides the parent class method.

Calling Parent Methods with `super()`

The `super()` function lets a child class call a method from its parent class. It is most useful for reusing the parent's `__init__()` and for extending an overridden method instead of fully replacing it.

Calling the parent constructor from the child constructor:

TEXT

Copy

```
class Animal:
    def __init__(self, name):
        self.name = name

class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__(name)
        self.breed = breed

dog = Dog("Tommy", "Labrador")
print(dog.name)
print(dog.breed)
```

Output:

PYTHON

Copy

```
Tommy
Labrador
```

- `super().__init__(name)` runs the parent's `__init__()` and sets `self.name`.
- The child then adds its own attribute `self.breed`.

Extending an overridden method with `super()`:

TEXT

Copy

```
class Animal:
    def sound(self):
        print("Animal makes a sound")

class Dog(Animal):
    def sound(self):
        super().sound()
        print("Dog barks")

dog = Dog()
dog.sound()
```

Output:

CODE

Copy

```
Animal makes a sound
Dog barks
```

- `super().sound()` runs the parent's version first.
- The child then adds its own behavior, so both messages appear.

Method Overloading vs Method Overriding

Method Overloading	Method Overriding
Same method name with different parameters	Same method name in parent and child classes
Used to handle different inputs	Used to change inherited behavior
Traditional overloading is not supported in Python	Fully supported in Python
Similar behavior is achieved using default arguments	Achieved using inheritance

Placement Quick Points

- Method Overloading and Method Overriding are concepts of Polymorphism.
- Python does not support traditional method overloading.
- Default arguments are commonly used to handle different numbers of inputs.
- Method Overriding requires inheritance.
- In overriding, the child class provides its own implementation of the parent method.

Interview Questions

Q1. What is Method Overloading?

Method Overloading means creating methods with the same name but different parameters. Python does not support traditional method overloading.

Q2. Does Python support Method Overloading?

No. Python does not support traditional method overloading. Similar behavior can be achieved using default arguments.

Q3. What is Method Overriding?

Method Overriding is when a child class provides its own implementation of a method that already exists in the parent class.

Q4. Does Method Overriding require inheritance?

Yes. Method Overriding is possible only through inheritance.

Q5. What is the difference between Method Overloading and Method Overriding?

Method overloading means using the same method name for different inputs. Method overriding means a child class redefines a method inherited from a parent class.

Practice

1. Create a `Calculator` class that uses default arguments to add two or three numbers.
2. Create a parent class `Animal` with a `sound()` method and override it in a child class `Dog`.
3. Create a parent class `Vehicle` with a `start()` method and override it in a child class `Car`.
4. Explain why Python uses default arguments instead of traditional method overloading.
5. Identify whether the following code demonstrates Method Overloading or Method Overriding:

CODE

Copy

```
class Animal:
    def sound(self):
        print("Animal Sound")

class Cat(Animal):
    def sound(self):
        print("Meow")

cat = Cat()
cat.sound()
```

Explain your answer.

Duck Typing

What is Duck Typing?

Duck Typing is a feature of Python where the type of an object is less important than the methods or behaviors it provides.

In simple words, if an object behaves like the expected object, Python allows it to be used, regardless of its actual class.

This idea comes from the famous saying:

"If it looks like a duck, swims like a duck, and quacks like a duck, then it is a duck."

Example:

CODE

Copy

```
class Dog:
    def sound(self):
        print("Dog barks")

class Cat:
    def sound(self):
        print("Cat meows")

def make_sound(animal):
    animal.sound()

dog = Dog()
cat = Cat()
make_sound(dog)
make_sound(cat)
```

Output:

CODE

Copy

```
Dog barks  
Cat meows
```

- `Dog` and `Cat` are different classes.
- Both have a `sound()` method.
- The function `make_sound()` doesn't check the object's class.
- It simply calls `sound()`.
- This is called Duck Typing.

Why Do We Use Duck Typing?

Duck Typing makes code more flexible because a function can work with different types of objects as long as they provide the required method.

Without Duck Typing:

CODE

Copy

```
class Dog:  
    def sound(self):  
        print("Dog barks")  
  
dog = Dog()  
dog.sound()
```

This code works only with `Dog` objects.

With Duck Typing:

CODE

Copy

```
class Dog:
    def sound(self):
        print("Dog barks")

class Cat:
    def sound(self):
        print("Cat meows")

def make_sound(animal):
    animal.sound()

make_sound(Dog())
make_sound(Cat())
```

Now the same function works with multiple objects.

How Does Duck Typing Work?

Duck Typing works by checking whether an object has the required method when the program runs.

Example:

CODE

Copy

```
class Bird:
    def move(self):
        print("Bird flies")

class Fish:
    def move(self):
        print("Fish swims")

def travel(obj):
    obj.move()

travel(Bird())
travel(Fish())
```

Output:

The `travel()` function works because both objects provide a `move()` method.

Real-World Example

Suppose a remote control can operate different devices.

CODE

Copy

```
class TV:
    def turn_on(self):
        print("TV Turned On")

class AC:
    def turn_on(self):
        print("AC Turned On")

def power_on(device):
    device.turn_on()

power_on(TV())
power_on(AC())
```

Output:

CODE

Copy

```
TV Turned On
AC Turned On
```

The `power_on()` function doesn't care whether the object is a `TV` or an `AC`. It only expects the object to have a `turn_on()` method.

Duck Typing vs Inheritance

Duck Typing	Inheritance
Focuses on an object's behavior	Focuses on parent-child relationships
No inheritance is required	Requires inheritance
Works if the required method exists	Child class inherits methods from the parent
More flexible	More structured

Placement Quick Points

- Duck Typing is based on an object's behavior, not its type.
- Python checks whether an object has the required method at runtime.
- Objects from different classes can be used interchangeably if they provide the same behavior.
- Duck Typing makes code more flexible and reusable.
- In Duck Typing, what an object can do is more important than what it is.

Interview Questions

Q1. What is Duck Typing?

Duck Typing is a feature where Python uses an object based on its behavior (methods) instead of its class type.

Q2. Why is it called Duck Typing?

It comes from the saying: "If it looks like a duck, swims like a duck, and quacks like a duck, then it is a duck."

Q3. Does Duck Typing require inheritance?

No. Duck Typing does not require inheritance.

Q4. What does Duck Typing focus on?

Duck Typing focuses on whether an object provides the required methods.

Q5. What is the main advantage of Duck Typing?

It makes programs more flexible because the same function can work with different types of objects.

Practice

1. Create classes `Dog` and `Cat` with a `sound()` method and write a function `make_sound()` that works with both.
2. Create classes `Car` and `Bike` with a `start()` method and write a function `start_vehicle()` that calls `start()`.
3. Create classes `TV` and `AC` with a `turn_on()` method and write a common function to turn on both devices.
4. Explain why the following code demonstrates Duck Typing:

CODE

[Copy](#)

```
class Bird:
    def move(self):
        print("Bird flies")

class Fish:
    def move(self):
        print("Fish swims")

def travel(obj):
    obj.move()
```

5. What will happen if an object passed to `travel()` does not have a `move()` method? Explain why.

Abstraction

What Is Abstraction?

Abstraction is an Object-Oriented Programming (OOP) concept that hides the implementation details and shows only the essential features of an object.

In simple words, the user knows what an object does, but not how it does it.

Example:

CODE

Copy

```
from abc import ABC, abstractmethod
class Animal(ABC):
    @abstractmethod
    def sound(self):
        pass
class Dog(Animal):
    def sound(self):
        print("Dog barks")
dog = Dog()
dog.sound()
```

Output:

CODE

Copy

```
Dog barks
```

- `Animal` is an abstract class.
- `sound()` is an abstract method.
- `Dog` provides the implementation of `sound()`.
- The user only calls `dog.sound()` without worrying about how it is implemented.

Why Do We Use Abstraction?

Abstraction hides unnecessary implementation details and exposes only the required functionality.

Without Abstraction:

CODE

Copy

```
class Dog:
    def sound(self):
        print("Dog barks")
dog = Dog()
dog.sound()
```

Everything is implemented directly in one class.

With Abstraction:

CODE

Copy

```
from abc import ABC, abstractmethod
class Animal(ABC):
    @abstractmethod
    def sound(self):
        pass
class Dog(Animal):
    def sound(self):
        print("Dog barks")
dog = Dog()
dog.sound()
```

The parent class defines what should be done, while the child class defines how it is done.

Understanding `abc`, `ABC`, and `@abstractmethod`

Before creating an abstract class, let's understand the keywords used.

Keyword	Purpose
<code>abc</code>	A built-in Python module used for creating abstract classes
<code>ABC</code>	A class from the <code>abc</code> module used as the base class for abstract classes
<code>@abstractmethod</code>	A decorator that marks a method as abstract

Import Statement:

CODE

Copy

```
from abc import ABC, abstractmethod
```

- `abc` is Python's Abstract Base Class module.
- `ABC` is used to create an abstract class.
- `@abstractmethod` tells Python that a method must be implemented in every child class.
- Without importing these, Python cannot create abstract classes.

Abstract Class

An abstract class is a class meant to act as a blueprint for other classes. In Python, a class with at least one `@abstractmethod` cannot be instantiated until all abstract methods are implemented by a child class.

To create an abstract class, inherit from `ABC`.

Example:

CODE

Copy

```
from abc import ABC
class Animal(ABC):
    pass
```

- `Animal` is based on `ABC`, so it is meant to be used as a blueprint.
- Because it has no abstract method here, Python can still create an object from it.

Abstract Method

An abstract method is a method declared using the `@abstractmethod` decorator.

It usually does not contain the final implementation in the abstract class and must be implemented by every concrete child class.

Example:

CODE

Copy

```
from abc import ABC, abstractmethod
class Animal(ABC):
    @abstractmethod
    def sound(self):
        pass
class Dog(Animal):
    def sound(self):
        print("Dog barks")
dog = Dog()
dog.sound()
```

Output:

CODE

Copy

```
Dog barks
```

- `@abstractmethod` tells Python that `sound()` is incomplete.
- Every child class must define its own version of `sound()`.
- Otherwise, Python will not allow an object of that child class to be created.

Placement Quick Points

- Abstraction hides implementation details and shows only essential features.
- Python provides the `abc` module to support abstraction.
- `ABC` is used to create an abstract class.
- `@abstractmethod` is used to create an abstract method.
- A class with unimplemented abstract methods cannot be instantiated.
- Every child class must implement all abstract methods.

Interview Questions

Q1. What is abstraction?

Abstraction is the process of hiding implementation details and exposing only the essential features of an object.

Q2. Which module is used for abstraction in Python?

The `abc` module.

Q3. What is `ABC`?

`ABC` (Abstract Base Class) is the base class used to create abstract classes.

Q4. What is `@abstractmethod`?

`@abstractmethod` is a decorator that marks a method as abstract and requires child classes to implement it.

Q5. Can we create an object of an abstract class?

Not if it still has unimplemented abstract methods.

Q6. Why do we use abstract classes?

To define a common blueprint that child classes must follow.

Practice

1. Import `ABC` and `abstractmethod` from the `abc` module.
2. Create an abstract class `Animal`.
3. Add an abstract method `sound()` to the class.
4. Create a child class `Dog` and implement the `sound()` method.
5. Create another child class `Cat` and implement the `sound()` method.
6. Identify the abstract class and the abstract method in the following code:

CODE

Copy

```
from abc import ABC, abstractmethod
class Vehicle(ABC):
    @abstractmethod
    def start(self):
        pass
class Car(Vehicle):
    def start(self):
        print("Car Started")
```

Inner Classes

What Are Inner Classes?

An Inner Class is a class that is defined inside another class. The outer class is called the Outer Class, and the class defined inside it is called the Inner Class.

Inner classes are used when one class is closely related to another class and is not intended to be used independently.

Example:

CODE

Copy

```
class College:
    class Student:
        def display(self):
            print("Student Details")
student = College.Student()
student.display()
```

Output:

CODE

Copy

```
Student Details
```

- `College` is the Outer Class.

- `Student` is the Inner Class.
- The inner class is accessed using `OuterClass.InnerClass()`.

Why Do We Use Inner Classes?

Inner classes help organize related classes together and improve code readability.

Without Inner Class:

CODE

Copy

```
class Student:
    def display(self):
        print("Student Details")
student = Student()
student.display()
```

`Student` exists as a separate class.

With Inner Class:

CODE

Copy

```
class College:
    class Student:
        def display(self):
            print("Student Details")
student = College.Student()
student.display()
```

Now, `Student` clearly belongs to `College`.

How to Create an Inner Class?

Define one class inside another class.

Syntax:

CODE

Copy

```
class OuterClass:
    class InnerClass:
        pass
```

Example:

CODE

Copy

```
class Company:
    class Employee:
        def work(self):
            print("Employee is working")
employee = Company.Employee()
employee.work()
```

Creating an Object of an Inner Class

To create an object of an inner class, use the outer class name followed by the inner class name.

Syntax:

CODE

Copy

```
class OuterClass:
    object_name = OuterClass.InnerClass()
```

Example:

CODE

Copy

```
class School:
    class Student:
        def study(self):
            print("Studying")
student = School.Student()
student.study()
```

Output:

CODE

Copy

Studying

Real-World Example

Suppose a `Car` has an `Engine`. The engine is closely related to the car and usually doesn't exist independently.

CODE

Copy

```
class Car:
    class Engine:
        def start(self):
            print("Engine Started")
engine = Car.Engine()
engine.start()
```

Output:

CODE

Copy

Engine Started

Inner Class vs Normal Class

Inner Class	Normal Class
Defined inside another class	Defined independently
Used for closely related classes	Used independently
Accessed using <code>OuterClass.InnerClass()</code>	Accessed directly using the class name
Improves code organization	General-purpose class

Placement Quick Points

- An Inner Class is a class defined inside another class.
- The enclosing class is called the Outer Class.
- Inner classes help group closely related classes together.
- An inner class object is created using `OuterClass.InnerClass()`.
- Inner classes improve code organization and readability.

Interview Questions

Q1. What is an Inner Class?

An Inner Class is a class defined inside another class.

Q2. Why are Inner Classes used?

They are used to group closely related classes and improve code organization.

Q3. How do you create an object of an Inner Class?

Using the syntax `OuterClass.InnerClass()`.

Q4. Can an Inner Class exist without an Outer Class?

No. It is defined inside an outer class and is typically accessed through it.

Q5. Give a real-world example of an Inner Class.

A `Car` containing an `Engine`, or a `College` containing a `Student`.

Practice

1. Create an outer class `College` and an inner class `Student`.
2. Create an object of the inner class and call its method.
3. Create an outer class `Company` and an inner class `Employee` with a `work()` method.
4. Create an outer class `Car` and an inner class `Engine` with a `start()` method.

5. Identify the Outer Class and the Inner Class in the following code:

CODE

Copy

```
class Library:
    class Book:
        def read(self):
            print("Reading Book")
```

6. Write the statement to create an object of the `Book` inner class and call the `read()` method.

End of Level 9

Advanced Python

Level: 10 | Version: v1.0 | Author: Meptrasoft

Real programs process large amounts of data — thousands of orders, huge files, endless data streams. Loading everything into memory at once is often wasteful, and repeating the same setup code (logging, timing, checks) around many functions is messy. **Iterators**, **generators**, and **decorators** solve exactly these problems. This module uses a **Food Delivery Order System** as the running example.

PYTHON

Copy

```
orders = [  
    {"food": "Burger", "price": 200, "status": "Confirmed"},  
    {"food": "Pizza", "price": 300, "status": "Pending"},  
    {"food": "Pasta", "price": 250, "status": "Confirmed"},  
]
```

Iterators

Why Iterators Matter

A `for` loop already lets us walk through a list. Behind the scenes, Python is using something called an **iterator** to do it — understanding that machinery is the foundation for generators later in this module.

PYTHON

Copy

```
foods = ["Burger", "Pizza", "Pasta"]  
for food in foods:  
    print(food)
```

Output:

TEXT

Copy

Burger
Pizza
Pasta

Iterable vs Iterator

An **iterable** is anything that can be looped over (a list, tuple, string, dictionary, set). An **iterator** is the object that actually remembers *where it is* and produces the values **one at a time**.

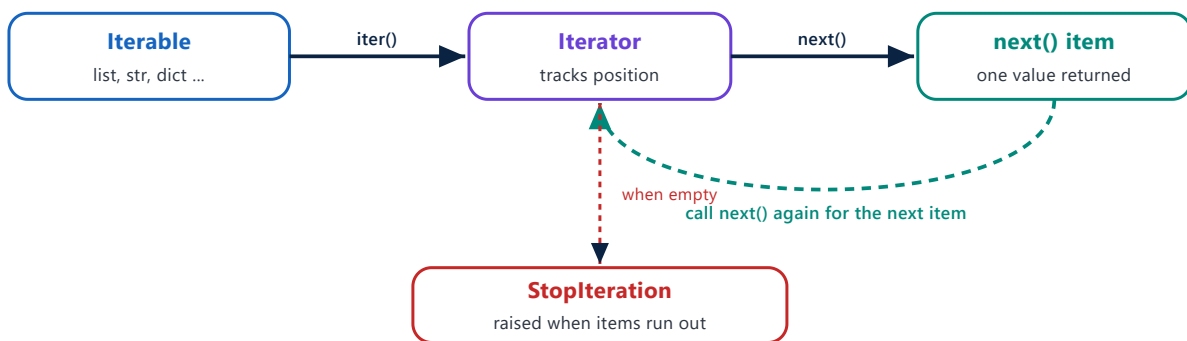
TEXT

Copy

ITERABLE → CAN BE LOOPED OVER (list, tuple, string, dict, set)
ITERATOR → REMEMBERS POSITION → PRODUCES ONE VALUE AT A TIME

You can get an iterator from an iterable by using the built-in `iter()` function.

The Iterator Protocol — `iter()` and `next()`



The iterator protocol: `iter()` gives an iterator, `next()` returns items one at a time, and `StopIteration` signals the end.

PYTHON

Copy

```
foods = ["Burger", "Pizza", "Pasta"]
food_iterator = iter(foods)

print(next(food_iterator))    # Burger
print(next(food_iterator))    # Pizza
print(next(food_iterator))    # Pasta
```

Output:

TEXT

Copy

```
Burger
Pizza
Pasta
```

`iter()` returns an iterator; `next()` asks it for the next value.

StopIteration

Calling `next()` after the last value raises `StopIteration` — this is how Python knows a loop has ended.

PYTHON

Copy

```
foods = ["Burger"]
food_iterator = iter(foods)
next(food_iterator)    # Burger
try:
    next(food_iterator)    # Raises StopIteration
except StopIteration:
    print("StopIteration raised!")
```

Output:

TEXT

Copy

```
StopIteration raised!
```

A `for` loop hides this by catching `StopIteration` automatically:

PYTHON

Copy

```
foods = ["Burger", "Pizza", "Pasta"]
food_iterator = iter(foods)

while True:
    try:
        food = next(food_iterator)
        print(food)
    except StopIteration:
        break
```

Output:

TEXT

Copy

```
Burger
Pizza
Pasta
```

TEXT

Copy

```
for loop → calls iter() ONCE → calls next() REPEATEDLY → stops on StopIteration
```

Building a Custom Iterator

Any class can become an iterator by implementing two special methods: `__iter__()` (returns the iterator itself) and `__next__()` (returns the next value, or raises `StopIteration`).

PYTHON

Copy

```
class OrderIterator:
    def __init__(self, orders):
        self.orders = orders
        self.index = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.index >= len(self.orders):
            raise StopIteration
        order = self.orders[self.index]
        self.index += 1
        return order

foods = OrderIterator(["Burger", "Pizza", "Pasta"])
for food in foods:
    print(food)
```

Output:

TEXT

Copy

```
Burger
Pizza
Pasta
```

Because `OrderIterator` implements `__iter__` and `__next__`, a normal `for` loop works on it exactly as it would on a list.

TEXT

Copy

```
__iter__() → RETURNS THE ITERATOR OBJECT
__next__() → RETURNS NEXT VALUE OR RAISES StopIteration
```

Iterable vs Iterator — Comparison

Iterable	Iterator
Has <code>__iter__()</code>	Has <code>__iter__()</code> and <code>__next__()</code>
Example: list, string, dict, tuple	Example: object returned by <code>iter(list)</code>
Can be looped over many times	Gets exhausted after one full pass
Does not track position itself	Remembers its current position

TEXT

Copy

ITERABLE → `iter()` → ITERATOR → `next()` → VALUES, ONE AT A TIME

Generators

Why Generators Matter

Imagine the food delivery platform has to process **one million** orders. Building a list of one million processed orders keeps all of them in memory at once. A **generator** produces values **one at a time, on demand**, so only one processed order needs to exist in memory at a time.

TEXT

Copy

LIST → ALL VALUES BUILT AND STORED IN MEMORY AT ONCE
 GENERATOR → ONE VALUE PRODUCED AT A TIME, ONLY WHEN ASKED

A generator is the simplest way to create an iterator — Python writes the `__iter__`/`__next__` machinery for you.

Generator Functions and `yield`

A **generator function** looks like a normal function but uses `yield` to produce values instead of returning all values at once.

PYTHON

Copy

```
def order_generator(orders):  
    for food in orders:  
        yield food  
  
foods = ["Burger", "Pizza", "Pasta"]  
gen = order_generator(foods)  
print(gen)
```

Output:

TEXT

Copy

```
<generator object order_generator at 0x000002B6AB249BE0>
```

Calling `order_generator(foods)` does **not** run the function body — it returns a **generator object**. The body only runs when values are pulled out of it with `next()` or a `for` loop.

PYTHON

Copy

```
def order_generator():  
    yield "Burger"  
    yield "Pizza"  
    yield "Pasta"  
  
gen = order_generator()  
print(next(gen))  
print(next(gen))  
print(next(gen))
```

Output:

TEXT

Copy

Burger
Pizza
Pasta

TEXT

Copy

yield → PAUSES the function, remembers exactly where it stopped, resumes on the next request

return VS yield

return	yield
Ends the function completely	Pauses the function, keeps its state
Sends back one value, once	Produces a value each time it is reached
Calling the function runs its body	Calling the function returns a generator
Function → normal value	Function → generator object

Looping Over a Generator Directly

Just like any iterator, a generator works naturally in a `for` loop:

PYTHON

Copy

```
def order_generator(orders):
    for food in orders:
        yield food

for food in order_generator(["Burger", "Pizza", "Pasta"]):
    print(food)
```

Output:

TEXT

Copy

Burger
Pizza
Pasta

Generator Expressions

A **generator expression** looks like a list comprehension but uses `()` instead of `[]`, and produces values lazily instead of building a full list.

PYTHON

Copy

```
prices = [200, 300, 150]
priced_with_tax = (price * 1.05 for price in prices)
print(priced_with_tax)
```

Output:

TEXT

Copy

<generator object <genexpr> at 0x000002B6AB249E50>

PYTHON

Copy

```
prices = [100, 200, 350]
priced_with_tax = (p * 1.05 for p in prices)
print(sum(priced_with_tax))
```

Output:

TEXT

Copy

682.5

List Comprehension vs Generator Expression

List Comprehension [...]	Generator Expression (...)
Builds the whole list immediately	Produces values one at a time, lazily
Uses more memory for large data	Uses far less memory for large data
Can be looped over many times	Gets exhausted after one full pass

TEXT

Copy

```
[...] → LIST NOW, IN MEMORY  
(...) → VALUES LATER, ON DEMAND
```

A Generator Runs Only Once

Once a generator has produced all its values, it is **exhausted** — looping over it again produces nothing.

PYTHON

Copy

```
def order_generator(items):  
    yield from items  
  
gen = order_generator(["Burger", "Pizza"])  
print(list(gen))  
print(list(gen))
```

Output:

TEXT

Copy

```
['Burger', 'Pizza']  
[]
```

To iterate again, create a **new** generator by calling the function again.

TEXT

Copy

```
GENERATOR EXHAUSTED AFTER ONE FULL PASS → CALL THE FUNCTION AGAIN FOR A NEW ONE
```

Chaining Generators — an Order Processing Pipeline

Generators can be chained so each stage only pulls one order at a time from the previous stage — no full intermediate lists are ever built, which matters a lot for large order feeds.

PYTHON

Copy

```
def read_orders(orders):
    for order in orders:
        yield order

def filter_confirmed(orders):
    for order in orders:
        if order["status"] == "Confirmed":
            yield order

def add_delivery_fee(orders):
    for order in orders:
        order["final_price"] = order["price"] + 40
        yield order

orders = [
    {"food": "Burger", "price": 200, "status": "Confirmed"},
    {"food": "Pizza", "price": 300, "status": "Pending"},
    {"food": "Pasta", "price": 250, "status": "Confirmed"},
]

pipeline = add_delivery_fee(filter_confirmed(read_orders(orders)))

for order in pipeline:
    print(order["food"], "->", order["final_price"])
```

Output:

TEXT

Copy

```
Burger -> 240
Pasta -> 290
```

Each function only produces one order when the next one is requested — the pipeline never holds the full processed list in memory.

TEXT

Copy

```
read_orders() → filter_confirmed() → add_delivery_fee() → ONE ORDER AT A TIME
```

Iterator vs Generator

Iterator (class-based)	Generator (<code>yield</code> -based)
Written with <code>__iter__</code> and <code>__next__</code>	Written with a normal function and <code>yield</code>
More code, full manual control	Less code, simpler for most cases
Custom state stored on <code>self</code>	State kept automatically by Python

TEXT

Copy

```
GENERATOR → THE EASIEST WAY TO BUILD AN ITERATOR
```

Decorators

Functions Are First-Class Objects

In Python, functions can be assigned to variables, passed as arguments, and returned from other functions — this is exactly what makes decorators possible.

PYTHON

Copy

```
def show_order():
    print("Order Received")

send_notification = show_order
send_notification()
```

Output:

TEXT

Copy

Order Received

PYTHON

Copy

```
def show_order():  
    print("Order Received")  
  
def run_function(func):  
    func()  
  
run_function(show_order)
```

Output:

TEXT

Copy

Order Received

Closures

A **closure** is an inner function that remembers variables from its enclosing function, even after the outer function has finished running.

PYTHON

Copy

```
def make_discount_applier(discount):  
    def apply_discount(price):  
        return price - (price * discount / 100)  
    return apply_discount  
  
ten_percent_off = make_discount_applier(10)  
print(ten_percent_off(200))
```

Output:

TEXT

Copy

180.0

`apply_discount` keeps access to `discount` (`10`) even though `make_discount_applier` already returned. Closures are the mechanism decorators are built on.

TEXT

Copy

CLOSURE → INNER FUNCTION + REMEMBERED OUTER VARIABLE(S)

Building a Basic Decorator

A **decorator** is a function that takes a function and returns a **new** function that adds extra behavior around it — without changing the original function's code.

PYTHON

Copy

```
def log_order(func):
    def wrapper():
        print("Order function starting...")
        func()
        print("Order function finished.")
    return wrapper

def show_order():
    print("Order Received")

show_order = log_order(show_order)
show_order()
```

Output:

TEXT

Copy

```
Order function starting...
Order Received
Order function finished.
```

Using `@` Syntax

Python provides `@decorator_name` as shorthand for reassigning a function to its decorated version.

PYTHON

Copy

```
def log_order(func):
    def wrapper():
        print("Order function starting...")
        func()
        print("Order function finished.")
    return wrapper

@log_order
def show_order():
    print("Order Received")

show_order()
```

Output:

TEXT

Copy

```
Order function starting...
Order Received
Order function finished.
```

TEXT

Copy

```
@log_order
def show_order(): ...
    ↓ exactly the same as ↓
show_order = log_order(show_order)
```

Decorating Functions That Take Arguments

The `wrapper` must accept `*args` and `**kwargs` so it works with **any** decorated function, and it should `return` the original function's result so callers still receive the value.

PYTHON

Copy

```
def log_order(func):
    def wrapper(*args, **kwargs):
        print("Order function starting...")
        result = func(*args, **kwargs)
        print("Order function finished.")
        return result
    return wrapper

@log_order
def calculate_total(price, quantity):
    return price * quantity

print(calculate_total(200, 2))
```

Output:

TEXT

Copy

```
Order function starting...
Order function finished.
400
```

Preserving Metadata — `functools.wraps`

A wrapper hides the original function's name and docstring, which can confuse debugging tools and documentation.

PYTHON

Copy

```
def log_order(func):
    def wrapper(*args, **kwargs):
        return func(*args, **kwargs)
    return wrapper

@log_order
def calculate_total(price, quantity):
    """Calculates the order total."""
    return price * quantity

print(calculate_total.__name__)
```

Output:

TEXT

Copy

wrapper

`functools.wraps` copies the original function's name and docstring onto the wrapper, fixing this.

PYTHON

Copy

```
from functools import wraps

def log_order(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        return func(*args, **kwargs)
    return wrapper

@log_order
def calculate_total(price, quantity):
    """Calculates the order total."""
    return price * quantity

print(calculate_total.__name__)
```

Output:

TEXT

Copy

calculate_total

TEXT

Copy

`@wraps(func)` → KEEPS THE ORIGINAL FUNCTION'S NAME AND DOCSTRING

In real projects, use `@wraps(func)` when writing decorators.

Decorators That Accept Their Own Arguments

Sometimes the decorator itself needs configuration — for example, how many times to **retry** a payment that may fail. This needs **three** levels of functions. This simple retry stops as soon as the call succeeds, and only tries again when the call raises an error.

PYTHON

Copy

```
from functools import wraps

def retry(times):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            for attempt in range(1, times + 1):
                print(f"Attempt {attempt}")
                try:
                    return func(*args, **kwargs) # success → stop retrying
                except Exception as error:
                    print(f"Failed: {error}")
            print("All attempts failed")
        return wrapper
    return decorator

attempts_so_far = 0

@retry(times=3)
def charge_card():
    global attempts_so_far
    attempts_so_far += 1
    if attempts_so_far < 3:
        raise ValueError("network timeout")
    print("Payment successful")

charge_card()
```

Output:

TEXT

Copy

```
Attempt 1
Failed: network timeout
Attempt 2
Failed: network timeout
Attempt 3
Payment successful
```

The loop retries only after a failure and returns immediately on success — so a working call runs the real logic just once.

TEXT

Copy

```
retry(times)      → returns decorator
decorator(func)   → returns wrapper
wrapper(*args, **kw) → runs the real logic
```

Chaining Multiple Decorators

Several decorators can be stacked on one function. Python applies them **bottom-up** when building the final function. When the final function is called, the outer wrapper starts first.

PYTHON

Copy

```
from functools import wraps

def log_order(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print("Logging order...")
        return func(*args, **kwargs)
    return wrapper

def check_auth(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print("Checking authentication...")
        return func(*args, **kwargs)
    return wrapper

@log_order
@check_auth
def place_order(food):
    print("Order placed for", food)

place_order("Burger")
```

Output:

TEXT

Copy

```
Logging order...
Checking authentication...
Order placed for Burger
```

TEXT

Copy

```
@log_order
@check_auth
def place_order(): ...
    ↓ exactly the same as ↓
place_order = log_order(check_auth(place_order))
```

Real-World Uses of Decorators

Use Case	What the Decorator Does
Logging	Prints/records details before and after a call
Timing	Measures how long a function takes to run
Authentication	Checks permissions before allowing a call
Caching	Stores results so repeated calls are faster
Retry	Re-attempts a failing operation
Validation	Checks arguments before the real function runs

PYTHON

Copy

```
import time
from functools import wraps

def timer(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(f"{func.__name__} took {end - start:.4f} seconds")
        return result
    return wrapper

@timer
def process_orders(orders):
    return sum(order["price"] for order in orders)

orders = [{"price": 200}, {"price": 300}, {"price": 150}]
print(process_orders(orders))
```

Output:

TEXT

Copy

```
process_orders took 0.0000 seconds
650
```

(The exact number will differ on your machine and on every run — only the format `x.xxxx` seconds matters.)

Built-in Decorators on Classes — Quick Look

Python also provides a few built-in decorators used inside classes.

Decorator	Purpose
<code>@staticmethod</code>	A method that needs neither <code>self</code> nor the class
<code>@classmethod</code>	A method that receives the class (<code>cls</code>) instead of an instance
<code>@property</code>	Lets a method be accessed like a plain attribute

TEXT

Copy

```

@staticmethod → NO self, NO cls
@classmethod → RECEIVES cls (the class itself)
@property → METHOD USED LIKE AN ATTRIBUTE (no parentheses)

```

Complete Example — Iterators, Generators, and Decorators Together

PYTHON

Copy

```
from functools import wraps

def log_order(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print("Starting order report...\n")
        result = func(*args, **kwargs)
        print("\nReport complete.")
        return result
    return wrapper

def read_orders(orders):
    for order in orders:
        yield order

def filter_confirmed(orders):
    for order in orders:
        if order["status"] == "Confirmed":
            yield order

@log_order
def generate_report(orders):
    total = 0
    for order in filter_confirmed(read_orders(orders)):
        print(order["food"], "-", order["price"])
        total += order["price"]
    return total

orders = [
    {"food": "Burger", "price": 200, "status": "Confirmed"},
    {"food": "Pizza", "price": 300, "status": "Pending"},
    {"food": "Pasta", "price": 250, "status": "Confirmed"},
]

print("Grand Total:", generate_report(orders))
```

Output:

TEXT

Copy

```
Starting order report...
```

```
Burger - 200
```

```
Pasta - 250
```

```
Report complete.
```

```
Grand Total: 450
```

`generate_report` is wrapped by a **decorator** (`log_order`), consumes a lazy **generator pipeline** (`read_orders` → `filter_confirmed`), and each stage is walked using the same iterator protocol that powers every `for` loop in Python.



Common Fresher Mistakes

X Mistake	✓ Correct Understanding
Thinking a list and an iterator are the same thing	A list is iterable; an iterator is what <code>iter()</code> produces from it
Calling <code>next()</code> after exhaustion and being surprised by an error	<code>StopIteration</code> is expected — that is how loops know to stop
Thinking calling a generator function runs its body immediately	It only returns a generator object; the body runs on <code>next()</code> /iteration
Reusing an already-exhausted generator	Create a new generator by calling the function again
Using <code>[...]</code> when data is huge	Prefer <code>(...)</code> (generator expression) to save memory
Forgetting <code>*args, **kwargs</code> in a decorator's <code>wrapper</code>	Without them the decorator breaks on any decorated function with arguments
Forgetting to <code>return</code> the result inside <code>wrapper</code>	Callers get <code>None</code> instead of the real value
Forgetting <code>@wraps(func)</code> in a real decorator	The decorated function's name/docstring get overwritten
Confusing decorator order when chaining	Decorators are applied bottom-up; during the call, the outer wrapper starts first

Placement Quick Points

- An **iterable** can be looped over; an **iterator** remembers its position and produces values one at a time.

- `iter()` returns an iterator; `next()` asks it for the next value; `StopIteration` signals the end.
- A custom class becomes an iterator by implementing `__iter__()` and `__next__()`.
- A **generator function** uses `yield` and is the simplest way to build an iterator.
- Calling a generator function returns a generator object — the body runs lazily, on demand.
- A generator expression uses `(...)` and produces values lazily, unlike a list comprehension's `[...]`.
- A generator is exhausted after one full pass; a new one must be created to iterate again.
- Generators can be chained into a pipeline that processes one item at a time, saving memory.
- A **decorator** is a function that takes a function and returns a new function with added behavior.
- `@decorator` above a function is shorthand for `function = decorator(function)`.
- A decorator's `wrapper` should accept `*args, **kwargs` and `return` the wrapped function's result.
- `functools.wraps` preserves the original function's name and docstring.
- Decorators that take their own arguments need three nested function levels.
- Chained decorators are applied bottom-up when the function is built; when called, the outer wrapper starts first.

Interview Questions

Q1. What is an iterable?

An object that can be looped over, such as a list, tuple, string, or dictionary.

Q2. What is an iterator?

An object that remembers its position and produces values one at a time using `__next__()`.

Q3. What is the difference between an iterable and an iterator?

An iterable can be looped over; an iterator is created from it with `iter()` and tracks position while producing values.

Q4. What does `next()` do?

It returns the next value from an iterator, or raises `StopIteration` when there are no more values.

Q5. What is `StopIteration`?

The exception an iterator raises to signal that no more values remain.

Q6. What two methods make a class an iterator?

`__iter__()`, which returns the iterator itself, and `__next__()`, which returns the next value.

Q7. What is a generator?

An iterator-like object that produces values one at a time. A generator function creates it using `yield`.

Q8. What happens when a generator function is called?

It does not run the function body; it returns a generator object. The body runs as values are requested.

Q9. What is the difference between `return` and `yield`?

`return` ends the function and sends back one value. `yield` pauses the function, remembers its state, and can produce more values later.

Q10. What is a generator expression?

An expression like `(x for x in iterable)` that produces values lazily, similar to a list comprehension but memory-efficient.

Q11. Can a generator be iterated over more than once?

No. Once exhausted, a new generator must be created by calling the function again.

Q12. Why are generators memory-efficient?

They produce one value at a time instead of building the entire result in memory upfront.

Q13. What is a closure?

An inner function that remembers variables from its enclosing function even after that function has finished running.

Q14. What is a decorator?

A function that takes another function and returns a new function that adds behavior around it.

Q15. What does `@decorator_name` above a function do?

It reassigns the function to the value returned by `decorator_name(function)`.

Q16. Why should a decorator's `wrapper` accept `*args, **kwargs`?

So the decorator works with any function regardless of how many arguments it takes.

Q17. Why should `wrapper` return the result of the original function?

So callers of the decorated function still receive the correct return value.

Q18. What does `functools.wraps` do?

It copies the original function's name and docstring onto the wrapper function.

Q19. How does a decorator accept its own arguments?

By adding an extra outer function level: the outer function takes the decorator's arguments and returns the actual decorator.

Q20. In what order are chained decorators applied?

They are applied bottom-up when Python builds the final function. When the function is called, the outer wrapper starts first.

Practice

Use a Food Delivery Order System.

1. Create a list of food items and loop over it with a `for` loop.
2. Create an iterator from that list using `iter()`.
3. Call `next()` on the iterator three times and print each result.
4. Call `next()` one time too many and observe `StopIteration`.
5. Rewrite the loop manually using `while True`, `next()`, and `try/except StopIteration`.
6. Build a custom `OrderIterator` class with `__iter__` and `__next__`.
7. Use the custom iterator in a `for` loop.
8. Explain the difference between an iterable and an iterator.
9. Write a generator function `order_generator()` that yields each food item.
10. Print the generator object before consuming it.
11. Consume the generator with `next()` and then with a `for` loop.
12. Write a generator expression that adds 5% tax to a list of prices.
13. Compare the generator expression to an equivalent list comprehension.
14. Try iterating over the same generator twice and observe the result.
15. Build a three-stage generator pipeline: read orders, filter confirmed orders, add a delivery fee.
16. Explain why the pipeline does not build an intermediate list at each stage.
17. Write a closure `make_discount_applier(discount)` that returns a function applying that discount.
18. Write a basic decorator `log_order` that prints messages before and after calling a function.
19. Apply it using `@log_order` syntax.
20. Update the decorator's `wrapper` to accept `*args, **kwargs`.
21. Verify the decorated function still returns the correct value.
22. Add `functools.wraps` and confirm `__name__` is preserved.
23. Write a decorator `retry(times)` that accepts its own argument.
24. Chain two decorators (`log_order` and `check_auth`) on one function and predict the print order.
25. Write a `timer` decorator that prints how long a function took to run.
26. Combine a decorator, a generator pipeline, and a `for` loop in one complete order-report example.

Predict the output:

PYTHON

Copy

```
def numbers():
    yield 1
    yield 2
    yield 3

gen = numbers()
print(next(gen))
print(next(gen))
print(list(gen))
```

PYTHON

Copy

```
def log(func):
    def wrapper(*args, **kwargs):
        print("before")
        result = func(*args, **kwargs)
        print("after")
        return result
    return wrapper

@log
def add(a, b):
    return a + b

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

PYTHON

Copy

```
gen = (x * 2 for x in [1, 2, 3])
print(list(gen))
print(list(gen))
```

Find the mistake:

PYTHON

Copy

```
def log(func):
    def wrapper():
        print("before")
        func()
        print("after")
    return wrapper

@log
def show_total(total):
    print(total)

show_total(400)
```

Final task: Build a small **Food Delivery Order Report** that reads a list of orders through a chained generator pipeline (filter confirmed orders, add a delivery fee), wraps the reporting function with a `@log_order` decorator that prints a start/end message, and prints each order followed by the grand total.

Output:

TEXT

Copy

ITERATOR	→ REMEMBERS POSITION, ONE VALUE AT A TIME
GENERATOR	→ SIMPLEST WAY TO BUILD AN ITERATOR, USES <code>yield</code>
DECORATOR	→ WRAPS A FUNCTION TO ADD BEHAVIOR AROUND IT

End of Level 10