Python Programming

Subject Code: DI02032011 / 1323203

Diploma in Engineering — ICT Branch

Gujarat Technological University

Comprehensive Detailed Textbook
With Full PYQ Solutions (Su'23 – W'25)

Milav Dabgar
AICTE QIP PG Certificate in Deep Learning (SVNIT Surat)
Diploma in Data Science (IIT Madras)

June 18, 2026

Preface

This textbook is designed for students of the **Python Programming** course (DI02032011 / 1323203) offered in the second semester of the Diploma in Engineering (Information & Communication Technology) programme under GTU.

The book follows the official GTU syllabus unit-by-unit. Each unit begins with detailed theoretical explanations, builds towards syntax and concept descriptions, and concludes with fully worked solutions to every previous year question (PYQ) from Summer 2023 through Winter 2025 — six complete examination papers.

How to use this book:

- Read the **theory sections** to build a solid conceptual foundation.
- Study the **syntax boxes** to learn the exact structure of each Python construct.
- Run every **code listing** mentally (or in a Python interpreter) to see how output is produced.
- Solve the **PYQ boxes** independently first, then compare with the provided solution.

Contents

CHAPTER 1

Problem Solving using Flowchart and Algorithm

1.1 Introduction to Problem Solving

Before writing a single line of code, every programmer must solve the problem in their mind. This structured thinking process — breaking a complex task into simple, ordered steps — is the heart of problem solving in computing. Problems that seem complex are invariably composed of simpler sub-problems; good programmers learn to identify these sub-problems and handle them systematically.

In our daily lives, we intuitively solve problems constantly. Whether it is deciding the fastest route to work, planning a monthly budget, or cooking a new recipe, we are continuously processing information and executing a sequence of steps to achieve a desired outcome. In the realm of computer science, however, problem solving must be extraordinarily precise. A computer does not possess common sense, intuition, or the ability to "guess" what you mean; it only does exactly what it is strictly instructed to do. Therefore, learning to program is less about memorizing language syntax and much more about developing a structured, analytical mindset.

Definition: Problem Solving

Problem solving in computing is the formal process of defining a problem clearly, analysing its core requirements, designing a step-by-step logical solution, and subsequently implementing that solution in a specific programming language.

To solve any problem computationally, programmers generally follow a standardized lifecycle comprising the following critical stages:

1. **Problem Analysis (Requirement Gathering):** This is the foundational step where the programmer must deeply understand the problem statement. What are the provided inputs? What is the expected exact output? Are there any specific constraints, such as time limits, memory restrictions, or specific data formats? For instance, if the problem is to sort a massive list of numbers, one must ask: Are the numbers integers or decimals? Can there be duplicate values? Are there millions of numbers or just ten? Clarifying these details early prevents catastrophic failures later.

2. **Algorithm Design:** Once the problem is fully understood, the next step is to formulate a clear, unambiguous sequence of steps that will transform the given input into the correct output. This step is entirely independent of any programming language. It is essentially the logical blueprint of the solution. If the algorithm is flawed, the resulting program will always be flawed, no matter how beautifully it is coded.

3. **Flowchart Representation:** Often, visualising the algorithm helps in identifying logical gaps or infinite loops. A flowchart is drawn as a graphical representation of the algorithm, using standardized geometric symbols to indicate different types of operations (like input/output, processing, and decision-making branches). It helps in verifying the flow of control at a glance.

4. **Coding (Implementation):** Only after the algorithm and flowchart have been verified does the actual programming begin. The logical steps are systematically translated into the syntax of a specific programming language, such as Python, Java, C++, or JavaScript. Because the core logic is already solved and documented, this step is merely about adhering to the syntax and grammatical rules of the chosen language.

5. **Testing and Debugging:** Once the code is written, it must be executed with various sets of sample data (test cases), including typical cases, extreme edge cases, and invalid inputs. If the program yields incorrect results or crashes, the programmer must debug it—tracing back through the code to find the logical or syntactical errors, fixing them, and testing again until the software is robust.

6. **Documentation and Maintenance:** Even after the program runs successfully and is deployed, the job is not completely finished. The code must be well-documented so that other developers (or the programmer themselves in the future) can understand how and why it was written. Maintenance involves updating the software continuously as new business requirements arise, APIs change, or hidden bugs are discovered in production.

1.2 Algorithm

An algorithm is the absolute core component of computational problem solving. While computer hardware provides the physical power to process data, it is the algorithm that provides the intelligence and instructions on how to process it.

Definition: Algorithm

An **algorithm** is a finite, ordered sequence of well-defined, unambiguous instructions that solves a problem or accomplishes a specific task.

The word “algorithm” has a rich and fascinating history. It is derived from the name of the 9th-century Persian mathematician, geographer, and astronomer Muhammad ibn Musa al-Khwarizmi. His pioneering work in mathematics, which was later translated into Latin as *Algoritmi de numero Indorum*, introduced the decimal positional number system

and fundamental algebra to the Western world. Over centuries, "Algoritmi" evolved into the modern English word "algorithm".

To understand an algorithm clearly, consider the real-world analogy of a recipe in a cookbook. If you want to bake a chocolate cake, you need a list of raw ingredients (Inputs) and a step-by-step set of instructions (Process) to produce the baked cake (Output). If the recipe says "bake until it looks done", that is highly ambiguous—an experienced human might understand it based on intuition, but a computer would not. An algorithmic instruction must instead say, "bake at 180 degrees Celsius for precisely 30 minutes, then turn off the heat".

1.2.1 Characteristics of a Good Algorithm

Not every sequence of instructions qualifies as a valid, useful algorithm. To be considered computationally correct and robust, a well-designed algorithm must strictly satisfy the following five fundamental properties, famously formalized by the prominent computer scientist Donald Knuth in his seminal work *The Art of Computer Programming*:

1. **Finiteness:** The algorithm must always terminate after a finite number of steps. An algorithm cannot run indefinitely under any circumstances. If it enters an infinite loop (e.g., waiting for a condition that can never become true), it fails to be a valid algorithm. For example, a set of instructions to "keep counting upwards forever" violates finiteness.

2. **Definiteness (Unambiguity):** Every single step of the algorithm must be precisely and clearly defined. There should be absolutely no room for multiple interpretations or guesswork. Words like "few", "some", "maybe", or "approximately" cannot be used. For instance, "add a little bit of salt" is ambiguous and invalid. "Add exactly 5 grams of salt" is definite.

3. **Input:** An algorithm typically takes zero or more inputs—these are the discrete values fed into it from the outside world before or during its execution. "Zero inputs" means that the algorithm might operate entirely on pre-defined constant values embedded within it (for example, an algorithm designed solely to print the first 100 prime numbers doesn't need external user input to run).

4. **Output:** An algorithm must produce one or more outputs. These are the finalized results produced as a direct consequence of executing the instructions. An algorithm that processes massive amounts of data but never actually outputs or saves any result is effectively useless, akin to a calculator that computes a complex sum but refuses to display the answer on the screen.

5. **Effectiveness:** Every step must be sufficiently basic, simple, and feasible so that it can, in principle, be executed mechanically with just a pencil and paper by a human in a finite amount of time. The operations must be realistically computable. An instruction like "Predict tomorrow's exact stock market prices" is not an effective algorithmic step because it lacks a definitive, guaranteed mathematical process.

1.2.2 Advantages and Limitations of Using Algorithms

Using algorithms provides immense benefits to software engineering, though there are also practical limitations to consider during the design phase.

Advantages:

- **Promotes Clear Thinking:** Writing an algorithm forces the programmer to think systematically and critically about the solution before getting bogged down by the strict syntax rules of a programming language. It perfectly separates the logic from the grammar.
- **Language Independence:** An algorithm is a universal, language-agnostic blueprint. Once an algorithm is elegantly designed, it can be seamlessly implemented in Python, C, Java, Ruby, or any other programming language. The core mathematical logic remains identical.
- **Simplifies Debugging:** It is significantly easier and faster to trace logical flaws and perform "dry runs" on paper while reviewing an algorithm than it is to dig through hundreds or thousands of lines of complex source code.
- **Enhances Reusability:** Well-established classical algorithms (such as Bubble Sort, Merge Sort for sorting, or Binary Search for searching) can be packaged as libraries and reused across countless different software projects globally without needing to reinvent the wheel.
- **Facilitates Communication:** Algorithms serve as an excellent standardized communication tool among software development teams. A senior system architect can design the overarching algorithm, and junior developers can implement it in various languages without costly misunderstandings.
- **Basis for Complexity Analysis:** An algorithm allows computer scientists to mathematically analyze its fundamental efficiency. They can formally calculate its Time Complexity (how execution time scales exponentially or linearly with input size) and Space Complexity (how memory usage scales) long before any actual code is written and executed.

Limitations:

- **Time-Consuming Setup:** Writing granular, detailed algorithms for very large, interconnected enterprise software applications can be extremely tedious and time-consuming, sometimes delaying the start of coding.
- **Difficult to Express Complex Branching:** Algorithms heavily reliant on complex conditional nested statements and multiple deep loops can become messy and difficult to read in plain text. (This is exactly where Flowcharts often do a much better job visually).
- **Lack of Standardization:** Unlike strict programming compilers, there is no universal program to verify an algorithm. Different developers might write pseudocode differently, occasionally leading to slight misinterpretations if not careful.

1.2.3 Basic Control Structures in Algorithms

Every algorithm in existence, no matter how profoundly complex, can be constructed using a combination of just three fundamental logical control structures:

1. **Sequence (Sequential Logic):** This is the simplest and default structure. Instructions are executed linearly, one strictly after the other, in the exact top-to-bottom order they appear. There is no branching, decision-making, or skipping of steps.

2. **Selection (Conditional Logic):** This structure allows the algorithm to make intelligent decisions. Based on whether a certain boolean condition evaluates to true or false, the algorithm will branch off to execute one set of instructions and bypass another. This is typically represented universally using `IF...THEN...ELSE` constructs.

3. **Iteration (Looping Logic):** Often, a specific block of instructions needs to be executed repeatedly until a certain condition is finally met or for a fixed number of times. This is iteration. It allows algorithms to be concise while handling massive arrays of data efficiently. It is standardly represented using `WHILE...DO`, `FOR`, or `REPEAT...UNTIL` constructs.

1.2.4 Writing an Algorithm: Conventions and Pseudocode

When writing an algorithm, programmers typically use a notation that sits somewhere comfortably between natural human language (like plain English) and formal, strict programming languages. This highly readable notation is commonly called **Pseudocode**. While there is no rigid compiler syntax for pseudocode, adhering to common industry conventions makes the algorithm universally readable by any programmer.

Standard Conventions:

- **Numbering:** Each instruction step should be numbered sequentially (e.g., Step 1, Step 2, Step 3) for easy reference during discussions or dry runs.

- **Start and Stop:** The algorithm must explicitly indicate where its execution begins using `START` (or `BEGIN`) and where it permanently terminates using `STOP` (or `END`).

- **Input Operations:** Use explicit uppercase keywords like `READ`, `INPUT`, or `GET` when actively receiving data from the user, a sensor, or a file. (e.g., `READ radius`).

- **Processing / Assignment:** Use clear keywords like `SET`, `ASSIGN`, `COMPUTE`, or `CALCULATE` to perform mathematical operations or assign state values to variables. (e.g., `SET Area = 3.14159 * r * r`).

- **Output Operations:** Use keywords like `PRINT`, `DISPLAY`, `OUTPUT`, or `WRITE` to manifest results to the user on a screen or printer. (e.g., `PRINT Area`).

- **Conditional Statements:** Use `IF`, `THEN`, `ELSE IF`, `ELSE`, and `ENDIF` to clearly delineate decision branches. Proper indentation is highly recommended and practically required here to visually show which steps belong inside which conditional block.

- **Loops:** Use `WHILE`, `FOR`, `REPEAT`, `UNTIL`, and `ENDWHILE` to represent looping mechanisms. Again, clear indentation is crucial to show the repeatable body of the loop.

1.3 Example Algorithms

To build a strong, practical foundation, let us examine progressively complex examples spanning all three logical structures: sequential, selection, and iteration.

1.3.1 Sequential Logic Examples

Algorithm to Find the Sum of Two Numbers

This is a straightforward sequential algorithm with no decisions or loops. It simply flows top to bottom.

Step 1: START

Step 2: READ two numbers into variables A and B.

Step 3: SET $Sum = A + B$

Step 4: PRINT Sum

Step 5: STOP

Algorithm to Find the Area of a Circle

This algorithm requires the usage of a constant value (π).

Step 1: START

Step 2: READ the radius of the circle into variable r.

Step 3: SET $Area = 3.14159 \times r \times r$

Step 4: PRINT "The Area of the circle is: ", $Area$

Step 5: STOP

Algorithm to Swap the Values of Two Variables

Swapping values is a fundamental concept in computing (especially in sorting algorithms). We usually need a temporary third variable to hold one value temporarily so it is not overwritten and lost.

Step 1: START

Step 2: READ variables X and Y.

Step 3: SET $Temp = X$

Step 4: SET $X = Y$

Step 5: SET $Y = Temp$

Step 6: PRINT "After swapping: X = ", X, " Y = ", Y

Step 7: STOP

1.3.2 Selection Logic Examples

Algorithm to Find the Larger of Two Numbers

Here, the algorithm must make an intelligent choice between different output paths based on a mathematical comparison.

Step 1: START

Step 2: READ two numbers A and B.

Step 3: IF $A > B$ THEN
PRINT "A is greater."

Step 4: ELSE IF $B > A$ THEN
PRINT "B is greater."

Step 5: ELSE
PRINT "Both numbers are exactly equal."

Step 6: ENDIF

Step 7: STOP

Algorithm to Check if a Number is Even or Odd

We can easily determine this by checking the remainder when the number is divided by 2. We use the modulo mathematical operator (MOD) to extract the remainder.

Step 1: START

Step 2: READ a number into variable N.

Step 3: COMPUTE $Remainder = N$ MOD 2

Step 4: IF $Remainder == 0$ THEN
PRINT N, " is an Even number."

Step 5: ELSE
PRINT N, " is an Odd number."

Step 6: ENDIF

Step 7: STOP

1.3.3 Iteration Logic Examples

Algorithm to Print Numbers from 1 to N

This algorithm utilizes a loop that runs multiple times continuously until a specific boundary condition fails.

Step 1: START

Step 2: READ the upper limit value N.

Step 3: SET $i = 1$

Step 4: WHILE $i \leq N$ DO
PRINT i
SET $i = i + 1$ *(Increment the counter)*

Step 5: ENDWHILE

Step 6: STOP

Algorithm to Find the Factorial of a Number

The factorial of a non-negative integer N (written mathematically as $N!$) is the product of all positive integers less than or equal to N. For example, $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$.

Step 1: START

Step 2: READ number N.

Step 3: IF $N < 0$ THEN
PRINT "Error: Factorial does not exist for negative numbers."

Step 4: ELSE
SET $Fact = 1$
SET $i = 1$
WHILE $i \leq N$ DO
SET $Fact = Fact \times i$
SET $i = i + 1$
ENDWHILE
PRINT "The Factorial is: ", $Fact$

Step 5: ENDIF

Step 6: STOP

Algorithm to Check if a Number is Prime

A prime number is a natural number strictly greater than 1 that cannot be formed by multiplying two smaller natural numbers (i.e., it is only divisible by 1 and itself). We can check this by trying to divide it by every number from 2 up to $N - 1$.

Step 1: START

Step 2: READ number N.

Step 3: IF $N \leq 1$ THEN
PRINT "Not a Prime number."
STOP

Step 4: ENDIF

Step 5: SET $i = 2$

Step 6: SET $IsPrime = TRUE$ *(Assume it is prime initially)*

Step 7: WHILE $i < N$ DO
IF (N MOD i) == 0 THEN
SET $IsPrime = FALSE$ *(We found a divisor, so it's not prime)*
BREAK loop *(Exit the while loop early for efficiency)*
ENDIF
SET $i = i + 1$

Step 8: ENDWHILE

Step 9: IF $IsPrime == TRUE$ THEN
PRINT N, " is a Prime number."

Step 10: ELSE
PRINT N, " is not a Prime number."

Step 11: ENDIF

Step 12: STOP

1.4 University Previous Year Questions

PYQ: Define algorithm. What are the advantages of an Algorithm? (Su'23)

Algorithm: An algorithm is a finite, ordered sequence of well-defined, unambiguous instructions that solves a computational problem or achieves a specific task. It inherently possesses five key characteristics: **Finiteness** (it must always eventually terminate), **Definiteness** (there is absolute clarity with no ambiguity in any step), **Input** (it securely accepts zero or more external inputs), **Output** (it reliably produces one or more correct outputs), and **Effectiveness** (each step is sufficiently basic and practically feasible to execute).
Advantages:

1. **Clear logical representation:** It forces systematic, structured thinking about the solution architecture before actual coding begins.
2. **Language independent framework:** The designed logic can be translated seamlessly into any modern programming language like Python, Java, or C++.
3. **Easy to debug and trace:** Deep logical errors can be identified and fixed on paper far more easily than debugging within thousands of lines of source code.
4. **Highly Reusable:** Classical, mathematically proven algorithms (like those for sorting arrays or searching databases) can be reused across multiple different enterprise projects.

5. **Efficient design analysis:** It allows developers to mathematically and thoroughly analyze both time and space complexity prior to any costly software implementation.

6. **Seamless Team communication:** It powerfully serves as a universal, standardized design language between developers, architects, and stakeholders working in different technological stacks.

PYQ: Define Algorithm. Write algorithm to find area of a circle. (W'25)

Algorithm: A step-by-step, finite, and strictly unambiguous set of logical instructions meticulously designed to solve a given computational problem efficiently.
Algorithm to find Area of a Circle:

Step 1: START

Step 2: READ radius r securely from the user input.

Step 3: SET $Area = 3.14159 \times r \times r$

Step 4: DISPLAY "Area of circle = ", $Area$

Step 5: STOP

PYQ: Prepare algorithms for: (1) To find maximum of two given numbers (2) To find sum of two given numbers. (Su'25)

Algorithm 1: Maximum of Two Numbers

Step 1: START

Step 2: READ numbers A and B

Step 3: IF $A > B$ THEN
PRINT "Maximum is ", A

Step 4: ELSE IF $B > A$ THEN
PRINT "Maximum is ", B

Step 5: ELSE
PRINT "Both numbers are identical and equal."

Step 6: ENDIF

Step 7: STOP

Algorithm 2: Sum of Two Numbers

Step 1: START

Step 2: READ numbers A and B

Step 3: SET $Sum = A + B$

Step 4: PRINT "Sum = ", Sum

Step 5: STOP

1.5 Flowcharts

Definition: Flowchart

A **flowchart** is a graphical (diagrammatic) representation of an algorithm or process, using standardised geometric symbols connected by arrows. Each symbol represents a specific type of operation. Flowcharts make the logic of an algorithm easy to visualise, understand, and communicate — even to non-programmers.

1.5.1 Standard Flowchart Symbols

The following symbols are standardised by ANSI/ISO and are universally recognised:

Symbol Name	Shape	Purpose
Terminal	Rounded rectangle (oval)	Marks the **START** or **STOP** of the flowchart. Every flowchart must have exactly one Start and at least one Stop.
Process	Rectangle	Represents an action, computation, or assignment (e.g., $Sum = A + B$).
Decision	Diamond	Represents a conditional test (YES/NO or TRUE/FALSE question). Has two or more exit paths.
Input / Output	Parallelogram	Represents reading data from the user (INPUT) or displaying data to the screen (OUTPUT).
Flow Arrow	Arrow / Line	Connects symbols and shows the direction of flow (sequence of execution).
Connector	Small circle	Used to connect different parts of a large flowchart without drawing long crossing lines. Often labelled with a letter.
Predefined Process	Rectangle with double side bars	Represents a subroutine or predefined function (a named block of steps defined elsewhere).

Table 1.1: Standard Flowchart Symbols and Their Meanings

1.5.2 Rules for Drawing Flowcharts

1. Every flowchart must have a unique **START** terminal and at least one **STOP** terminal.
2. The flow of control must always be indicated by **arrows** (arrowheads on every connecting line).

3. The general direction of flow should be **top to bottom** and **left to right** for readability.
4. Each **Decision** diamond must have exactly two (or more) labelled exit paths — typically "Yes/No" or "True/False".
5. Flowcharts must be **finite** — the flow must eventually reach a STOP symbol; there must be no infinite loops without a termination condition.
6. Only one flow line should enter a symbol (connector can be an exception).
7. Crossing flow lines should be avoided; use connectors to bridge different sections.
8. All symbols should be properly sized and neatly drawn; standard shapes must be maintained.

1.5.3 Advantages of Flowcharts

- **Visual clarity:** The graphical format makes the logic of the program immediately apparent to anyone who reads it.
- **Easy communication:** Flowcharts can be understood by both technical programmers and non-technical stakeholders.
- **Error detection:** Logical errors (such as loops that never terminate or missing branches) are much easier to spot in a flowchart than in code.
- **Documentation:** Flowcharts form an essential part of program documentation, helping future programmers understand the original design.
- **Problem-solving aid:** Drawing a flowchart forces the programmer to think through every possible path (normal, error, edge case) before coding.

1.5.4 Limitations of Flowcharts

- For large programs, flowcharts can become extremely complex and difficult to modify.
- Drawing flowcharts is time-consuming when the program is large.
- There is no standardised way to represent certain modern programming concepts (object-oriented structures, exception handling).
- Flowcharts are not directly executable — they must be manually translated into code.

1.5.5 Types of Flowchart Structures

Every algorithm can be expressed as a combination of three fundamental control structures (Bohm-Jacopini theorem):

1. **Sequence:** Instructions executed one after another in order, top to bottom. Every step is always executed exactly once.

2. **Selection (Decision):** A condition is tested; different paths are followed depending on whether the condition is True or False. This is represented by the diamond symbol with two exit paths.

3. **Repetition (Loop):** A block of instructions is executed repeatedly as long as (or until) a condition is true. The diamond checks the condition; a loop-back arrow returns to the start of the block.

1.5.6 Example Flowcharts

Flowchart: Find Simple Interest

Formula: $SI = \dfrac{P \times R \times T}{100}$
Flowchart description (vertical sequence):

$$\text{START} \rightarrow \text{READ } P, R, T \rightarrow \text{SET } SI = \frac{P \times R \times T}{100} \rightarrow \text{PRINT } SI \rightarrow \text{STOP}$$

This is a pure **sequence** flowchart with no decisions or loops.

Flowchart: Find Whether a Number is Even or Odd

Logic: If a number N is divisible by 2 (i.e., $N \bmod 2 = 0$), it is even; otherwise it is odd.
Flow (using selection):

1. START
2. INPUT N
3. DECISION: Is $N \bmod 2 = 0$?
   - **YES** $\rightarrow$ PRINT "N is Even" $\rightarrow$ STOP
   - **NO** $\rightarrow$ PRINT "N is Odd" $\rightarrow$ STOP

Flowchart: Sum of First Twenty Even Natural Numbers

Even positive integers: 2, 4, 6, ..., 40 (the first 20 even natural numbers).
Flow (using repetition loop):

1. START
2. SET $Sum = 0$, $i = 2$, $count = 0$
3. DECISION: Is $count < 20$?
   - **YES** $\rightarrow$ SET $Sum = Sum + i$; SET $i = i + 2$; SET $count = count + 1$; GO TO step 3
   - **NO** $\rightarrow$ PRINT Sum $\rightarrow$ STOP

PYQ: Define flowchart and list any four symbols with their purpose. (W'24, Su'25)

Flowchart: A diagram that uses standardised geometric shapes connected by arrows to represent the step-by-step logic of an algorithm. It makes the flow of a program visually understandable.
Four Standard Symbols:

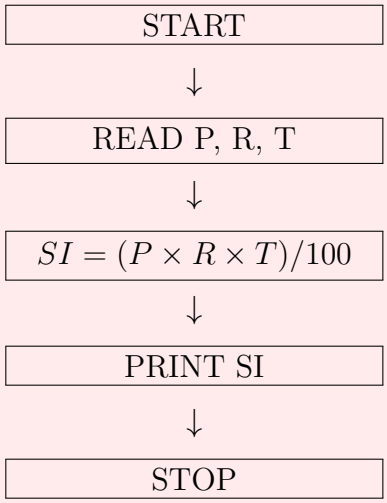
Symbol	Purpose
Terminal (Oval)	Marks the START or STOP of the flowchart.
Process (Rectangle)	Represents a computation or assignment operation, e.g., $Sum = A + B$.
Decision (Diamond)	Tests a condition (Yes/No). Directs flow along two paths depending on the result.
Input/Output (Parallelogram)	Represents reading input from the user or displaying output on screen.

PYQ: State rules for problem solving using flowchart. Design flowchart to find simple interest. (Su'23, W'25)

Rules for flowcharts:

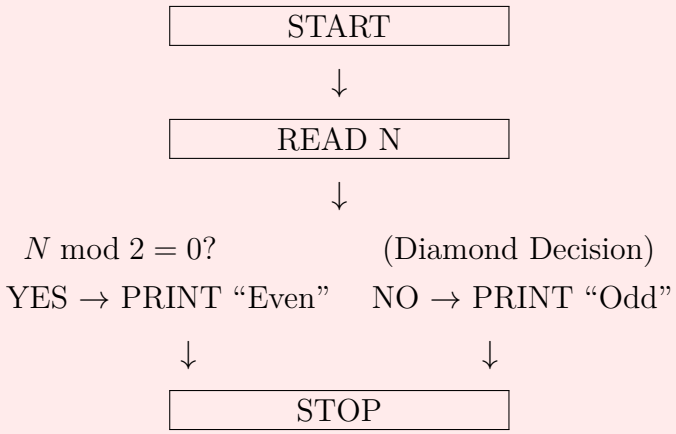
1. Every flowchart must begin with a START and end with a STOP terminal.
2. Arrows must always show the direction of flow.
3. Decision diamonds must have clearly labelled exit paths (Yes/No).
4. Flow should go top-to-bottom; avoid crossing arrows.
5. Flowcharts must be finite (must always end).

Flowchart for Simple Interest:



PYQ: Draw flowchart to find whether the entered number is even or odd. (Su'24)

Flowchart logic: Read a number N. If $N \bmod 2 == 0$, print "Even"; otherwise print "Odd".



PYQ: Design flowchart to calculate sum of first twenty even natural numbers. (W'24)

The first 20 even natural numbers are 2, 4, 6, ..., 40. We accumulate their sum using a counter-controlled loop.
Flowchart description:

1. START
2. SET $Sum = 0$, $count = 0$, $n = 2$
3. DECISION: Is $count < 20$?
   - YES: $Sum = Sum + n$; $n = n + 2$; $count = count + 1$; **go back to step 3**

- NO: Continue to step 4

4. PRINT "Sum of first 20 even natural numbers = ", Sum
5. STOP

   The expected result: $Sum = 2 + 4 + 6 + \cdots + 40 = 420$.

PYQ: Write the difference between flowchart and algorithm. (Su'23 OR)

Parameter	Algorithm	Flowchart
Format	Written in structured English or pseudocode.	Drawn using geometric symbols and arrows.
Nature	Textual, sequential listing of steps.	Visual, graphical representation.
Ease of understanding	Requires reading and comprehension of logic.	Easier to follow visually; good for non-programmers.
Modification	Easy to modify — just rewrite the steps.	Difficult to modify for large or complex programs.
Use of symbols	No special symbols required.	Uses standardised symbols (oval, rectangle, diamond, parallelogram).
Detail level	Can describe steps in any level of detail.	May become unwieldy for very detailed or long processes.

1.6 Pseudocode

Definition: Pseudocode

Pseudocode is an informal, high-level description of a computer program or algorithm that uses the structural conventions of programming languages but is written in natural English (or a mix of English and simple programming constructs). It is not executable by a computer; it is purely a design and communication tool for the programmer.
The word "pseudo" means "false" or "imitation" — pseudocode imitates a programming language without being one.

1.6.1 Features of Pseudocode

- Language-independent: Not tied to any programming language.
- Human-readable: Easier and faster to write than a formal flowchart.
- Structured: Uses indentation and keywords (IF, WHILE, FOR) to show structure.
- Not executable: Cannot be run directly; must be translated into actual code.
- Flexible: No rigid syntax rules — as long as it is clear and unambiguous.

1.6.2 Pseudocode Keywords and Conventions

Keyword	Meaning
`BEGIN / END`	Start and end of the pseudocode block.
`READ / INPUT`	Accept input from the user.
`PRINT / OUTPUT / DISPLAY`	Show output to the user.
`SET / LET`	Assign a value to a variable.
`IF ... THEN ... ELSE ... ENDIF`	Conditional branching.
`WHILE ... DO ... ENDWHILE`	Loop that checks condition before each iteration.
`FOR i = a TO b DO ... ENDFOR`	Counter-controlled loop.
`REPEAT ... UNTIL condition`	Loop that checks condition after each iteration.

PYQ: Define pseudocode. Write pseudocode to find the smallest of two numbers. (Su'23)

Pseudocode: An informal, English-like description of a computer program that uses programming constructs (IF, WHILE, etc.) but is not executable code. It bridges the gap between an algorithm (plain English) and actual code.
Pseudocode to find smallest of two numbers:

```
BEGIN
    INPUT A, B
    IF A < B THEN
        PRINT "Smallest is", A
    ELSE
        PRINT "Smallest is", B
    ENDIF
END
```

PYQ: Write pseudocode to check if the entered number is positive or negative. (W'23)

```
BEGIN
    INPUT N
    IF N > 0 THEN
        PRINT "Number is Positive"
    ELSE IF N < 0 THEN
        PRINT "Number is Negative"
    ELSE
        PRINT "Number is Zero"
    ENDIF
END
```

PYQ: Define pseudocode. Write pseudocode to find the largest of three numbers x, y, z. (Su'24)

Pseudocode: A structured, plain-language representation of an algorithm — language-independent and not directly executable, but close enough to code that translating it to Python (or any language) is straightforward.

Pseudocode to find largest of x, y, z:

```
BEGIN
    INPUT x, y, z
    SET largest = x
    IF y > largest THEN
        SET largest = y
    ENDIF
    IF z > largest THEN
        SET largest = z
    ENDIF
    PRINT "Largest number is", largest
END
```

PYQ: Create algorithm to print odd numbers between 1 to 20. (W'24 OR)

Algorithm:

Step 1: START

Step 2: SET $i = 1$

Step 3: WHILE $i \leq 20$ DO

- IF $i \bmod 2 \neq 0$ THEN PRINT i ENDIF
- SET $i = i + 1$

Step 4: ENDWHILE

Step 5: STOP

Odd numbers from 1 to 20: 1, 3, 5, 7, 9, 11, 13, 15, 17, 19.

CHAPTER 2

Introduction to Python

2.1 What is Python?

Python is a high-level, general-purpose, interpreted programming language created by **Guido van Rossum**. Its design began in the late 1980s as a successor to the ABC programming language, and it was first released in **1991**. Contrary to popular belief, Van Rossum named the language not after the non-venomous snake, but after the British comedy television series *"Monty Python's Flying Circus,"* of which he was a huge fan. Over the decades, Python has evolved significantly. Python 2.0 was released in 2000, bringing features like list comprehensions and a garbage collection system, while Python 3.0, released in 2008, was a major revision that modernised the language but was not strictly backwards-compatible.

Today, Python consistently ranks as one of the world's most popular programming languages in multiple indices, such as the TIOBE index and Stack Overflow Developer Surveys. Its popularity is driven by its ease of use, an enormous ecosystem of libraries, and its dominance in trending fields like artificial intelligence and data science.

Python emphatically prioritises code readability. Its syntax is meticulously designed to be clean, expressive, and incredibly close to plain English, which reduces the cost of program maintenance and development. The core Python philosophy can be experienced directly in the interpreter by typing `import this`, which prints **The Zen of Python** by Tim Peters. Some of its most famous aphorisms include:

- *Beautiful is better than ugly.*
- *Explicit is better than implicit.*
- *Simple is better than complex.*
- *Readability counts.*
- *There should be one — and preferably only one — obvious way to do it.*

Definition: Python

Python is a high-level, interpreted, object-oriented, dynamically-typed, general-purpose programming language. It was designed with the philosophy that code readability is paramount and that the programmer's time and developer experience are far more valuable than the computer's processing time.
Key characteristics that define Python include:

- **High-level:** Python abstracts away the intricate, low-level details of the computer system, such as memory management and hardware architecture. Programmers write code that is closer to human cognition.
- **Interpreted:** Python source code is executed line-by-line by the Python interpreter at runtime, which bypasses the need for a separate, manual compilation step into machine code.
- **Dynamically typed:** The data type of a variable is inferred automatically by the interpreter at runtime. There is no need to declare variable types explicitly in the code.
- **Object-oriented:** Everything in Python is an object, meaning integers, strings, lists, functions, and even modules have properties and methods.
- **Open source:** Python is entirely free to use, modify, and distribute, even for commercial purposes. Its source code is collaboratively maintained and publicly available at `python.org`.

2.2 Features of Python

Python's immense success is attributed to its rich set of features that cater to both beginners and seasoned professionals. Below is a detailed look at the core features of Python:

1. **Simple and Easy to Learn:** Python boasts a simple, minimalist syntax. Reading a well-written Python program feels almost like reading English. A "Hello World" program requires just one line: `print("Hello, World!")`. This low barrier to entry allows absolute beginners to become productive incredibly fast.
2. **Interpreted Language:** Python programs are executed directly by the Python interpreter. Behind the scenes, Python source code is automatically compiled into an intermediate *bytecode*, which is then executed by the Python Virtual Machine (PVM). Because there is no manual compilation step (like in C or Java), code changes can be tested immediately, heavily accelerating the debugging and development cycle.
3. **Platform Independent (Cross-platform):** Python is fully cross-platform. You can write a Python script on a Windows machine and run the exact same script on macOS, Linux, or even a Raspberry Pi without making any modifications. The principle of "Write once, run anywhere" strictly applies to Python.
4. **Free and Open Source:** Python is distributed under the OSI-approved Python Software Foundation License. This means it is free to download and use in any application. Furthermore, its open-source nature means that a vast, global community of developers continuously contributes to improving the language and its libraries.
5. **High-level Language:** When programming in Python, developers never have to worry about low-level system details. The language automatically handles memory

allocation, pointer arithmetic, and garbage collection (reclaiming memory that is no longer in use). This allows programmers to focus solely on solving the problem at hand.

6. **Multi-paradigm (OOP, Procedural, Functional):** Python does not force a programmer into a specific way of thinking. It fully supports Object-Oriented Programming (encapsulation, inheritance, polymorphism) for building complex, reusable architectures. It also supports procedural programming (writing code as a sequence of steps) and functional programming paradigms.

7. **Dynamically Typed:** In statically typed languages like C++ or Java, variables must be declared with their type (e.g., `int x = 5;`). Python uses dynamic typing, meaning types are determined at runtime. You can assign an integer to a variable, and later assign a string to that same variable:

```
x = 10          # x is an integer
x = "Hello"     # x is now a string
```

Listing 2.1: Dynamic Typing Example

8. **Extensive Standard Library:** Python follows a "batteries included" philosophy. Its massive standard library ships with the language, providing pre-written modules and functions for a wide variety of tasks. Whether you need to process text using regular expressions, read and write files, handle internet protocols (HTTP, FTP), work with dates, or parse JSON, the standard library has a module ready to use.

9. **Extensible and Embeddable:** Python plays well with other languages. If you have a highly critical piece of code that requires maximum execution speed, you can write that specific module in C or C++ and call it from your Python script (extensibility). Conversely, you can embed Python into a C/C++ application to provide a scripting interface for the application's users.

10. **Robust Exception Handling:** Python provides a structured, elegant way to handle runtime errors using `try`, `except`, `finally`, and `raise` blocks. This prevents programs from crashing abruptly and allows developers to write robust, fault-tolerant software.

PYQ: List the features of Python. (Su'25, W'25)

Key Features of Python:

1. **Simple and Easy to Learn** — clean, readable syntax close to English.
2. **Interpreted** — code is executed line-by-line without a manual compilation step.
3. **Platform Independent** — runs on Windows, Linux, macOS without changes.
4. **Free and Open Source** — freely available at python.org; no licence fee.
5. **High-level Language** — automatic memory management and garbage collection.

6. **Multi-paradigm** — supports object-oriented, procedural, and functional programming.
7. **Dynamically Typed** — no need to declare variable types; determined at runtime.
8. **Large Standard Library** — "batteries included"; extensive built-in modules.
9. **Extensible** — can integrate C/C++ code for performance-critical tasks.
10. **Embeddable** — can be embedded in other applications as a scripting engine.

2.3 Applications of Python

Because Python is a general-purpose language, it is not restricted to any single domain. Its versatility, combined with powerful third-party libraries, makes it a dominant force across numerous tech industries. Top-tier companies like Google, Netflix, Spotify, Instagram, and NASA rely heavily on Python in their technology stacks.

- **Web Development:** Python is widely used to build the backend logic of web applications. Frameworks like **Django** and **Flask** allow developers to create secure, scalable, and complex web applications rapidly. Recently, **FastAPI** has also gained immense popularity for building high-performance web APIs.
- **Data Science and Data Analytics:** Python is the undisputed king of the data world. Libraries such as **Pandas** (for data manipulation), **NumPy** (for numerical data arrays), and **Matplotlib** / **Seaborn** (for data visualisation) allow analysts to extract, clean, and draw insights from massive datasets.
- **Machine Learning and Artificial Intelligence:** The AI revolution is largely powered by Python. The most popular frameworks for building neural networks, predictive models, and generative AI are Python-based, including **TensorFlow**, **PyTorch**, and **scikit-learn**. Python's simple syntax allows researchers to focus on the mathematics of AI rather than the complexities of programming.
- **Scientific and Numeric Computing:** In research and academia (physics, chemistry, biology, astronomy), Python has largely replaced older languages like Fortran and MATLAB. Libraries like **SciPy** and **SymPy** provide advanced mathematical functions, optimisation algorithms, and symbolic computation.
- **Automation and Scripting:** Python excels at automating boring, repetitive tasks. System administrators and DevOps engineers use Python scripts to automate server provisioning, parse logs, rename batches of files, and interact with operating system APIs. Tools like **Selenium** and **BeautifulSoup** are heavily used for automating web browsers and scraping data from websites.
- **Software Testing:** Python is highly regarded in the software testing industry. Frameworks like **PyTest** and **Robot Framework** are extensively used to write automated test cases for both software and hardware.

- **Desktop GUI Applications:** Though web apps are more common today, Python still provides robust tools for building cross-platform desktop applications with Graphical User Interfaces (GUIs). **Tkinter** is built into the standard library, while **PyQt** and **wxPython** offer more advanced UI components.
- **Game Development:** While not typically used for AAA 3D titles, Python is excellent for building 2D games and prototypes using the **Pygame** library. Additionally, Python scripts are often embedded within major game engines to control game logic and artificial intelligence.
- **Database Programming:** Python provides standard interfaces (DB-API) for connecting to virtually any database engine, including relational databases like MySQL, PostgreSQL, and SQLite, as well as NoSQL databases like MongoDB and Redis.
- **Embedded Systems and IoT (Internet of Things):** Optimised versions of Python, such as **MicroPython** and **CircuitPython**, are designed to run on microcontrollers with extremely limited resources (e.g., Raspberry Pi Pico, ESP32), allowing hardware enthusiasts to write embedded software in Python instead of C.

PYQ: List the applications of Python. (Su'25 OR, W'25 OR)

Applications of Python:

1. **Web Development** — Django, Flask frameworks power web applications.
2. **Data Science & Analytics** — Pandas, NumPy, Matplotlib for data analysis and visualisation.
3. **Machine Learning & AI** — TensorFlow, PyTorch, scikit-learn for model development.
4. **Scientific Computing** — SciPy, SymPy used in research and academia.
5. **Automation and Scripting** — file operations, web scraping, test automation.
6. **Software Testing** — PyTest used for automated QA testing.
7. **Game Development** — Pygame for 2D games and prototyping.
8. **Desktop GUI** — Tkinter, PyQt for graphical desktop applications.
9. **Database Programming** — SQLite, MySQL, MongoDB interfaces.
10. **Embedded / IoT** — MicroPython on Raspberry Pi and microcontrollers.

2.4 Execution Model of Python

Understanding how Python runs code helps clarify why it is called an "interpreted" language. When you execute a Python script (e.g., `program.py`), a multi-step process occurs automatically in the background:

1. **Translation to Bytecode:** First, the Python compiler reads the source code and translates it into an intermediate, lower-level format known as **bytecode** (saved in `.pyc` files inside a `__pycache__` directory). Bytecode is platform-independent.

2. **Python Virtual Machine (PVM):** This bytecode is not understandable by the CPU hardware directly. Instead, it is sent to the Python Virtual Machine (PVM). The PVM acts as a runtime engine that interprets the bytecode instructions one by one and executes them on the host operating system.

Because of this two-step architecture, Python is technically both compiled (to bytecode) and interpreted (by the PVM). However, since the compilation step is entirely hidden from the user, it is broadly categorised as an interpreted language.

2.5 Basic Structure of a Python Program

A Python program is a sequence of **statements**. A statement is an instruction that the Python interpreter can execute (e.g., assigning a value to a variable, calling a function, looping over a list). Unlike languages like C, C++, or Java, Python does not use semicolons (`;`) to terminate statements. Instead, a statement typically ends with a **newline character** (pressing Enter).

Another major distinction is how Python defines code blocks. While other languages use curly braces `\{\}` to group statements together (for example, inside an `if` statement or a function), Python uses **indentation** (whitespace at the beginning of a line). According to the official PEP 8 style guide, the standard indentation is **4 spaces**. Incorrect indentation in Python is not just a formatting issue; it will result in an `IndentationError` and crash the program.

2.5.1 Comments in Python

Comments are human-readable text written inside the code to explain logic, leave notes, or temporarily disable parts of the code. The Python interpreter completely ignores comments.

- **Single-line comments:** Start with a hash/pound symbol (`\#`). Everything after the `\#` on that line is ignored.

- **Multi-line comments:** Python does not have a specific syntax for multi-line comments. Instead, developers often use multi-line string literals enclosed in triple quotes (`"""` or `'''`). If a string is not assigned to a variable, the interpreter ignores it, effectively treating it as a comment.

```
"""
This is a multi-line comment or docstring.
This program demonstrates basic Python structure.
"""

# This is a single-line comment
name = "Alice"           # Variable assignment
age = 20                 # Integer variable
```

```
9
10 # A simple if-statement demonstrating indentation
11 if age >= 18:
12     # Notice the 4 spaces of indentation here!
13     print("Hello,", name)
14     print("You are an adult.")
15 else:
16     print("You are a minor.")
```

Listing 2.2: Basic Structure, Comments, and Indentation

Output

```
Hello, Alice
You are an adult.
```

2.6 Keywords and Identifiers

When writing statements in Python, you construct them using a combination of Keywords (reserved words) and Identifiers (custom names).

2.6.1 Keywords

Definition: Keywords

Keywords (also called *reserved words*) in Python are predefined, special words that have a fixed meaning, syntax, and purpose in the language. Because the interpreter uses them to understand the structure of the program, they cannot be used as variable names, function names, class names, or any other custom identifiers.

As of Python 3.9+, there are **35 keywords**. You can view the full list at any time in the Python interpreter by typing `help("keywords")`. All keywords are written in entirely lowercase letters, with only three exceptions: `True`, `False`, and `None`, which must be Capitalised.

Keyword	Purpose
`if`, `elif`, `else`	Conditional branching (decision making)
`for`, `while`	Iteration (loops)
`break`, `continue`, `pass`	Altering loop control and placeholder statements
`def`	Used to define a custom function
`return`	Used to return a value from a function
`import`, `from`, `as`	Importing external modules and libraries
`class`	Used to define a class (Object-Oriented Programming)
`True`, `False`, `None`	Boolean values and the null object
`and`, `or`, `not`	Logical operators for combining conditions
`in`, `is`	Membership testing and object identity operators
`try`, `except`, `finally`, `raise`	Exception handling blocks
`global`, `nonlocal`	Variable scope declaration
`lambda`	Creating anonymous, single-line functions
`with`	Context managers (ensuring files/resources are closed properly)

Table 2.1: Commonly Used Python Keywords

2.6.2 Identifiers

Definition: Identifiers

An **identifier** is a custom name given by the programmer to identify a variable, function, class, module, or any other entity in a Python program. It is essentially the label you attach to an object in memory.

To ensure the interpreter can distinguish your custom identifiers from operators and numbers, Python enforces a strict set of rules for naming them.

Rules for Defining Python Identifiers

1. Identifiers can contain **alphabetic letters** (A–Z, a–z), **digits** (0–9), and the **underscore** character (_).
2. An identifier **cannot start with a digit**. For example, `employee\_1` is valid, but `1\_employee` will result in a SyntaxError.
3. Identifiers are strictly **case-sensitive**. The variables `age`, `Age`, and `AGE` represent three entirely different memory locations in Python.
4. **Keywords cannot be used** as identifiers. You cannot name a variable `class` or `for`.

5. **No special characters or spaces** are allowed. Characters like @, $, %, -, and whitespace are strictly forbidden. `first-name` is invalid (Python interprets the hyphen as subtraction), whereas `first\_name` is valid.

6. Identifiers can theoretically be of **any length**, but practically they should be kept reasonably short and highly descriptive of the data they hold.

Naming Conventions (PEP 8)

While the rules above are strictly enforced by the interpreter, Python developers also follow stylistic conventions (outlined in PEP 8) to ensure code remains consistent and readable across the global community:

- **Variables and Functions:** Use `snake\_case` (all lowercase, words separated by underscores). Examples: `student\_name`, `calculate\_total\_score()`.
- **Constants:** Use `UPPER\_SNAKE\_CASE` (all uppercase, words separated by underscores). Constants are variables whose values are not meant to change. Examples: `PI`, `MAX\_CONNECTIONS`.
- **Classes:** Use `PascalCase` (or `CamelCase` where the first letter of each word is capitalised, with no underscores). Examples: `StudentRecord`, `DatabaseConnection`.

PYQ: List out rules for defining variables in Python. (W'23 OR)

Rules for Python Identifiers/Variables:

1. Must start with a letter (a–z, A–Z) or an underscore (_); it cannot start with a digit.
2. Can contain only letters, digits (0–9), and underscores.
3. Are case-sensitive: `Total` and `total` represent different variables.
4. Must not be a reserved Python keyword (e.g., `if`, `for`, `while` cannot be used as variable names).
5. No special characters allowed: !, @, #, $, %, -, spaces, etc.
6. Can be of any length (but should be meaningful and readable).

Valid examples: `name`, `student\_1`, `\_total\_marks`, `myVar`
Invalid examples: `1student` (starts with a digit), `for` (is a keyword), `my-var` (hyphen is not allowed), `first name` (space is not allowed)

2.7 Variables and Data Types in Python

2.7.1 Variables and Memory Allocation

A **variable** is a fundamental concept in any programming language. It serves as a named memory location used to store data that can be referenced and manipulated by the program. However, Python handles variables differently compared to statically typed languages like C, C++, or Java.

In Python, a variable is not a container that holds a value, but rather a *label* or a *reference* pointing to an object residing in the computer's memory. When you assign a value to a variable, Python creates an object in memory, determines its type, and then binds the variable name to that object. There is no explicit "declaration" step. The variable is created the moment you first assign a value to it.

Dynamic Typing

Python is a dynamically typed language. This means you do not need to specify the data type of a variable when you create it. The Python interpreter automatically determines the type of the variable from its assigned value at runtime. Furthermore, a variable can be reassigned to an object of a different type later in the program without causing an error.

```
# Variables taking on different types dynamically
data = 100            # data is an integer
print(type(data))     # <class 'int'>

data = "Now I am a string"  # data is reassigned to a string
print(type(data))           # <class 'str'>
```

Listing 2.3: Dynamic typing in Python

Variable Naming Rules

To define valid variable names (identifiers) in Python, you must follow specific rules:

- A variable name must start with a letter (a-z, A-Z) or an underscore (_).
- The rest of the variable name can consist of letters, numbers (0-9), and underscores.
- Variable names are case-sensitive (`age`, `Age`, and `AGE` are three distinct variables).
- You cannot use Python's reserved keywords (e.g., `if`, `for`, `class`, `True`) as variable names.

Naming Conventions: It is considered best practice to use descriptive variable names to make your code more readable. Python programmers generally use *snake_case* (e.g., `student_name`, `total_score`) for variable and function names.

Assignments and Multiple Assignments

Python provides elegant ways to assign values to variables.

```
# Standard assignment
x = 10
y = 3.14
name = "Alice"
flag = True

# Multiple assignment (chaining)
a = b = c = 0    # All three variables point to the same
    integer object 0

# Tuple unpacking / simultaneous assignment
p, q, r = 1, 2, 3  # p=1, q=2, r=3

# Swapping variables elegantly without a temporary variable
p, q = q, p         # Now p=2, q=1
```

Listing 2.4: Various variable assignment techniques

Output

```
10 3.14 Alice True
0 0 0
1 2 3
```

2.7.2 Python Data Types

Data types dictate what kind of operations can be performed on a value and how it is stored in memory. Python provides a rich set of built-in data types.

1. Integer (`int`)

Integers are whole numbers without a fractional or decimal component. They can be positive, negative, or zero. One of the powerful features of Python 3 is that integers have **unlimited precision**. They can grow to any size that your system's memory can accommodate, eliminating the risk of integer overflow errors common in other languages.

You can also represent integers in different number systems by using specific prefixes:

- **Binary (Base 2):** Prefix with `0b` or `0B` (e.g., `0b1010` is 10).
- **Octal (Base 8):** Prefix with `0o` or `0O` (e.g., `0o12` is 10).
- **Hexadecimal (Base 16):** Prefix with `0x` or `0X` (e.g., `0xA` is 10).

Python allows the use of underscores as visual separators within large numerical literals to enhance readability.

```
age = 25
temperature = -10
```

```
population = 1_400_000_000    # Underscores for readability,
    evaluated as 1400000000

# Base representations
binary_val = 0b1011           # 11 in decimal
hex_val = 0xFF                # 255 in decimal

print(type(age))              # <class 'int'>
print(binary_val, hex_val)
```

Listing 2.5: Integer representations and features

2. Float (`float`)

Floating-point numbers, or floats, represent real numbers that contain a decimal point. Internally, Python stores floats using the IEEE 754 double-precision (64-bit) standard. This provides approximately 15 to 17 decimal digits of precision.

Floats can also be expressed using scientific or exponential notation, which is particularly useful for very large or very small numbers. The letter `e` or `E` signifies "times 10 to the power of".

```
pi = 3.14159
height = 1.75

# Scientific notation
speed_of_light = 3e8          # 3 * 10^8 = 300000000.0
electron_mass = 9.1e-31       # 9.1 * 10^-31

print(type(pi))               # <class 'float'>
print(speed_of_light)
print(electron_mass)
```

Listing 2.6: Float data type and scientific notation

Note: Because floats are represented as binary fractions internally, some decimal numbers cannot be represented exactly. This leads to minor precision issues (e.g., `0.1 + 0.2` yields `0.30000000000000004`).

3. Complex Numbers (`complex`)

Unlike many other languages, Python provides built-in support for complex numbers. A complex number consists of a real part and an imaginary part, suffixed with a `j` or `J` (electrical engineering convention, instead of i).

```
z1 = 3 + 4j
z2 = complex(2, -5)           # Another way to create complex
    numbers

print(type(z1))               # <class 'complex'>
print(z1.real)                # 3.0 (Returns a float)
print(z1.imag)                # 4.0 (Returns a float)
print(z1 + z2)                # (5-1j) (Supports arithmetic
    operations)
```

Listing 2.7: Complex numbers

4. String (`str`)

A string is an ordered, sequential collection of Unicode characters used to represent text. Strings are enclosed in single quotes (`'...'`), double quotes (`"..."`), or triple quotes (`'''...'''` or `"""..."""`). Triple quotes are primarily used for multi-line strings or docstrings (function documentation).

Strings in Python are **immutable**. This means once a string object is created in memory, its contents cannot be altered in-place. Any operation that appears to modify a string actually generates a brand new string object.

```
name = 'Alice'
greeting = "Hello, World!"
multi_line = """This string
spans multiple lines."""

# Indexing and Slicing
print(greeting[0])            # H (0-indexed)
print(greeting[-1])           # ! (Negative index counts from
    the end)
print(greeting[0:5])          # Hello (Slicing: [start:stop]
    excluding stop)

# String Methods
print(greeting.lower())       # hello, world!
print(greeting.upper())       # HELLO, WORLD!
print(greeting.replace("World", "Python")) # Hello, Python!

# String Formatting (f-strings, introduced in Python 3.6)
age = 20
intro = f"My name is {name} and I am {age} years old."
print(intro)
```

Listing 2.8: Working with Strings

5. Boolean (`bool`)

A Boolean represents one of two possible states of truth: `True` or `False`. In Python, Boolean constants must be capitalized.

Under the hood, Booleans are implemented as a subclass of the integer type. `True` evaluates to 1 and `False` evaluates to 0.

```
is_student = True
has_passed = False

print(type(is_student))       # <class 'bool'>
print(True + True)            # 2 (1 + 1)
print(False * 50)             # 0 (0 * 50)

```

```
# Used extensively in comparisons and logical operations
x = 10
y = 20
print(x < y)                    # True
print(x == y)                   # False
```

Listing 2.9: Boolean operations

6. NoneType (`None`)

`None` is a special singleton object in Python that represents the *absence of a value* or a null value. It is the sole instance of the `NoneType` class. Variables that have been declared but not yet assigned a meaningful value often start as `None`. It is also the default return value of functions that do not explicitly return anything.

```
result = None
print(type(result))             # <class 'NoneType'>

# Best practice is to use 'is' for comparing with None
if result is None:
    print("No result found.")
```

Listing 2.10: The None type

7. Collection Types (Brief Overview)

Python provides powerful compound data types to hold collections of items. These will be explored deeply in Unit 5, but here is a brief introduction:

- **List (`list`):** An ordered, mutable (changeable) collection of items, potentially of mixed types. Defined using square brackets. E.g., `my_list = [1, "apple", 3.14, True]`
- **Tuple (`tuple`):** An ordered, immutable collection. Once defined, elements cannot be added or changed. Defined using parentheses. E.g., `my_tuple = (1, 2, 3)`
- **Set (`set`):** An unordered collection of unique elements. Useful for mathematical set operations (union, intersection). Defined using curly braces. E.g., `my_set = \{1, 2, 3, 3\}` becomes `\{1, 2, 3\}`
- **Dictionary (`dict`):** An unordered collection of key-value pairs. Keys must be immutable and unique. Defined using curly braces with colons. E.g., `student = \{ "name": "Bob", "age": 22\}`

2.7.3 Checking Data Types: `type()` and `isinstance()`

Because Python is dynamically typed, it is often necessary to check the type of a variable dynamically during program execution.

- `type(object)`: Returns the exact class type of the object.

- `isinstance(object, classinfo)`: Returns `True` if the object is an instance of the given class (or a subclass). It is generally preferred over `type()` for checking types because it supports inheritance.

```
x = 42
y = 3.14
z = "Hello"

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

# Using isinstance()
print(isinstance(x, int))      # True
print(isinstance(z, float))    # False

# Checking multiple types
print(isinstance(y, (int, float))) # True (y is either int or
    float)
```

Listing 2.11: Identifying data types

2.7.4 Type Conversion (Type Casting)

Definition: Type Conversion

Type conversion (also known as *type casting*) is the process of converting a value from one data type to another. It ensures that variables are correctly processed by functions or operators that expect specific data types.

Type conversion in Python falls into two main categories: Implicit and Explicit.

Implicit (Automatic) Type Conversion

Python automatically converts smaller or simpler data types to larger or more complex ones during certain operations to prevent data loss. This is known as coercion.

For example, when an integer is added to a float, Python automatically promotes the integer to a float before performing the addition.

```
integer_num = 5        # int
float_num = 2.5        # float

# Python automatically converts integer_num to 5.0
result = integer_num + float_num

print(result)          # 7.5
print(type(result))    # <class 'float'>
```

Listing 2.12: Implicit type conversion

Explicit (Manual) Type Conversion

Sometimes, automatic conversion is not possible or desired. In such cases, the programmer must explicitly instruct Python to convert the data type using built-in constructor functions.

- `int()`: Constructs an integer from a float (by truncating the decimal) or a string containing a whole number.
- `float()`: Constructs a float from an integer or a string containing a decimal number.
- `str()`: Constructs a string representation of various data types.
- `bool()`: Evaluates a value and returns `True` or `False`.

```
# int() conversions
print(int(3.9))        # 3 (Note: it truncates, does NOT round)
print(int("42"))       # 42 (String to integer)
# print(int("3.14"))   # ValueError: invalid literal for int()

# float() conversions
print(float(7))        # 7.0
print(float("3.14"))   # 3.14

# str() conversions
print(str(100))        # "100"
print(str(True))       # "True"

# bool() conversions (Truthy and Falsy)
print(bool(0))         # False (Zero is falsy)
print(bool(42))        # True  (Non-zero numbers are truthy)
print(bool(""))        # False (Empty sequences are falsy)
print(bool("Python")) # True  (Non-empty sequences are truthy)
print(bool(None))      # False (None is falsy)
```

Listing 2.13: Explicit type conversion

2.7.5 Previous Year Questions and Practical Examples

This section contains solutions to frequently asked exam questions, demonstrating how the concepts of variables and data types are applied to solve practical mathematical problems.

PYQ: Describe data types in Python with examples. (W'23)

Python Data Types: Python provides several built-in data types. Every variable in Python has a type determined by the value assigned to it at runtime.

Type Concept	Keyword	Description & Example
Integer	`int`	Whole numbers of arbitrary size. E.g., `age = 25`
Floating Point	`float`	Numbers with decimal points. E.g., `pi = 3.1415`
String	`str`	Immutable text sequences. E.g., `name = "Alice"`
Boolean	`bool`	Represents True or False. E.g., `is_valid = True`
Complex	`complex`	Numbers with real and imaginary parts. E.g., `z = 3 + 4j`
List	`list`	Mutable ordered collection. E.g., `nums = [1, 2, 3]`
Tuple	`tuple`	Immutable ordered collection. E.g., `t = (1, 2, 3)`
Set	`set`	Unordered collection of unique items. E.g., `s = {1, 2, 3}`
Dictionary	`dict`	Key-value mappings. E.g., `d = {"a": 1, "b": 2}`
NoneType	`None`	Represents the absence of a value. E.g., `result = None`

PYQ: List various data types in Python and explain any three with example. (W'24)

List of Python data types: `int`, `float`, `str`, `bool`, `complex`, `list`, `tuple`, `set`, `dict`, `NoneType`.

1. `int` (Integer): Represents whole numbers without any fractional part. They can be positive, negative, or zero. Python 3 integers have unlimited precision.

```
marks = 95
temperature = -5
print(type(marks))     # Output: <class 'int'>
```

2. `float` (Floating Point): Represents real numbers that contain a decimal point. They are useful for continuous data like measurements, prices, and scientific calculations.

```
price = 49.99
height = 1.75
print(type(price))     # Output: <class 'float'>
```

3. `str` (String): Represents a sequence of Unicode characters used to store text data. Strings are immutable and can be manipulated using indexing, slicing, and built-in string methods.

```python
city = "Ahmedabad"
print(type(city))       # Output: <class 'str'>
print(city[0])          # Output: A (Accessing the first
    character)
print(len(city))        # Output: 9 (Finding the length of
     the string)
```

PYQ: Develop Python code to read three numbers and find their average. (Su'23)

```python
# Read three numbers from user and find average
# Using float() to explicitly convert string inputs to
    floating-point numbers
a = float(input("Enter first number: "))
b = float(input("Enter second number: "))
c = float(input("Enter third number: "))

# Calculate the sum
total = a + b + c

# Calculate the average
average = total / 3

print(f"Sum = {total}")
print(f"Average = {average}")
```

Listing 2.14: Average of three numbers

Output

```
Enter first number: 10
Enter second number: 20
Enter third number: 30
Sum = 60.0
Average = 20.0
```

PYQ: Develop Python code to convert temperature from Celsius to Fahrenheit. (W'23)

Formula utilized: $F = \left(\frac{9}{5} \times C\right) + 32$

```python
# Temperature conversion: Celsius to Fahrenheit
# The input() function returns a string, so we must cast
    it to a float.
celsius = float(input("Enter temperature in Celsius: "))
```

```python

# Applying the mathematical conversion formula
fahrenheit = (9 / 5) * celsius + 32

# Using f-strings to format the output nicely to 2
    decimal places
print(f"{celsius°}C is equivalent to {fahrenheit:.2f°}F")
```

Listing 2.15: Celsius to Fahrenheit conversion

Output

```
Enter temperature in Celsius: 100
100.0°C is equivalent to 212.00°F
```

PYQ: Develop a program to calculate simple interest and compound interest. (Su'24 OR)

Mathematical Formulae:

- Simple Interest (SI): $SI = \dfrac{P \times R \times T}{100}$
- Compound Interest (CI): $CI = P \times \left(1 + \dfrac{R}{100}\right)^{T} - P$

```python
# Program to compute Simple and Compound Interest
P = float(input("Enter Principal amount (P): "))
R = float(input("Enter Rate of Interest in % (R): "))
T = float(input("Enter Time in years (T): "))

# 1. Simple Interest Calculation
SI = (P * R * T) / 100
total_amount_si = P + SI
print(f"\nSimple Interest Calculation:")
print(f"Interest Generated = {SI:.2f}")
print(f"Total Amount (Principal + SI) = {total_amount_si
    :.2f}")

# 2. Compound Interest Calculation
# Note: The ** operator is used for exponentiation (
    power)
Amount_CI = P * ((1 + R / 100) ** T)
CI = Amount_CI - P
print(f"\nCompound Interest Calculation:")
print(f"Interest Generated = {CI:.2f}")
print(f"Total Amount (Principal + CI) = {Amount_CI:.2f}"
    )
```

Listing 2.16: Simple and compound interest calculation

Output

```
Enter Principal amount (P): 1000
Enter Rate of Interest in % (R): 10
Enter Time in years (T): 2

Simple Interest Calculation:
Interest Generated = 200.00
Total Amount (Principal + SI) = 1200.00

Compound Interest Calculation:
Interest Generated = 210.00
Total Amount (Principal + CI) = 1210.00
```

PYQ: Create a program to calculate total and average marks of a student based on four subject marks. (W'24)

```python
# Calculate total and average marks for a student across
    4 subjects
print("Please enter the marks obtained out of 100:")
s1 = float(input("Enter marks for Subject 1: "))
s2 = float(input("Enter marks for Subject 2: "))
s3 = float(input("Enter marks for Subject 3: "))
s4 = float(input("Enter marks for Subject 4: "))

# Summing up the marks
total = s1 + s2 + s3 + s4

# Calculating the average
average = total / 4

print("\n--- Result Summary ---")
print(f"Total Marks Obtained: {total:.2f} / 400.00")
print(f"Average Percentage:   {average:.2f}%")
```

Listing 2.17: Total and average marks calculation

2.8 Operators in Python

An **operator** is a special symbol or keyword that tells the Python interpreter to perform a specific mathematical, relational, logical, or bitwise operation on one or more **operands**. Operands are the values or variables that the operators act upon. For example, in the expression `5 + 3`, the `+` is the operator, and `5` and `3` are the operands. The combination of operators and operands forms an **expression**, which the Python interpreter evaluates to produce a resulting value.

Python has a rich and versatile set of built-in operators designed to handle a wide range of programming scenarios. They can be broadly categorized into several types based on their functionality:

- Arithmetic Operators
- Comparison (Relational) Operators
- Assignment Operators
- Logical Operators
- Bitwise Operators
- Identity Operators
- Membership Operators

Understanding how each of these operators works, their nuances, and their order of precedence is crucial for writing accurate, efficient, and bug-free Python code. In the following subsections, we will explore each category of operators in detail, along with practical examples.

2.8.1 1. Arithmetic Operators

Arithmetic operators are used with numeric values to perform common mathematical operations such as addition, subtraction, multiplication, and division. Python supports operations on integers, floating-point numbers, and even complex numbers.

Operator	Name	Description	Example (x=10, y=3)
`+`	Addition	Adds values on either side of the operator.	`x + y` → 13
`-`	Subtraction	Subtracts right-hand operand from left-hand operand.	`x - y` → 7
`*`	Multiplication	Multiplies values on either side of the operator.	`x * y` → 30
`/`	True Division	Divides left-hand operand by right-hand operand, always resulting in a floating-point number.	`x / y` → 3.333...
`//`	Floor Division	Divides and returns the integer part of the quotient (truncating towards negative infinity).	`x // y` → 3
`%`	Modulo	Divides left-hand operand by right-hand operand and returns the remainder.	`x % y` → 1
`**`	Exponentiation	Calculates the exponential (power) value: x^y.	`x ** y` → 1000

Table 2.2: Python Arithmetic Operators

Nuances of Arithmetic Operators in Python:

- **True Division (`/`) vs. Floor Division (`//`):** In Python 3, the `/` operator always returns a `float`, even if the numbers divide evenly (e.g., `10 / 2` evaluates to `5.0`). On the other hand, the `//` operator performs floor division, rounding the result down to the nearest integer. If operands are floats, floor division returns a float representing an integer value (e.g., `10.0 // 3` evaluates to `3.0`).
- **Floor Division with Negative Numbers:** Floor division rounds towards negative infinity. Therefore, `-10 // 3` results in `-4`, not `-3`.
- **Modulo Operator with Negative Numbers:** The modulo operator (`\%`) in Python takes the sign of the divisor (right operand). Thus, `-10 \% 3` yields `2`, whereas `10 \% -3` yields `-2`. This is mathematically consistent with the relationship: `a = (a // b) * b + (a \% b)`.

```
x = 10
y = 3

print("Addition:         ", x + y)    # 13
print("Subtraction:      ", x - y)    # 7
print("Multiplication:   ", x * y)    # 30
print("True Division:    ", x / y)    # 3.3333333333333335
```

```
print("Floor Division:  ", x // y)   # 3
print("Modulo:          ", x % y)    # 1
print("Exponentiation:  ", x ** y)   # 1000

# Edge cases
print("Negative Floor:  ", -10 // 3) # -4
print("Negative Modulo: ", -10 % 3)  # 2
```

Listing 2.18: Comprehensive Arithmetic operators demonstration

2.8.2 2. Comparison (Relational) Operators

Comparison operators, also known as relational operators, are used to compare the values of two operands. They evaluate the relationship between the operands and return a Boolean value: `True` if the condition is met, and `False` otherwise. These operators are fundamental in decision-making statements like `if`, `elif`, and `while` loops.

Operator	Name	Description	Example (x=5, y=8)
`==`	Equal to	Returns True if the values of two operands are equal.	`x == y` → False
`!=`	Not equal to	Returns True if the values of two operands are not equal.	`x != y` → True
`>`	Greater than	Returns True if the left operand is strictly greater than the right.	`x > y` → False
`<`	Less than	Returns True if the left operand is strictly less than the right.	`x < y` → True
`>=`	Greater than or equal	Returns True if the left operand is greater than or equal to the right.	`x >= y` → False
`<=`	Less than or equal	Returns True if the left operand is less than or equal to the right.	`x <= y` → True

Table 2.3: Python Comparison Operators

Chaining Comparison Operators: A unique and elegant feature of Python is the ability to chain comparison operators. For example, instead of writing `x > 0 and x < 10`, you can write `0 < x < 10`. This syntax is more readable and mathematically intuitive. The chained expression `a < b < c` is evaluated as `a < b and b < c`.

```
a = 15
b = 20
```

```
3
4  print("Equal to:", a == b)           # False
5  print("Not equal to:", a != b)       # True
6  print("Greater than:", a > 10)       # True
7
8  # Chaining comparisons
9  age = 25
10 is_young_adult = 18 <= age < 30
11 print("Is young adult?", is_young_adult)  # True
```

Listing 2.19: Comparison operators and chaining

2.8.3 3. Assignment Operators

Assignment operators are used to assign values to variables. The most basic assignment operator is the equals sign (=), which assigns the value on its right to the variable on its left.

Python also provides compound (or augmented) assignment operators. These operators combine an arithmetic or bitwise operation with assignment, providing a more concise way to update a variable's value based on its current value. For instance, `x += 5` is a shorthand for `x = x + 5`.

Operator	Equivalent to	Example (Assume initial x=10)
`=`	Simple assignment	`x = 10`
`+=`	`x = x + y`	`x += 5` → x becomes 15
`-=`	`x = x - y`	`x -= 3` → x becomes 7
`*=`	`x = x * y`	`x *= 2` → x becomes 20
`/=`	`x = x / y`	`x /= 4` → x becomes 2.5
`//=`	`x = x // y`	`x //= 3` → x becomes 3
`%=`	`x = x % y`	`x %= 3` → x becomes 1
`**=`	`x = x ** y`	`x **= 2` → x becomes 100

Table 2.4: Python Assignment Operators

The Walrus Operator (:=): Introduced in Python 3.8, the assignment expression operator, colloquially known as the "walrus operator", allows you to assign a value to a variable as part of a larger expression. This is particularly useful in `while` loops or `if` statements to avoid calculating the same value twice.

```
1  # Without walrus operator
2  user_input = "Hello World"
3  n = len(user_input)
4  if n > 10:
5      print(f"Input is too long ({n} characters)")
6
```

```python
# With walrus operator
if (n := len(user_input)) > 10:
    print(f"Input is too long ({n} characters)")
```

Listing 2.20: Walrus operator example

2.8.4 4. Logical Operators

Logical operators are used to combine multiple conditional statements or to reverse the logical state of an expression. They evaluate to either `True` or `False`. These operators are essential for building complex logical conditions.

Operator	Description	Example
`and`	Logical AND: Returns True if *both* operands are True.	`(5 > 3) and (8 > 2)` → True
`or`	Logical OR: Returns True if *at least one* operand is True.	`(5 > 10) or (8 > 2)` → True
`not`	Logical NOT: Unary operator that reverses the Boolean value of its operand.	`not (5 > 3)` → False

Table 2.5: Python Logical Operators

Short-Circuit Evaluation: Python evaluates logical expressions using short-circuit logic. This means that the evaluation stops as soon as the outcome is determined.

- For `A and B`: If `A` is `False`, the overall expression cannot be `True`. Python immediately returns `False` (or the value of `A`) without evaluating `B`.
- For `A or B`: If `A` is `True`, the overall expression is guaranteed to be `True`. Python immediately returns `True` (or the value of `A`) without evaluating `B`.

This behavior can be leveraged to prevent runtime errors, such as checking if a variable is not `None` before accessing its attributes.

```python
def complex_function():
    print("Function executed")
    return True

# Short-circuit in 'and': complex_function() is NOT called
result1 = False and complex_function()

# Short-circuit in 'or': complex_function() is NOT called
result2 = True or complex_function()
```

Listing 2.21: Short-circuit evaluation

2.8.5 5. Bitwise Operators

Bitwise operators perform operations at the bit level. These operators treat numeric operands as strings of binary digits (bits) and perform operations bit by bit. They are primarily used in low-level programming, such as cryptography, network programming, and systems programming, where manipulation of individual bits is required.

In Python, integers are internally represented in binary format. Bitwise operators work directly on this binary representation.

Operator	Name	Description
`&`	Bitwise AND	Sets each bit to 1 if both corresponding bits are 1.
`|`	Bitwise OR	Sets each bit to 1 if at least one of the corresponding bits is 1.
`^`	Bitwise XOR	Sets each bit to 1 if only one of the corresponding bits is 1.
`~`	Bitwise NOT	Inverts all bits (1s become 0s, 0s become 1s).
`<<`	Left Shift	Shifts the bits to the left by the specified number of positions, filling with zeros from the right. Equivalent to multiplying by powers of 2.
`>>`	Right Shift	Shifts the bits to the right by the specified number of positions. Equivalent to integer division by powers of 2.

Table 2.6: Python Bitwise Operators

```
a = 10   # Binary: 1010
b = 4    # Binary: 0100

print("a & b =", a & b)    # 0000 -> 0
print("a | b =", a | b)    # 1110 -> 14
print("a ^ b =", a ^ b)    # 1110 -> 14
print("~a =", ~a)          # Inverts bits, result is -(a + 1)
    -> -11
print("a << 1 =", a << 1) # 10100 -> 20 (multiplied by 2)
print("a >> 1 =", a >> 1) # 0101 -> 5 (divided by 2)
```

Listing 2.22: Bitwise operators demonstration

2.8.6 6. Identity Operators

Identity operators are used to compare the memory locations of two objects. They do not compare the values of the objects, but rather check if two variables point to the exact same object in memory. This is particularly relevant in Python due to its object referencing model.

Operator	Description	Example
`is`	Returns True if both variables point to the same object in memory.	`x is y`
`is not`	Returns True if variables do not point to the same object in memory.	`x is not z`

Table 2.7: Python Identity Operators

Difference between == and `is`: The == operator checks for *value equality* (whether two objects have the same content), while the `is` operator checks for *identity* (whether they are the exact same object in memory).

```
list1 = [1, 2, 3]
list2 = [1, 2, 3]
list3 = list1

print(list1 == list2)  # True: They have the same values
print(list1 is list2)  # False: They are different objects in memory
print(list1 is list3)  # True: list3 points to the exact same object as list1
```

Listing 2.23: Identity vs Equality

2.8.7 7. Membership Operators

Membership operators are used to test whether a specific value or variable is found within a sequence. A sequence in Python can be a string, list, tuple, set, or the keys of a dictionary. These operators provide a clean and highly readable way to perform inclusion checks.

Operator	Description	Example
`in`	Returns True if the value is found in the sequence.	`"a" in "apple"` → True
`not in`	Returns True if the value is NOT found in the sequence.	`5 not in [1, 2, 3]` → True

Table 2.8: Python Membership Operators

```
# String membership
print('h' in 'hello')          # True

# List membership
vowels = ['a', 'e', 'i', 'o', 'u']
print('x' not in vowels)       # True
```

```
7
8  # Dictionary membership (checks keys, not values)
9  user = {"name": "John", "age": 30}
10 print("name" in user)          # True
11 print("John" in user)          # False
```

Listing 2.24: Membership operators examples

2.8.8 8. Operator Precedence and Associativity

When an expression contains multiple operators, Python needs a set of rules to determine the order in which the operations are evaluated. This set of rules is known as **Operator Precedence**. Operators with higher precedence are evaluated before those with lower precedence.

If operators have the same precedence, Python uses **Associativity** to determine the order. Most operators in Python are evaluated from left to right (Left-to-Right associativity). However, exponentiation (`**`) is evaluated from right to left.

The following table lists the operators from highest precedence to lowest:

Operator	Description
`()`	Parentheses (Grouping)
`**`	Exponentiation
`+x`, `-x`, `~x`	Unary plus, Unary minus, Bitwise NOT
`*`, `/`, `//`, `%`	Multiplication, Division, Floor Division, Modulo
`+`, `-`	Addition and Subtraction
`<<`, `>>`	Bitwise Left and Right Shifts
`&`	Bitwise AND
`^`	Bitwise XOR
`|`	Bitwise OR
`==`, `!=`, `>`, `>=`, `<`, `<=`, `is`, `is not`, `in`, `not in`	Comparisons, Identity, and Membership
`not`	Logical NOT
`and`	Logical AND
`or`	Logical OR
`=`, `+=`, `-=`, ...	Assignment Operators

Table 2.9: Python Operator Precedence (Highest to Lowest)

Parentheses can always be used to explicitly define the order of evaluation, overriding the default precedence rules. This not only ensures mathematical correctness but also significantly improves code readability.

```
1  result = 10 + 2 * 3
```

```python
print(result) # Output: 16 (Multiplication has higher precedence)

result_parens = (10 + 2) * 3
print(result_parens) # Output: 36 (Parentheses override precedence)

# Right-to-left associativity for exponentiation
power_result = 2 ** 3 ** 2
print(power_result) # Output: 512 (Calculated as 2 ** (3 ** 2) = 2 ** 9)
```

Listing 2.25: Operator Precedence

2.8.9 Past Examination Questions

PYQ: Discuss membership operator of Python with example. (W'24)

Membership Operators: Used to test whether a value or variable is present in a sequence (such as a string, list, tuple, set, or dictionary keys). They evaluate to a Boolean value.

Operators:

- `in` — Returns `True` if the value is found within the specified sequence.
- `not in` — Returns `True` if the value is NOT found within the specified sequence.

```python
# Membership operators demonstration
fruits = ["apple", "banana", "cherry"]

print("apple" in fruits)        # True
print("grape" in fruits)        # False
print("grape" not in fruits)    # True

# Works on strings too
word = "Python"
print("P" in word)              # True
print("z" not in word)         # True

# Works on dict (checks keys)
student = {"name": "Alice", "age": 20}
print("name" in student)        # True
print("marks" in student)       # False
```

Listing 2.26: Membership operators example

PYQ: Write a short note on assignment operator in Python. (W'24 OR)

Assignment Operators: These operators are used to assign values to variables. The fundamental assignment operator in Python is `=`, which assigns the value on its right side to the variable on its left.
In addition to the simple assignment, Python provides several compound (or augmented) assignment operators that combine a mathematical or bitwise operation with the assignment itself. This allows for cleaner, more compact code.

```
x = 10              # Simple assignment
print("Initial x =", x)

x += 5              # Equivalent to: x = x + 5
print("After +=5:", x)     # 15

x -= 3              # Equivalent to: x = x - 3
print("After -=3:", x)     # 12

x *= 2              # Equivalent to: x = x * 2
print("After *=2:", x)     # 24

x //= 5             # Equivalent to: x = x // 5
print("After //=5:", x)    # 4

x **= 3             # Equivalent to: x = x ** 3
print("After **=3:", x)    # 64

x %= 10             # Equivalent to: x = x % 10
print("After %%=10:", x)   # 4
```

Listing 2.27: Assignment operators demonstration

PYQ: List all logical operators and explain each with Python code example. (Su'24)

Python Logical Operators: These operators are used to evaluate and combine Boolean expressions, returning a final Boolean value based on logical rules.

1. `and` — Returns `True` only if BOTH operands evaluate to `True`. If any operand is `False`, it returns `False`.
2. `or` — Returns `True` if AT LEAST ONE of the operands evaluates to `True`. It returns `False` only if both operands are `False`.
3. `not` — A unary operator that reverses the logical state of its operand. If the operand is `True`, it returns `False`; if `False`, it returns `True`.

```
a = True
```

```python
b = False

# 'and' operator
print(a and b)        # False (both must be True)
print(a and True)     # True

# 'or' operator
print(a or b)         # True (at least one is True)
print(b or False)     # False

# 'not' operator
print(not a)          # False
print(not b)          # True

# Practical example: checking voting eligibility
age = 20
has_id = True
can_vote = (age >= 18) and has_id
print("Can vote:", can_vote)  # True
```

Listing 2.28: Logical operators demonstration

PYQ: List assignment operators in Python and build Python code to demonstrate any three. (Su'23)

Assignment operators in Python: =, +=, -=, *=, /=, //=, \%=, **=, \&=, |=, \^=, >>=, <<=

```python
# Demonstrating +=, *= and **= assignment operators
num = 100

# 1. Addition assignment (+=)
num += 50           # num = 100 + 50 = 150
print("After += 50:", num)    # 150

# 2. Multiplication assignment (*=)
num *= 2            # num = 150 * 2 = 300
print("After *= 2:", num)     # 300

# 3. Exponentiation assignment (**=)
base = 3
base **= 4          # base = 3 ** 4 = 81
print("After **= 4:", base)   # 81
```

Listing 2.29: Three assignment operators demonstrated

PYQ: Develop code to find square and cube of a given number input by user. (W'24 OR)

```python
# Find square and cube of user input number
num = float(input("Enter a number: "))

# Using the exponentiation operator (**)
square = num ** 2
cube   = num ** 3

print(f"Number  = {num}")
print(f"Square  = {square}")
print(f"Cube    = {cube}")
```

Listing 2.30: Square and cube of a number

Output

```
Enter a number: 5
Number  = 5.0
Square  = 25.0
Cube    = 125.0
```

CHAPTER 3

Flow of Control

3.1 Introduction to Flow of Control

In our journey of learning Python programming, we have so far written programs where instructions are executed sequentially. By default, Python executes statements **sequentially** — starting from the very first line at the top of the script and moving downwards, one line at a time, until the end of the file is reached. This linear execution model is sufficient for basic scripts that simply calculate a value or print a fixed output. However, real-world problems are rarely purely linear.

Imagine waking up in the morning: your subsequent actions depend on various conditions. If it is raining, you might take an umbrella; if it is sunny, you might wear sunglasses. Furthermore, some tasks require repetition, such as stirring a cup of coffee until the sugar is completely dissolved. Software programs must similarly adapt to different conditions, inputs, and states. Real-world programs need the ability to make **decisions** (executing different blocks of code based on a specific condition) and to **repeat** blocks of code multiple times (loops). These essential capabilities are provided by Python's **control flow** statements.

Definition: Flow of Control

Flow of control (or control flow) refers to the specific order in which individual statements, instructions, or function calls of an imperative program are executed or evaluated. In Python, the control flow is dictated by the program logic, which evaluates conditions and directs execution accordingly. Python provides three primary control structures to manage this flow:

1. **Sequence (Sequential Flow):** The default top-to-bottom execution mechanism where no branching or looping occurs. Statements are executed one after another in the exact order they appear.

2. **Selection (Decision-Making/Branching):** The capability to execute different, alternative code paths based on whether a specific condition evaluates to True or False. This is achieved using `if`, `elif`, and `else` statements.

3. **Repetition (Iteration/Looping):** The ability to repeatedly execute a block of code as long as a condition remains true or for a specific number of times. This is implemented using `for` and `while` loops.

Understanding control flow is arguably one of the most critical aspects of learning any programming language. It is what transforms a script from a static list of commands into a dynamic, responsive program that can handle varied user inputs, process complex data structures, and automate repetitive tasks efficiently.

3.2 Understanding Conditions and Boolean Logic

Before diving into selection statements, it is crucial to understand how decisions are formulated in Python. Decisions are based on **conditions**, which are expressions that evaluate to either `True` or `False`. These Boolean values are the fundamental building blocks of control flow.

3.2.1 Relational Operators

Conditions are most commonly created using relational (comparison) operators. These operators compare two values and return a Boolean result:

- `==` (Equal to): Returns `True` if both operands are equal. (e.g., `5 == 5` is `True`)
- `!=` (Not equal to): Returns `True` if operands are not equal. (e.g., `5 != 3` is `True`)
- `>` (Greater than): Returns `True` if the left operand is greater than the right.
- `<` (Less than): Returns `True` if the left operand is less than the right.
- `>=` (Greater than or equal to): Returns `True` if the left operand is greater or equal.
- `<=` (Less than or equal to): Returns `True` if the left operand is less or equal.

3.2.2 Logical Operators

To formulate more complex conditions, multiple relational expressions can be combined using logical operators:

- `and`: Returns `True` if **both** conditions are True. (e.g., `age >= 18 and citizen == "yes"`)
- `or`: Returns `True` if **at least one** condition is True. (e.g., `day == "Saturday" or day == "Sunday"`)
- `not`: Reverses the Boolean value of a condition. (e.g., `not (x == 5)` becomes `True` if `x` is not 5).

3.2.3 Truth Value Testing (Truthy and Falsy)

In Python, any object can be tested for its truth value, not just explicit Boolean expressions. Python classifies values as either "truthy" or "falsy".

- **Falsy values** include: `False`, `None`, numeric zero of all types (`0`, `0.0`, `0j`), and empty sequences/collections (like `""`, `[]`, `()`, `\{\}`).

- **Truthy values** are everything else. By default, any non-zero number or non-empty string/list evaluates to `True` in a conditional context.

This feature allows for concise and Pythonic code, such as `if my_list:` instead of `if len(my_list) > 0:`.

3.3 Selection Statements (Decision Making)

Selection statements enable a program to test one or more conditions and execute specific blocks of code depending on the outcomes. Python provides several mechanisms for decision-making.

3.3.1 The `if` Statement

The `if` statement is the most fundamental selection statement. It evaluates a single condition. If the condition is `True`, the block of code immediately following the `if` statement (its body) is executed. If the condition is `False`, the body is completely skipped, and the program continues with the next sequential statement after the `if` block.

Syntax: if statement

```
if condition:
    statement(s)   # executed only if condition is True
```

Notice the colon (`:`) at the end of the `if` line. This colon tells Python that a new block of code is about to begin.

Key Concept: Python Indentation is Mandatory

Unlike C, C++, Java, or JavaScript, which use curly braces `\{\}` to define the scope of a code block, Python uses **indentation** (whitespace at the start of a line). The standard convention dictated by PEP 8 (Python's style guide) is to use **4 spaces** per indentation level.
Indentation is not just for readability in Python; it is a syntactical requirement. All statements within the same block must be indented by the exact same amount. If you fail to indent consistently, Python will raise an `IndentationError` and refuse to run your program. This design choice enforces clean, highly readable code.

```
1  age = 20
2  print("Checking eligibility...")
3
4  # The condition age >= 18 evaluates to True
5  if age >= 18:
6      print("You are eligible to vote.")
7      print("Please proceed to the registration desk.")
8
9  # This unindented statement is outside the if-block
10 print("Program continues here regardless of age.")
```

Listing 3.1: Simple if statement example

Output

```
Checking eligibility...
You are eligible to vote.
Please proceed to the registration desk.
Program continues here regardless of age.
```

In the example above, because the `age` variable is initialized to 20, the condition `age >= 18` evaluates to `True`. Therefore, the two indented `print` statements inside the `if` block are executed. The final `print` statement is unindented, meaning it is not part of the `if` structure and will execute unconditionally.

3.3.2 The `if...else` Statement

The simple `if` statement is useful when we want to execute code only when a condition is met. However, frequently, we also have an alternative action we want to perform when the condition is **not** met. For this scenario, Python provides the `if...else` statement.

The `if...else` statement offers two mutually exclusive paths. If the condition is `True`, the block of code under the `if` clause executes. If the condition is `False`, the block of code under the `else` clause executes. It is guaranteed that exactly one of the two blocks will run.

Syntax: if...else statement

```
if condition:
    statement(s)    # executed if condition is True
else:
    statement(s)    # executed if condition is False
```

Like the `if` statement, the `else` keyword must be followed by a colon (`:`), and the code block beneath it must be indented. Furthermore, the `else` keyword must be aligned precisely with its corresponding `if` statement.

```
# Take input from the user and convert it to an integer
num = int(input("Enter an integer number: "))

# Check if the remainder when divided by 2 is zero
if num % 2 == 0:
    print(f"{num} is an Even number.")
else:
    print(f"{num} is an Odd number.")
```

Listing 3.2: if...else example: determining even or odd

In this program, we use the modulo operator (`\%`) to find the remainder of the division by 2. If the remainder is strictly equal to 0, the number is even, and the `if` block runs. Otherwise, the number must be odd, and the `else` block runs instead.

3.3.3 The `if...elif...else` Ladder

When there are **multiple conditions** to test, an `if...else` statement is insufficient on its own. While you could technically nest `if...else` statements deeply, this rapidly becomes unreadable. Instead, Python offers the `elif` (short for "else if") keyword to chain multiple conditions together cleanly.

In an `if...elif...else` ladder, Python evaluates each condition in sequential order from top to bottom. As soon as Python encounters a condition that evaluates to `True`, it executes the corresponding block of code and entirely skips the rest of the ladder. If none of the `if` or `elif` conditions are `True`, the optional `else` block at the end is executed as a fallback or default action.

Syntax: if-elif-else ladder

```
if condition1:
    block1           # Executed if condition1 is True
elif condition2:
    block2           # Executed if condition1 is False AND condition2 is True
elif condition3:
    block3           # Executed if condition1, condition2 are False AND condition3
...
else:
    default_block    # Executed if ALL above conditions are False
```

A key detail to remember is that **at most one** block in an `if...elif...else` chain will execute. Even if multiple conditions would mathematically be true, only the first one encountered will trigger its block.

```
marks = int(input("Enter the student's marks (0-100): "))

# Validate input first
if marks < 0 or marks > 100:
    print("Invalid marks entered. Please enter a value
   between 0 and 100.")
else:
    # Determine the grade using a ladder
    if marks >= 90:
        grade = "O (Outstanding)"
    elif marks >= 80:
        grade = "A+ (Excellent)"
    elif marks >= 70:
        grade = "A (Very Good)"
    elif marks >= 60:
        grade = "B+ (Good)"
    elif marks >= 50:
        grade = "B (Above Average)"
    elif marks >= 40:
        grade = "C (Pass)"
    else:
        grade = "F (Fail)"

```

```
    print(f"The student's grade is: {grade}")
```

Listing 3.3: Grade calculator using if-elif-else

In the above example, if a student enters 85, the first condition (`marks >= 90`) is `False`. The execution then checks the next condition (`marks >= 80`), which is `True`. Python assigns `"A+ (Excellent)"` to the `grade` variable and then skips the rest of the `elif` and `else` blocks completely.

3.3.4 Nested `if` Statements

A **nested** `if` is simply an `if` statement placed inside the block of another `if`, `elif`, or `else` statement. Nesting is extremely useful when a decision heavily depends on the outcome of a previous decision, representing multi-level conditional testing.

When writing nested `if` statements, consistent indentation is paramount. The inner `if` statement must be further indented to indicate that it belongs inside the outer block. Python allows deep nesting, but excessively nested code (sometimes called "spaghetti code") can become difficult to read, maintain, and debug. As a best practice, if you find yourself nesting beyond three levels, you should consider restructuring your logic using `and`/`or` operators or extracting the logic into a separate function.

```
a = int(input("Enter first number (a): "))
b = int(input("Enter second number (b): "))
c = int(input("Enter third number (c): "))

if a >= b:
    # We are here if a is greater than or equal to b
    # Now we just need to compare 'a' with 'c'
    if a >= c:
        print(f"Maximum is {a}")
    else:
        # If 'a' is >= 'b', but 'c' is > 'a', then 'c' is the
    largest
        print(f"Maximum is {c}")
else:
    # We are here if 'b' is strictly greater than 'a'
    # Now we compare 'b' with 'c'
    if b >= c:
        print(f"Maximum is {b}")
    else:
        print(f"Maximum is {c}")
```

Listing 3.4: Nested if: Find the maximum of three numbers

This algorithm efficiently finds the largest of three numbers by narrowing down the candidates step by step. If `a >= b`, we definitively eliminate `b` from being the maximum, so the inner logic only compares `a` and `c`.

3.3.5 The `pass` Statement

In Python, a code block cannot be completely empty. Sometimes, you might want to write an `if` statement to outline your logic, but you have not yet decided what code to put

inside it. If you leave the block entirely empty, Python will throw an `IndentationError` or a `SyntaxError`.

To handle this, Python provides the `pass` statement. The `pass` statement is a null operation; it does absolutely nothing. It simply acts as a placeholder so that the syntactical requirement of having a code block is fulfilled.

```
age = 25

if age >= 18:
    pass  # TODO: Implement complex voter registration logic later
else:
    print("Minor")
```

Listing 3.5: Using the pass statement as a placeholder

3.3.6 Conditional Expressions (Ternary Operator)

For very simple `if...else` situations where you just want to assign a value based on a condition, Python offers a concise syntax known as a **conditional expression** (often called the ternary operator in other programming languages). It allows you to write a simple `if...else` block in a single line.

Syntax: Conditional Expression

```
variable = value_if_true if condition else value_if_false
```

Instead of writing:

```
age = 22
if age >= 18:
    status = "Adult"
else:
    status = "Minor"
```

You can write this elegant, single-line equivalent:

```
age = 22
status = "Adult" if age >= 18 else "Minor"
print(status)  # Output: Adult
```

While conditional expressions save vertical space, they should be used judiciously. Overusing them or nesting them (e.g., `val1 if cond1 else val2 if cond2 else val3`) severely harms readability and defeats the purpose of clean Pythonic code.

3.3.7 Structural Pattern Matching (The `match` Statement)

Introduced recently in Python 3.10, the `match` statement acts as a powerful alternative to long and repetitive `if...elif...else` chains. It is conceptually similar to the `switch ...case` statements found in legacy languages like C, C++, and Java, but it brings advanced pattern-matching capabilities.

The `match` statement takes an expression and compares its value against successive `case` blocks. When a match is found, the corresponding block executes.

```
status_code = int(input("Enter an HTTP status code: "))

match status_code:
    case 200:
        print("OK - The request was successful.")
    case 404:
        print("Not Found - The requested resource could not be found.")
    case 500:
        print("Internal Server Error - Something went wrong on the server.")
    case 401 | 403:
        # We can match multiple values using the bitwise OR operator (|)
        print("Authentication/Authorization Error.")
    case _:
        # The underscore (_) acts as a wildcard or default case (like 'else')
        print("Unknown status code.")
```

Listing 3.6: Using the match statement for menu selection

Pattern matching goes beyond simple value comparison; it can unpack data structures like lists, tuples, and dictionaries directly inside the `case` statement, making it a highly expressive tool for advanced data parsing. However, for basic numeric or string inequality comparisons (e.g., `age > 18`), `if...elif...else` remains the required standard approach.

3.4 Best Practices and Common Pitfalls

When working with selection statements, beginner programmers often encounter a few common stumbling blocks:

- **Assignment instead of comparison:** A classic mistake is using the assignment operator (`=`) instead of the equality operator (`==`) inside an `if` condition. In Python, trying to write `if x = 5:` will raise a `SyntaxError`, preventing you from accidentally running faulty logic (unlike in C or C++ where it silently causes logical bugs). Always ensure you use `==` for comparisons.
- **Explicitly comparing to True/False:** Beginners often write `if is_valid == True:`. While this works technically, it is un-Pythonic and redundant. Simply write `if is_valid:`. Similarly, for false checks, instead of `if is_valid == False:`, use `if not is_valid:`.
- **Deep nesting (Arrow Anti-Pattern):** Writing many nested `if` statements creates a shape that looks like an arrowhead (`>`) pointing to the right margin of your screen. This makes code hard to read on smaller screens and difficult to mentally trace. Use techniques like returning early, using logical operators (`and`/`or`), or creating helper functions to flatten your logic structure.

3.5 Previous Year Questions (PYQ) and Solutions

The following section contains frequently asked questions from previous university examinations. Mastering these concepts is critical for both academic success and practical programming competence.

PYQ: Explain if...else statement with suitable example. (Su'23, Su'25, W'25)

if...else Statement: Provides two alternative execution paths. If the test expression (condition) evaluates to `True`, the `if` block executes. If the test expression evaluates to `False`, the `else` block is executed. It guarantees that exactly one of the two blocks will execute, providing a clear binary choice.
Syntax:

```
if condition:
    # executes when condition is True
else:
    # executes when condition is False
```

Example — Check if a number is positive or negative:

```
num = int(input("Enter a number: "))
if num >= 0:
    print("The number is Positive (or Zero)")
else:
    print("The number is Negative")
```

In this snippet, any number greater than or equal to zero triggers the first print statement. Any negative number falls back to the else block.

PYQ: Explain if-elif-else statement with flowchart and suitable example. (W'24)

if-elif-else Statement: A multi-way branching construct designed to test multiple conditions in sequence. Python tests each condition linearly from the top down. The very first condition that evaluates to `True` triggers its corresponding block of code to run. Once a block runs, the entire rest of the `if-elif-else` structure is bypassed. If none of the explicit conditions are met, the optional `else` block serves as a default fallback.
Syntax:

```
if condition1:
    block1          # runs if condition1 is True
elif condition2:
    block2          # runs if condition2 is True (and condition1 False)
else:
    default_block   # runs if ALL conditions are False
```

Example — Day of week determination:

```python
day = int(input("Enter day number (1-7): "))

if day == 1:
    print("Monday")
elif day == 2:
    print("Tuesday")
elif day == 3:
    print("Wednesday")
elif day == 4:
    print("Thursday")
elif day == 5:
    print("Friday")
elif day == 6:
    print("Saturday")
elif day == 7:
    print("Sunday")
else:
    print("Invalid day number. Please enter a value between 1 and 7.")
```

PYQ: Explain Nested if statement in Python with example. (W'23, Su'24, W'25 OR)

Nested if: An `if` statement logically placed entirely inside another `if`, `elif`, or `else` statement block. This allows programmers to test secondary conditions only if a primary condition has already been met. It is essential for multi-level hierarchical conditional logic. Proper indentation is heavily emphasized to indicate logical grouping.

```python
# Nested if: check if a person is fully eligible to vote
age = int(input("Enter your age: "))

# First-level condition
if age >= 18:
    print("Age criteria met.")
    citizen = input("Are you a legal citizen? (yes/no): ").lower()

    # Second-level condition (Nested)
    if citizen == "yes":
        print("Verification complete. You are fully eligible to vote.")
    else:
        print("Eligibility failed. Only legal citizens can vote.")
else:
    print(f"Eligibility failed. You must be 18 or older. You have {18 - age} years left.")
```

Listing 3.7: Nested if: Comprehensive voting eligibility check

PYQ: Develop Python programs: To check if entered number is odd or even; To find average of numbers from 1 to n. (Su'25 OR)

Program 1 — Checking Odd or Even Number: This program uses the modulo operator (\%) which returns the remainder of a division. An even number divided by 2 leaves a remainder of 0.

```
num = int(input("Enter an integer number: "))

if num % 2 == 0:
    print(f"The number {num} is Even")
else:
    print(f"The number {num} is Odd")
```

Program 2 — Average of Numbers from 1 to n: This program utilizes both a sequence generator (`range`) and iteration (`for` loop) to aggregate a sum before calculating the average.

```
n = int(input("Enter the value of n: "))

if n <= 0:
    print("Please enter a positive integer greater than 0.")
else:
    total_sum = 0
    # Loop from 1 up to and including n
    for i in range(1, n + 1):
        total_sum += i

    average = total_sum / n
    print(f"The total sum is {total_sum}.")
    print(f"The average of numbers from 1 to {n} is {average:.2f}")
```

PYQ: Explain different types of selection/decision making structures with example. (W'23)

Selection (Decision-making) Statements in Python: Python implements several forms of decision-making to handle different logical requirements:
1. Simple `if` Statement: Executes a target block of code only when a specified condition evaluates to True. It is a single-branch decision.

```
balance = 5000
if balance > 1000:
```

```
    print("You have sufficient funds.")
```

2. `if...else` Statement: Provides exactly two alternative paths. It handles the True outcome and explicitly handles the False outcome.

```
temperature = 35
if temperature > 30:
    print("It is a hot day.")
else:
    print("The weather is pleasant.")
```

3. `if-elif-else` Ladder: Evaluates multiple sequential conditions. Useful when an entity can fall into one of several distinct categories.

```
traffic_light = "Yellow"
if traffic_light == "Red":
    print("Stop the vehicle.")
elif traffic_light == "Yellow":
    print("Slow down and prepare to stop.")
elif traffic_light == "Green":
    print("You may go.")
else:
    print("Invalid signal light.")
```

4. Nested `if` Statement: A decision within a decision. Tests secondary logic dependent on a primary logic's result.

```
has_ticket = True
has_id = False

if has_ticket:
    if has_id:
        print("Access Granted. Enjoy the movie.")
    else:
        print("Access Denied. Please show your ID.")
else:
    print("Access Denied. You must purchase a ticket.")
```

3.6 Repetition (Looping Statements)

In programming, we frequently encounter situations where a specific task needs to be performed repeatedly. For example, you might want to process a list of user inputs, display numbers from 1 to 100, or send an email to every subscriber in a database. Writing the same lines of code multiple times is not only tedious and error-prone but also violates the DRY (Don't Repeat Yourself) principle of software development.

To handle such repetitive tasks efficiently, Python provides loop structures. Loops allow a block of code to be executed **repeatedly** as long as a certain condition remains True, or for a predetermined number of iterations over a sequence. By using loops, we can write concise, readable, and maintainable code. Python primarily features two main looping constructs: the `for` loop and the `while` loop. Each serves a distinct purpose and is chosen based on the specific requirements of the problem at hand.

3.6.1 The `for` Loop

The `for` loop in Python is a definite iteration structure, meaning it is designed to run a specific number of times. Unlike `for` loops in languages like C or Java, which rely on a counter variable, the Python `for` loop is specifically designed to iterate directly over a **sequence** (such as a list, tuple, string, or dictionary) or any other *iterable* object. It automatically fetches each item from the sequence one by one and executes the loop's body for that item.

Syntax: for loop

```
for loop_variable in sequence:
    statement(s)   # body - executed once for each item in the sequence
```

In this syntax, `loop_variable` is a temporary variable that takes the value of the next item in the `sequence` during each iteration. The `statement(s)` form the body of the loop, which must be uniformly indented.

The `range()` Function

When we want to run a `for` loop for a specific number of times without iterating over an existing collection like a list, we often pair it with the `range()` function. The `range()` function is a built-in Python tool that generates a sequence of immutable integers. It is highly flexible and can be called with one, two, or three arguments:

- `range(stop)`: Generates a sequence of numbers starting from 0 up to, but not including, the `stop` value. For example, `range(5)` produces 0, 1, 2, 3, 4.
- `range(start, stop)`: Generates numbers from `start` up to `stop - 1`. For example, `range(1, 6)` produces 1, 2, 3, 4, 5.
- `range(start, stop, step)`: Generates numbers from `start` to `stop - 1`, incrementing (or decrementing) by the `step` value. A negative step allows for counting backwards. For example, `range(10, 0, -2)` produces 10, 8, 6, 4, 2.

```python
# 1. Using range() with one argument (0 to 4)
print("Range 5:")
for i in range(5):
    print(i, end=" ")
print("\n")

# 2. Using range() to print numbers from 1 to 5
print("Range 1 to 6:")
for i in range(1, 6):
    print(i, end=" ")
print("\n")

# 3. Using range() with a step size to print even numbers
print("Even numbers up to 10:")
for i in range(2, 11, 2):
    print(i, end=" ")
print("\n")

# 4. Counting backwards with a negative step
print("Countdown:")
for i in range(5, 0, -1):
    print(i, end=" ")
print("\n")

# 5. Iterating over a list of strings
fruits = ["apple", "banana", "cherry"]
print("List iteration:")
for fruit in fruits:
    print(f"I like {fruit}s.")

# 6. Iterating over a string (character by character)
print("\nString iteration:")
for char in "Python":
    print(char, end="-")
print()
```

Listing 3.8: for loop with range and sequence examples

Output

```
Range 5:
0 1 2 3 4

Range 1 to 6:
1 2 3 4 5

Even numbers up to 10:
2 4 6 8 10

Countdown:
```

```
5 4 3 2 1

List iteration:
I like apples.
I like bananas.
I like cherrys.

String iteration:
P-y-t-h-o-n-
```

3.6.2 The `while` Loop

The `while` loop in Python is an indefinite iteration structure. It repeatedly executes a target block of code **as long as a given Boolean condition evaluates to True**. The condition is evaluated **before** executing the loop's body in each iteration. Because the condition is checked at the top of the loop, if it evaluates to False on the very first check, the body of the loop is completely bypassed and never executes.

Syntax: while loop

```
while condition:
    statement(s)   # body - executed as long as the condition remains True
```

A well-structured `while` loop generally relies on three critical components to ensure logical correctness:

1. **Initialization:** Setting up a loop control variable before the loop begins.
2. **Condition:** A boolean expression tested before every iteration to decide whether the loop should run.
3. **Update:** Modifying the loop control variable inside the loop body so that the condition eventually becomes False, terminating the loop.

Failure to update the control variable properly will result in an **infinite loop**—a loop that runs forever and causes the program to hang, freeze, or consume excessive memory resources.

```
# 1. Simple while loop to print numbers 1 to 5
print("Counting to 5:")
i = 1                    # Initialization
while i <= 5:            # Condition
    print(i, end=" ")
    i += 1               # Update step - extremely important!
print("\n")

# 2. Calculating the sum of the first N natural numbers
total = 0
n = 1
limit = 100
```

```
while n <= limit:
    total += n
    n += 1
print(f"Sum of numbers from 1 to {limit} is: {total}")

# 3. Processing user input until a specific condition is met
print("\nInteractive Loop Example:")
user_input = ""
# The loop continues until the user types 'quit'
while user_input.lower() != "quit":
    # In an actual interactive shell, you would use input() here:
    # user_input = input("Enter a command (type 'quit' to exit): ")
    # For demonstration, we directly assign to exit:
    user_input = "quit"
print("Exited the interactive loop.")
```

Listing 3.9: while loop examples and avoiding infinite loops

Output

```
Counting to 5:
1 2 3 4 5

Sum of numbers from 1 to 100 is: 5050

Interactive Loop Example:
Exited the interactive loop.
```

3.6.3 Comparison: `for` vs `while`

While both loops achieve repetition, they are conceptually different and are suited for distinct scenarios. Understanding when to use which loop is a fundamental skill in programming.

Feature	`for` loop	`while` loop
Primary Use Case	Best when the number of iterations is **known** in advance or when you need to process every item in a collection.	Best when the number of iterations is **unknown** and depends on a dynamic condition being met.
Iteration Control	Automatically extracts the next element from the sequence or range without manual intervention.	Must manually update the loop control variable inside the loop block.
Condition Check	Implicitly checks for the exhaustion of the sequence (end of iterable).	Explicitly tests a Boolean expression at the start of every iteration.
Risk of Infinite Loops	Very low. The loop naturally terminates when it reaches the end of the finite sequence.	High. If the update step is forgotten or the condition logic is flawed, the loop may never end.
Typical Scenarios	Iterating over arrays, lists, strings; executing a block a fixed, predefined number of times.	Reading a file until the end; validating user input until it is correct; running a continuous game loop.

Table 3.1: Detailed Comparison of `for` and `while` Loops

3.6.4 Nested Loops

A nested loop is simply a loop placed entirely within the body of another loop. Python allows you to nest any type of loop inside any other type of loop (e.g., a `for` loop inside a `for` loop, a `while` loop inside a `for` loop, etc.).

When dealing with nested loops, it is crucial to understand the execution flow: for every single iteration of the **outer loop**, the **inner loop** executes completely from its beginning to its end. For example, if the outer loop runs 5 times and the inner loop is set to run 5 times, the code inside the inner loop will be executed a total of $5 \times 5 = 25$ times.

Nested loops are heavily relied upon when working with multi-dimensional data structures (like 2D lists or matrices) and for generating complex grid-like formats or visual patterns on the console.

```
# 1. Printing a matrix / multiplication table
print("Multiplication Table (1 to 5):")
for i in range(1, 6):           # Outer loop controls rows
    for j in range(1, 6):       # Inner loop controls columns
        # Format output to take up 4 spaces for alignment
        print(f"{i*j:4}", end="")
    print()                     # Move to the next line after
   finishing a row

print("\nRight-angled Triangle Pattern:")
```

```
# 2. Printing a star pattern
rows = 5
for i in range(1, rows + 1):    # Outer loop determines the
    row number
     for j in range(1, i + 1):  # Inner loop determines stars
    in that row
          print("*", end=" ")
     print()                    # New line for the next row
```

Listing 3.10: Nested loops: Multiplication table and Star Pattern

Output

```
Multiplication Table (1 to 5):
   1   2   3   4   5
   2   4   6   8  10
   3   6   9  12  15
   4   8  12  16  20
   5  10  15  20  25

Right-angled Triangle Pattern:
*
* *
* * *
* * * *
* * * * *
```

3.7 Loop Control and Jump Statements

Normally, a loop continues to execute until its given sequence is completely exhausted (in a `for` loop) or its condition evaluates to False (in a `while` loop). However, situations often arise where you need to alter this normal sequential flow. For instance, you might want to terminate the loop early upon finding a specific item, or perhaps skip the current iteration and move directly to the next one. Python provides **jump statements**—namely `break`, `continue`, and `pass`—to exert fine-grained control over loop execution.

3.7.1 The `break` Statement

The `break` statement is used to **immediately terminate** the loop in which it resides. When the Python interpreter encounters a `break` statement, it entirely abandons the current loop and transfers execution control to the very next statement immediately following the loop block. If the `break` statement is used inside a nested loop, it will only terminate the innermost loop that contains it.

The `break` statement is incredibly useful in linear search algorithms, where you are looking for a specific item in a collection and want to stop searching as soon as you find it, saving valuable processing time. It is also used to safely exit intentional infinite loops.

```
# Example 1: Exiting a for loop early
print("Searching for the number 7:")
```

```python
for num in range(1, 15):
    print(f"Checking {num}...")
    if num == 7:
        print("Found the lucky number 7! Stopping search.")
        break  # Exit the loop entirely
print("Search complete.\n")

# Example 2: Breaking out of a while loop
print("Finding the first multiple of 13 greater than 50:")
counter = 51
while True:  # Intentional infinite loop
    if counter % 13 == 0:
        print(f"The number is {counter}.")
        break  # The only way to exit the infinite loop
    counter += 1
```

Listing 3.11: Using the break statement

Output

```
Searching for the number 7:
Checking 1...
Checking 2...
Checking 3...
Checking 4...
Checking 5...
Checking 6...
Checking 7...
Found the lucky number 7! Stopping search.
Search complete.

Finding the first multiple of 13 greater than 50:
The number is 52.
```

3.7.2 The `continue` Statement

While the `break` statement terminates the entire loop, the `continue` statement acts more subtly: it only terminates the **current iteration**. When `continue` is executed, the interpreter skips any remaining statements in the current loop body and abruptly jumps back to the top of the loop. For a `for` loop, it retrieves the next item in the sequence. For a `while` loop, it re-evaluates the loop condition.

The `continue` statement is highly effective when you want to filter data—processing valid values while simply ignoring or bypassing problematic or irrelevant ones.

```python
print("Printing numbers from 1 to 10, but skipping 4 and 7:")
for i in range(1, 11):
    if i == 4 or i == 7:
        continue  # Skip the rest of the body for 4 and 7
    # This print statement is ignored when i is 4 or 7
    print(i, end=" ")
print("\n")
```

```
8
9  print("Printing only odd numbers:")
10 counter = 0
11 while counter < 10:
12     counter += 1
13     if counter % 2 == 0:
14         continue  # Skip even numbers
15     print(counter, end=" ")
16 print()
```

Listing 3.12: Using the continue statement

Output

```
Printing numbers from 1 to 10, but skipping 4 and 7:
1 2 3 5 6 8 9 10

Printing only odd numbers:
1 3 5 7 9
```

3.7.3 The pass Statement

The `pass` statement is a unique feature in Python. It is a **null operation**—when it is executed, absolutely nothing happens. It is primarily used as a syntactic placeholder. Python's syntax requires a code block (an indented section) after control flow statements like `if`, `for`, `while`, or function definitions. If you are designing your code structure and haven't yet decided what logic to put inside a loop or conditional block, you cannot simply leave it blank, as that will raise an `IndentationError`. Placing a `pass` statement safely satisfies the interpreter and solves this problem.

```
1 # A loop that does nothing yet
2 for item in [1, 2, 3]:
3     # TODO: Implement processing logic later
4     pass  # Prevents IndentationError
5
6 print("Loop executed successfully without doing anything.")
```

Listing 3.13: Using the pass statement

3.7.4 The else Clause with Loops

Python supports a unique and often misunderstood feature: attaching an `else` clause directly to `for` and `while` loops. The `else` block is executed **only if the loop completes all its iterations naturally** (i.e., the loop condition eventually evaluates to False, or the sequence is completely exhausted, and it was *not* interrupted by a `break` statement). If the loop is terminated prematurely by a `break`, the `else` block is completely skipped.

This is particularly elegant in search operations where you want to execute some code if an item was *not* found after checking all elements, avoiding the need for cumbersome boolean flag variables.

```python
search_target = 99
numbers = [10, 20, 30, 40, 50]

print(f"Searching for {search_target}...")
for num in numbers:
    if num == search_target:
        print(f"Found {search_target}!")
        break
else:
    # This executes ONLY if the loop finishes without hitting a break
    print(f"Item {search_target} was not found in the list.")
```

Listing 3.14: Using else with loops

Output

```
Searching for 99...
Item 99 was not found in the list.
```

3.8 Previous Year Questions (PYQs)

PYQ: Explain for loop statement with example. (Su'23 OR, W'23 OR, W'24 OR, Su'25, W'25)

`for` Loop Overview: The `for` loop in Python is utilized to iterate over a sequence (such as a list, string, or tuple) or an iterable range of numbers. It systematically executes a block of code once for each item in the sequence, sequentially retrieving each element without requiring manual initialization or counter management.
Syntax:

```
for variable in sequence:
    statement(s)
```

Example 1 — Print numbers from 1 to 10: Using the `range()` function, we can generate an integer sequence from 1 up to (but not including) 11.

```
for i in range(1, 11):
    print(i, end=" ")
```

Output: `1 2 3 4 5 6 7 8 9 10`
Example 2 — Calculating the sum of elements in a list: The loop iterates directly through the list elements, adding each to an accumulator variable.

```
nums = [10, 20, 30, 40, 50]
total = 0
for n in nums:
    total += n
print("Sum =", total)  # Output will be: Sum = 150
```

PYQ: Explain while loop with syntax, flowchart and Python code example. (Su'24)

`while` Loop Overview: A `while` loop repeatedly executes a target block of code as long as a specified condition evaluates to True. The condition is evaluated at the very beginning of each iteration. If the condition becomes False, the loop terminates immediately. It is absolutely essential to update the variables within the loop body to avoid infinite loop scenarios.
Syntax:

```
while boolean_condition:
    statement(s)      # Executed while condition is True
    # Loop update expression (e.g., counter += 1)
```

Flowchart Description:

1. **START**
2. Initialise the loop control variable.
3. Evaluate the **Condition**.

4. If **YES (True)**: Execute the loop body, update the control variable, and loop back to step 3.
5. If **NO (False)**: Exit the loop and continue with the rest of the program.
6. **STOP**

Example Code:

```
# Print numbers from 1 to 5 using while loop
i = 1                # Initialization
while i <= 5:        # Condition
    print(i, end=" ")
    i += 1           # Update
```

Output: 1 2 3 4 5

PYQ: Give differences between for loop and while loop. Develop a Python program to print numbers 1 to 25 using for loop. (W'25)

Differences between `for` and `while` loops:

for loop	while loop
Typically used when the exact number of iterations is known in advance.	Typically used when the number of iterations is unknown and relies on a dynamic condition.
Iterates over a sequence (like lists, strings) or a specific `range()`.	Iterates purely based on a boolean condition remaining True.
The loop counter is managed implicitly by the iterator.	The loop counter must be initialised and manually updated by the programmer.
Carries a significantly lower risk of resulting in an infinite loop.	Carries a high risk of an infinite loop if the condition is never updated to False.

Python program to print numbers 1 to 25 using a `for` loop:

```
print("Numbers from 1 to 25:")
for i in range(1, 26):
    print(i, end=" ")
```

Output: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25

PYQ: Describe the use of break statement with necessary examples. (Su'23)

`break` Statement Application: The `break` statement acts as an emergency exit for a loop structure. When the interpreter encounters a `break` statement, it immediately terminates the current loop's execution. Control is instantly transferred to the first statement following the loop block.
Use Cases: It is highly useful for halting search operations early once the target is successfully located, preventing unnecessary iterations. It is also used to break out of interactive, indefinite programs when the user signals they wish to exit.
Example Program:

```
# Exit the loop as soon as we find the number 5
nums = [1, 2, 3, 4, 5, 6, 7, 8, 9]
for n in nums:
    if n == 5:
        print("Found 5! Breaking out of loop early.")
        break  # The loop stops entirely here
    print(n, end=" ")
print("\nExecution continues here after the loop.")
```

Output:

```
1 2 3 4 Found 5! Breaking out of loop early.
Execution continues here after the loop.
```

PYQ: Describe the use of continue statement with necessary examples. (Su'23 OR, Su'24 OR)

`continue` Statement Application: The `continue` statement is used to skip the remainder of the code inside the current iteration of the loop. Unlike `break`, it does not terminate the entire loop. Instead, it forces the loop to proceed directly back to the top to initiate the next iteration.
Use Cases: It is an excellent tool for data filtering operations, such as ignoring invalid inputs, passing over missing data, or intentionally bypassing certain values (like skipping even numbers to only process odd ones).
Example Program:

```
# Print numbers from 1 to 10, skipping multiples of 3
for i in range(1, 11):
    if i % 3 == 0:
        continue       # Skips printing for 3, 6, and 9
    print(i, end=" ")
```

Output:

```
1 2 4 5 7 8 10
```

PYQ: Write Python code to show whether the entered number is prime or not. (Su'23 7M, W'25 OR)

Prime Number Logic: A prime number is a positive integer strictly greater than 1 that has no positive divisors other than 1 and itself. To optimize the program, instead of checking all possible divisors up to the given number, we only need to mathematically check up to the square root of the number. If no divisor is found in that range, the number is definitively prime.

Example Program:

```
# Take integer input from the user
num = int(input("Enter a positive integer: "))

is_prime = True  # Assume it is prime initially

# Prime numbers must be strictly greater than 1
if num < 2:
    is_prime = False
else:
    # Check for factors from 2 up to the square root of the number
    # We use int(num ** 0.5) to calculate the square root efficiently
    for i in range(2, int(num ** 0.5) + 1):
        if num % i == 0:  # If num is evenly divisible by i, it's not prime
            is_prime = False
            break         # No need to check further, exit the loop

# Output the final result based on the boolean flag
if is_prime:
    print(f"{num} is a Prime number.")
else:
    print(f"{num} is NOT a Prime number.")
```

PYQ: Develop a program to find odd and even numbers from 1 to N using loops. (W'23 OR)

Program Logic: This script prompts the user for an upper limit N, and then iterates through the range from 1 to N. By using the modulo operator (\%), it evaluates the remainder. If dividing by 2 leaves a remainder of 0, the number is even. If it leaves a remainder of 1, the number is odd.

Example Program:

```
n = int(input("Enter upper limit N: "))

print(f"\nEven numbers from 1 to {n}:")
```

```
# Loop to find and print even numbers
for i in range(1, n + 1):
    if i % 2 == 0:  # Even condition (divisible by 2)
        print(i, end=" ")

print(f"\n\nOdd numbers from 1 to {n}:")
# Loop to find and print odd numbers
for i in range(1, n + 1):
    if i % 2 != 0:  # Odd condition (not perfectly
   divisible by 2)
        print(i, end=" ")
print()
```

Sample Output:

```
Enter upper limit N: 10

Even numbers from 1 to 10:
2 4 6 8 10

Odd numbers from 1 to 10:
1 3 5 7 9
```

3.9 Pattern Programs Using Nested Loops

Pattern printing is a classic and highly effective application of nested loops in programming. It serves as an excellent pedagogical tool to help students visualize the flow of control, understand loop iterations, and master the manipulation of loop variables. In a pattern program, the outer loop typically controls the rows, while the inner loop controls the elements printed across each column within a given row. Because of their ability to test logical thinking and loop construction, pattern programs are extremely common in university and GTU examinations.

3.9.1 Understanding Nested Loops

Before diving into complex patterns, it is crucial to understand the fundamental concept of a nested loop. A firm grasp on nested iterations is the foundation for solving any pattern-related programming problem.

PYQ: Explain nested loop with a suitable example. (W'24 OR)

Nested Loop Concept: A nested loop is simply a loop written inside the body of another loop. The outer loop determines how many times the inner loop will be executed. For every single iteration of the outer loop, the inner loop completes its entire cycle of iterations from start to finish. This makes nested loops ideal for working with multi-dimensional data, such as a two-dimensional grid, a matrix, or a coordinate system where an (x, y) position is required.

Syntax:

```
for outer_variable in sequence_outer:
    # Outer loop body starts
    for inner_variable in sequence_inner:
        # Inner loop body
        # This executes completely for EACH iteration of the outer loop
    # Outer loop body continues
```

Execution Flow: When the program execution reaches the outer loop, its first iteration begins. Then, the program encounters the inner loop. The inner loop will run through all of its iterations until it terminates. Once the inner loop finishes, the outer loop completes its first iteration, evaluates its condition again, and moves to its second iteration. The inner loop then restarts and runs completely again. This process continues until the outer loop sequence is exhausted.

Example — 3x3 Multiplication Grid: Consider a scenario where we want to generate a mathematical multiplication table in a grid format. We can use the outer loop to represent the multiplier (rows) and the inner loop to represent the multiplicand (columns).

```
for i in range(1, 4):      # Outer loop: controls rows (1, 2, 3)
    for j in range(1, 4): # Inner loop: controls columns (1, 2, 3)
```

```
        # Calculate the product and print it.
        # \t is a tab character to align the columns.
        print(i * j, end="\t")
    print() # Moves to the next line after completing a
    row
```

Listing 3.15: Generating a 3x3 grid

Output

```
1 2 3
2 4 6
3 6 9
```

3.9.2 The Mechanics of Pattern Printing

When constructing pattern programs, we are essentially plotting characters on a two-dimensional console screen. The logic always boils down to three primary components:

- **The Outer Loop (Rows):** The outer loop is responsible for generating the rows. If you want a pattern with 5 rows, the outer loop must iterate 5 times. The loop variable (often named `i`) acts as the row counter.
- **The Inner Loop (Columns):** The inner loop is responsible for printing the actual characters (stars, numbers, or letters) from left to right on a single line. The number of times the inner loop runs depends on the current row. The loop variable (often named `j`) acts as the column counter.
- **The Print Functions:** In Python, the `print()` function automatically adds a newline character at the end of the output. To print elements side-by-side on the same row, we must override this default behavior using the `end` parameter.
  - `print(element, end=" ")` prints the element followed by a space, keeping the cursor on the same line.
  - `print()` (with no arguments) is placed outside the inner loop but inside the outer loop. It simply outputs a newline, moving the cursor to the beginning of the next row once all columns for the current row are printed.

3.9.3 Left-Aligned Increasing Triangles

The most fundamental shape in pattern programming is the left-aligned increasing right-angled triangle. In this pattern, the first row has 1 element, the second row has 2 elements, the third row has 3 elements, and so on.

The logic here is that the **number of columns is equal to the row number**. Therefore, if the outer loop variable `i` represents the row number, the inner loop should run exactly `i` times.

PYQ: Develop Python program to print the following star pattern (W'23): * / * * / * * * / * * * * / * * * * *

```
# Left-aligned star triangle
n = 5 # Total number of rows

# Outer loop: i goes from 1 to 5
for i in range(1, n + 1):
    # Inner loop: j runs 'i' times
    for j in range(i):
        print("*", end=" ") # Print star on the same line
    print()                  # Move to the next line after the row finishes
```

Listing 3.16: Left-aligned growing star triangle

Output

```
*
* *
* * *
* * * *
* * * * *
```

Dry Run of the Star Triangle Logic: Let us trace the execution step-by-step for `n = 3` to understand the variable states:

1. **Outer loop `i = 1`:** Inner loop `range(1)` runs for `j = 0`. Prints one `*`. Inner loop ends. Outer loop executes `print()` to go to a new line.
2. **Outer loop `i = 2`:** Inner loop `range(2)` runs for `j = 0, 1`. Prints two `*` characters separated by space. Inner loop ends. New line.
3. **Outer loop `i = 3`:** Inner loop `range(3)` runs for `j = 0, 1, 2`. Prints three `*` characters. Inner loop ends. New line.

This logic can be applied to any character or symbol simply by changing the string literal inside the `print()` function. It is very common to see variations of this basic structure in exams.

PYQ: Create program to display: (A) & character triangle (B) number triangle 1/12/123/1234/12345. (Su'25)

Part A — & character triangle:

```
n = 5
for i in range(1, n + 1):
    for j in range(i):
        print("&", end=" ")
    print()
```

Listing 3.17: & character triangle

Output

```
&
& &
& & &
& & & &
& & & & &
```

Part B — Number triangle (1 / 1 2 / 1 2 3 / ...): Instead of printing a static character like `"*"` or `"&"`, we print the inner loop variable `j`. Notice that we start the inner loop from 1 instead of 0 to get the desired output numbers.

```
n = 5
for i in range(1, n + 1):
    # Inner loop j goes from 1 to i
    for j in range(1, i + 1):
        print(j, end=" ")
    print()
```

Listing 3.18: Number triangle 1 to n

Output

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

PYQ: Write Python code to display the following patterns: (A) number triangle (B) inverted star triangle. (Su'23 OR)

Part A — Number triangle:

```
n = 5
for i in range(1, n + 1):
    for j in range(1, i + 1):
        print(j, end=" ")
    print()
```

Listing 3.19: Number triangle

Output

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

Part B — Inverted star triangle: This will be explicitly detailed in the subsequent section on inverted patterns, but the code relies on running the outer loop backward.

```
n = 5
for i in range(n, 0, -1):
    for j in range(i):
        print("*", end=" ")
    print()
```

Listing 3.20: Inverted star triangle

Output

```
* * * * *
* * * *
* * *
* *
*
```

3.9.4 Inverted Triangles (Decreasing Patterns)

Inverted patterns start with the maximum number of elements on the first row and decrease by one element in each subsequent row.

There are two primary ways to approach the logic for inverted triangles:

1. **Decreasing Outer Loop:** Run the outer loop backward. If `n = 5`, the outer loop generates the sequence `5, 4, 3, 2, 1`. The inner loop then simply runs `i` times. This is the most straightforward approach in Python using the `range(start, stop, step)` function with a negative step.

2. **Mathematical Formula:** Run the outer loop normally from `1` to `n`. Make the inner loop run `n - i + 1` times. If `n=5` and `i=1`, it runs `5-1+1 = 5` times. Both logic styles yield the exact same pattern output, though the first is generally preferred for its readability.

PYQ: Create program: (A) @ inverted character triangle (B) squared number series (1 / 1 4 / 1 4 9 / 1 4 9 16). (W'25)

Part A — Inverted @ triangle: Here, we utilize the first approach: a decreasing outer loop.

```
n = 5
# Start at 5, stop before 0, step backwards by -1
for i in range(n, 0, -1):
    # Inner loop runs 'i' times.
    # Row 1: i=5 (runs 5 times). Row 2: i=4 (runs 4 times)...
    for j in range(i):
        print("@", end=" ")
    print()
```

Listing 3.21: Inverted @ triangle

Output

```
@ @ @ @ @
@ @ @ @
@ @ @
@ @
@
```

Part B — Squared number series (1 / 1 4 / 1 4 9 / 1 4 9 16): This is a variation of the standard increasing number triangle. Instead of printing the loop variable `j` directly, we print its square (`j * j`) or use the exponentiation operator (`j ** 2`).

```
n = 5
for i in range(1, n + 1):
    for j in range(1, i + 1):
        # Calculate the square dynamically
        print(j * j, end=" ")
    print()
```

Listing 3.22: Squared number triangle

Output

```
1
1 4
1 4 9
1 4 9 16
1 4 9 16 25
```

3.9.5 Advanced Number Patterns

While simple number patterns rely directly on the loop indices (`i` or `j`), more complex patterns require external counters or mathematical manipulations to achieve the desired output.

Continuous Counter (Floyd's Triangle)

A classic problem is Floyd's Triangle, where the numbers do not reset at the beginning of each row but instead continue sequentially from where the previous row left off. To achieve this, we must initialize a counter variable *before* the loops begin and increment it every time a number is printed.

PYQ: Write Python program to print the number triangle pattern: 1 / 2 3 / 4 5 6 / 7 8 9 10. (Su'24 OR)

```
# Numbers continue from where the previous row ended
n = 4       # number of rows
num = 1     # Initialize an independent starting number

for i in range(1, n + 1):
    for j in range(i):
        # Print the current value of num
        print(num, end=" ")
        # Increment num so the next iteration prints the
    next integer
        num += 1
    print()
```

Listing 3.23: Consecutive number triangle (Floyd's Triangle)

Output

```
1
2 3
4 5 6
7 8 9 10
```

3.9.6 Alphabet and Character Patterns

Printing alphabet patterns requires an understanding of ASCII (American Standard Code for Information Interchange) values. In programming, every character is represented by a specific integer behind the scenes in the computer's memory.

- The uppercase letter `'A'` starts at ASCII value **65**.
- The uppercase letter `'Z'` ends at ASCII value **90**.
- The lowercase letter `'a'` starts at ASCII value **97**.
- The lowercase letter `'z'` ends at ASCII value **122**.

Python provides two essential built-in functions to bridge the gap between characters and their ASCII integers seamlessly:

- `ord(char)`: Takes a single string character and returns its integer ASCII value. (e.g., `ord('A')` returns `65`).
- `chr(int)`: Takes an integer and returns its corresponding string character. (e.g., `chr(65)` returns `'A'`).

To print alphabetical patterns, we can use an integer counter starting at 65 and convert it to a character using `chr()` right before printing.

PYQ: Create program to display pattern A/AB/ABC/ABCD/ABCDE. (W'24 OR)

```
n = 5
for i in range(1, n + 1):
    for j in range(i):
        # We start with 65 ('A') and add the inner loop variable 'j'
        # When j=0: 65+0=65 -> 'A'
        # When j=1: 65+1=66 -> 'B'
        # When j=2: 65+2=67 -> 'C'
        # Notice we use end="" to print characters tightly without spaces
        print(chr(65 + j), end="")
    print()
```

Listing 3.24: Alphabet triangle A/AB/ABC...

Output

```
A
AB
ABC
ABCD
ABCDE
```

3.9.7 Right-Aligned and Pyramid Patterns (Using Spaces)

Until now, all patterns have been left-aligned, meaning printing begins immediately at the leftmost edge of the console. Right-aligned triangles and centered pyramids require printing blank spaces before printing the actual characters to offset them from the margin.

This requires **two inner loops** inside the main outer loop:

1. The first inner loop prints the leading spaces.
2. The second inner loop prints the characters (stars, numbers, etc.).

Right-Aligned Triangle

In a right-aligned triangle of size 5, the first row has 4 spaces and 1 star. The second row has 3 spaces and 2 stars. The mathematical relationship is that row `i` has `n - i` spaces and `i` stars.

```
n = 5
for i in range(1, n + 1):
    # Inner loop 1: Print spaces
    for j in range(n - i):
        print(" ", end=" ")

    # Inner loop 2: Print stars
    for k in range(i):
        print("*", end=" ")

    print() # Newline after both inner loops are done
```

Listing 3.25: Right-aligned star triangle

Output

```
        *
      * *
    * * *
  * * * *
* * * * *
```

Full Pyramid (Equilateral Triangle)

A full pyramid is remarkably similar to a right-aligned triangle. The only difference is the spacing. If you print a single space *after* each star while keeping the leading spaces tight, the characters naturally center themselves, creating a beautifully balanced pyramid structure.

```
n = 5
for i in range(1, n + 1):
    # Inner loop 1: Print spaces (same count as right-aligned)
    # Using end="" without an extra space to keep spacing tight
    for j in range(n - i):
        print(" ", end="")

    # Inner loop 2: Print stars separated by single spaces
    for k in range(i):
        print("* ", end="")

    print()
```

Listing 3.26: Full Pyramid Pattern

Output

```
    *
   * *
  * * *
 * * * *
* * * * *
```

Diamond Pattern

A diamond pattern is simply a combination of a full pyramid (the top half) and an inverted full pyramid (the bottom half). By stacking these two patterns together, you can create more complex geometric shapes. This demonstrates how basic building blocks can form intricate outputs.

```
n = 5
# Top half (Pyramid)
for i in range(1, n + 1):
    for j in range(n - i):
        print(" ", end="")
    for k in range(i):
        print("* ", end="")
    print()

# Bottom half (Inverted Pyramid)
for i in range(n - 1, 0, -1):
    for j in range(n - i):
        print(" ", end="")
    for k in range(i):
        print("* ", end="")
    print()
```

Listing 3.27: Diamond Pattern

Output

```
    *
   * *
  * * *
 * * * *
* * * * *
 * * * *
  * * *
   * *
    *
```

3.9.8 Summary of Pattern Building Techniques

To master pattern programs and succeed in GTU examinations, keep these critical rules in mind during your practice:

- **Row Count First:** Always identify the total number of rows first. This definitively establishes the boundaries of your outer `for` loop.
- **Analyze Relationships:** Look at how the number of characters in a row relates to the row number. Is it equal to the row number? Is it decreasing? Use this mathematical relationship to set the precise `range()` for your inner loop.
- **Mind the Spaces:** Check if the pattern requires leading spaces. If the left side of the pattern is a straight vertical line, no spaces are needed. If the pattern is pushed to the right or visually centered, you must write a dedicated loop for printing spaces *before* printing the characters.
- **Control the Cursor:** Remember to use `print(..., end="")` or `print(..., end=" ")` to stay on the same line. Critically, do not forget the blank `print()` at the end of the outer loop body to drop to the next line.
- **Data Types and Counters:** For number and alphabet patterns, determine whether the sequence resets on each row (meaning you should use the inner loop variable directly) or continues indefinitely across rows (meaning you must declare and increment an external counter variable).
- **Dry Run Your Code:** Tracing your code manually on a piece of paper by tracking the values of `i` and `j` is the best way to debug a pattern that does not print correctly.

CHAPTER 4
Functions

4.1 Introduction to Functions

As programming problems become larger and more complex, writing all the code in a single, continuous sequence becomes difficult to manage, understand, and debug. To overcome this, programmers use a technique called **modular programming**, which involves breaking a large program down into smaller, self-contained, and manageable units. In Python, these modular units are known as **functions**.

Definition: Function

A **function** is a self-contained, named block of reusable code that performs a specific, well-defined task. Once a function is defined, it can be *called* (or invoked) multiple times from different parts of a program, or even from entirely different programs, without needing to rewrite the code.

Functions are a fundamental building block in Python programming. They provide several significant advantages:

- **Reusability (Write Once, Use Many Times):** The primary benefit of a function is that it eliminates code duplication. If you need to perform the same operation multiple times throughout your program, you write the logic once within a function and simply call the function whenever needed. This prevents redundancy.
- **Modularity (Divide and Conquer):** Complex problems can be broken down into smaller, simpler sub-problems. Each sub-problem can be solved by a separate function. This makes the overall problem easier to conceptualize, solve, and test.
- **Readability and Abstraction:** By giving a descriptive name to a block of code (e.g., `calculate_taxes()`), the code becomes self-documenting. A programmer can understand what the function does just by reading its name, without needing to examine the complex internal logic. This concept of hiding the complex details behind a simple interface is called *abstraction*.
- **Maintainability and Debugging:** When an error occurs or a business logic change is required, you only need to update the code in one place—inside the function. The change will automatically be reflected everywhere the function is called, significantly reducing the chance of introducing new bugs and saving development time.

4.2 Types of Functions in Python

Functions in Python can be broadly categorized into three main types based on their origin and how they are made available to the programmer:

1. **Built-in Functions:** These are pre-defined functions that are always available in the standard Python interpreter. You do not need to import any module to use them. Python comes with a rich set of built-in functions that perform common, everyday tasks. Examples include:
   - `print()`: Outputs textual data or variables to the console.
   - `input()`: Reads string input from the user via the keyboard.
   - `len()`: Returns the length of an object (like the number of characters in a string or items in a list).
   - `type()`: Returns the data type class of an object.
   - `abs()`, `max()`, `min()`, `sum()`, `range()`: Common mathematical and sequence-related operations.
2. **Standard Library Functions:** Python provides a vast standard library organized into various modules. These modules contain pre-written functions tailored for specific domains (like advanced mathematics, statistics, file handling, network communication, etc.). To use these functions, you must first import the corresponding module using the `import` statement.
   - `math.sqrt(x)`: Returns the square root of x (requires `import math`).
   - `random.randint(a, b)`: Returns a random integer between a and b (requires `import random`).
   - `datetime.now()`: Returns the current date and time (requires `import datetime`).
3. **User-Defined Functions:** These are custom functions created by the programmer to perform specific tasks relevant to their unique application. The programmer has full control over defining the function's name, its input parameters, and the algorithmic code it executes. We will focus extensively on user-defined functions throughout the rest of this chapter.

4.3 Defining and Calling Functions

To utilize a user-defined function in Python, a two-step process is required: you must first *define* the function, and later, you can *call* (or execute) it.

4.3.1 Syntax of a Function Definition

Python uses the `def` keyword to introduce a function definition.

Syntax: Function definition and call

```
def function_name(parameter1, parameter2, ...):
    """Docstring: Optional description of the function's purpose."""
    statement(s)
    return value   # Optional
```

Let's break down the individual components of a function definition:

- **`def` keyword:** This is a reserved keyword that explicitly tells the Python interpreter that a new function block is starting.
- **Function Name:** A unique identifier for the function. It should follow the same naming conventions as variables (start with a letter or underscore, contain only alphanumeric characters and underscores). By convention, function names should be descriptive, lowercase, with words separated by underscores to improve readability (e.g., `calculate_total_score`).
- **Parameters (Optional):** Enclosed within parentheses `()`. Parameters act as variables that hold the data the function needs to perform its task. If a function doesn't require any outside data to operate, the parentheses are simply left empty `()`.
- **Colon (`:`):** This punctuation mark signifies the end of the function header and the beginning of the function's indented body.
- **Docstring (Optional but highly recommended):** The first line inside the function body can be an optional multi-line string (enclosed in triple quotes *`"""..."""`*). This is known as a documentation string, or docstring. It explains what the function does, its parameters, and what it returns. Tools and IDEs often use this to provide helpful pop-ups to developers.
- **Function Body:** A block of one or more Python statements that define the operational logic of the function. Python relies on indentation to define scope; therefore, every line in the function body *must be consistently indented* (usually by 4 spaces) relative to the `def` keyword.
- **`return` Statement (Optional):** Used to send a calculated result back to the point where the function was called. If a function is meant to just perform an action (like printing) and not produce data, the `return` statement can be omitted, in which case the function implicitly returns `None`.

4.3.2 Calling a Function

Merely defining a function does not execute the code within it; it simply creates a function object and registers its name. To execute the code, you must **call** the function.

You call a function by writing its name followed by parentheses. If the function requires inputs, you place the *arguments* inside those parentheses.

```
1 # Function Definition
2 def greet_user():
```

```
    """This function prints a simple greeting message."""
    print("Welcome to Python programming!")
    print("Have a great day ahead.")

# Function Call
greet_user()
```

Listing 4.1: Defining and Calling a Simple Function

Understanding Execution Flow: When a running Python script encounters a function call, the following steps occur: 1. The execution of the main program is temporarily paused. 2. The control jumps to the first line of the called function. 3. All statements within the function body are executed sequentially from top to bottom. 4. Once the function finishes (either by reaching the end of its indented block or by hitting a `return` statement), control jumps back to the exact location where the function was originally called, and the main program resumes its execution from that point onwards.

4.4 Parameters and Arguments

In most programming scenarios, functions need external data to process. This data exchange happens using parameters and arguments. While the terms are often used interchangeably in casual conversation, there is a technical distinction:

- **Parameters** are the variables listed inside the parentheses in the function *definition*. They act as empty placeholders waiting to receive data.
- **Arguments** are the actual, concrete values (or variables holding values) passed into the function when it is *called*.

```
def greet_person(name):  # 'name' is the parameter
    print(f"Hello, {name}!")

greet_person("Alice")    # "Alice" is the argument
greet_person("Bob")      # "Bob" is the argument
```

Listing 4.2: Function with Parameters

Python provides highly flexible ways to pass arguments to a function:

4.4.1 Positional Arguments

These are the most common type of arguments. They are passed to a function in the exact same sequential order as the parameters are defined in the function header. The number of arguments provided must exactly match the number of parameters.

```
def subtract(a, b):
    print(f"{a} - {b} = {a - b}")

subtract(10, 5)  # Output: 10 - 5 = 5 (a becomes 10, b
    becomes 5)
subtract(5, 10)  # Output: 5 - 10 = -5 (Order strictly
    matters!)
```

4.4.2 Keyword Arguments

To improve code clarity and bypass positional restrictions, you can pass arguments using the names of the parameters. When using keyword arguments, the order in which they are passed does not matter, as Python explicitly matches the names. This is especially useful for functions with many parameters.

```
def display_info(name, age):
    print(f"Name: {name}, Age: {age}")

# Order doesn't matter when specifying the parameter names explicitly
display_info(age=25, name="John")
```

4.4.3 Default Parameters

You can assign default fallback values to parameters in the function definition. If a corresponding argument is provided during the function call, it overrides the default value. If no argument is provided, the function gracefully falls back to using the default value. *Crucial Rule: Default parameters must be placed after non-default (positional) parameters in the definition header.*

```
def greet(name, message="Good morning!"):
    print(f"Hello {name}, {message}")

greet("Alice")                          # Uses the default message
greet("Bob", "How are you today?")      # Overrides the default message
```

4.4.4 Variable-Length Arguments (*args and **kwargs)

Sometimes, you cannot predict in advance how many arguments a user might pass into your function. Python accommodates this using variable-length argument syntax.

- **`*args` (Non-Keyword Arguments):** By prefixing a parameter name with a single asterisk (`*`), you instruct the function to accept an arbitrary number of positional arguments. Python internally collects these arguments and packs them into a *tuple*. The name `args` is an industry-standard convention; you could legally use `*numbers` or `*values`.

```
def calculate_sum(*args):
    total = 0
    for num in args:
        total += num
    return total

print(calculate_sum(10, 20))        # Output: 30
print(calculate_sum(1, 2, 3, 4, 5)) # Output: 15
```

- **`**kwargs` (Keyword Arguments):** By prefixing a parameter name with a double asterisk (`**`), you allow the function to accept an arbitrary number of keyword arguments. Python internally packs these into a *dictionary*, where the parameter names become the keys and the passed arguments become the associated values.

```
def display_student_info(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

display_student_info(name="Alice", grade="A", age=20)
# Output:
# name: Alice
# grade: A
# age: 20
```

4.5 The `return` Statement

While functions can print output directly to the console, printing is mostly useful for debugging or simple scripts. In real-world applications, it is much more valuable for functions to perform computations and securely pass the resulting data back to the caller. This allows the caller program to store the result in a variable, use it in subsequent mathematical expressions, or pass it along to another function. This data hand-off is achieved using the `return` statement.

When a `return` statement is executed, the function immediately terminates processing, and the specified value is routed back to the exact location of the function call.

4.5.1 Returning a Single Value

```
def calculate_area(length, width):
    area = length * width
    return area

# The returned value is securely stored in the 'rectangle_area' variable
rectangle_area = calculate_area(5, 4)
print(f"The calculated area is {rectangle_area}")
```

Listing 4.3: Function Returning a Calculated Value

4.5.2 Returning Multiple Values

Unlike languages such as C++ or Java, Python provides a highly elegant mechanism allowing a function to return multiple distinct values simultaneously. It achieves this by implicitly grouping the comma-separated values into a *tuple*. The calling program can then immediately unpack these values into separate variables.

```
def get_user_details():
    name = "Alice"
    age = 30
```

```
    city = "New York"
    # Returns an implicit tuple: ("Alice", 30, "New York")
    return name, age, city

# Unpacking the returned tuple into three distinct variables
user_name, user_age, user_city = get_user_details()
print(f"User {user_name} is {user_age} years old and lives in
    {user_city}.")
```

Listing 4.4: Function Returning Multiple Values

4.5.3 Implicit Return (`None`)

If a function block concludes without encountering a `return` statement, or if it executes a standalone `return` statement devoid of a value, Python automatically returns the special built-in object `None`. `None` represents the intentional absence of a value.

4.6 Variable Scope and Lifetime

Variables created in Python are not universally accessible. The region of a program where a variable is actively recognized and accessible is called its **scope**. The duration of time during program execution that a variable resides in the system's memory is called its **lifetime**. Understanding scope is crucial for preventing variable naming collisions and subtle bugs.

4.6.1 Local Scope

Variables that are defined or assigned a value *inside* a function block are designated as **local variables**. They possess a local scope.

- They can only be read or modified from within the specific function where they are created.
- Their lifetime begins the moment the function is called and terminates immediately when the function returns. The memory allocated to them is reclaimed by Python's garbage collector.

```
def my_function():
    local_var = 100  # Local variable restricted to
   my_function
    print("Inside function:", local_var)

my_function()
# print(local_var)  # Error: NameError. local_var does not
   exist here.
```

4.6.2 Global Scope

Variables initialized *outside* of any function wrapper are known as **global variables**. They possess a global scope.

- They can be freely accessed and read from anywhere within the script file, including from inside any function.
- Their lifetime persists for the entire duration the program is actively running.

```
global_var = 50  # Global variable available file-wide

def show_global():
    # Functions can freely read global variables
    print("Inside function accessing global:", global_var)

show_global()
print("Outside function:", global_var)
```

4.6.3 The `global` Keyword

While functions can freely *read* global variables, modifying them is treated differently. If you attempt to assign a new value to a global variable from inside a function, Python assumes you are creating a brand new local variable that happens to share the same name, leaving the original global variable entirely unaffected.

To explicitly command Python that you intend to *modify* the existing global variable, you must declare it using the `global` keyword before assignment.

```
counter = 0  # Global counter

def increment_counter():
    global counter  # Explicitly bind to the global 'counter'
    counter += 1    # Now modifies the global variable

increment_counter()
increment_counter()
print("Final counter value:", counter)  # Output: 2
```

4.7 Lambda Functions (Anonymous Functions)

For scenarios requiring a small, structurally simple, throwaway function—often intended to be passed as an argument to higher-order functions like `map()` or `filter()`—Python offers **anonymous functions**. These are functions that are not formally bound to an identifier name. They are created using the `lambda` keyword.

Syntax: `lambda arguments: expression`

Key characteristics of Lambda functions:

- They can accept any number of arguments.

- They are heavily restricted and can only consist of one single, evaluatable expression. They cannot contain complex statements or multiple lines of logic.
- They automatically and implicitly return the evaluated result of that single expression.

```python
# A standard named function
def square(x):
    return x ** 2

# The equivalent anonymous lambda function
square_lambda = lambda x: x ** 2

print(square(5))          # Output: 25
print(square_lambda(5))   # Output: 25

# Lambda accommodating multiple arguments
add_lambda = lambda x, y: x + y
print(add_lambda(10, 20)) # Output: 30
```

Listing 4.5: Using Lambda Functions

4.8 Recursive Functions

A function in Python possesses the extraordinary ability to call itself from within its own body. This advanced programming technique is formally known as **recursion**. A function that employs this technique is referred to as a **recursive function**.

Recursion is an exceptionally powerful mathematical concept used to elegantly solve problems that can be naturally decomposed into smaller, identical sub-problems. A structurally sound recursive function requires two critical, indispensable components:

1. **Base Case (The Anchor):** The defined condition under which the recursion must immediately cease. It provides the final answer for the simplest instance of the problem. Without a properly defined base case, the function would plunge into an infinite loop of self-calls, eventually triggering a fatal ‘RecursionError‘ as it exhausts system memory.
2. **Recursive Step (The Engine):** The segment of the function where it invokes itself, but importantly, passes a slightly modified, smaller input parameter. Each consecutive call must progressively inch closer to the designated base case.

```python
def factorial_recursive(n):
    # Base case: factorial of 0 or 1 is firmly 1
    if n == 0 or n == 1:
        return 1
    # Recursive step: recursively calculate (n-1)! and
   multiply by n
    else:
        return n * factorial_recursive(n - 1)

```

```python
print(factorial_recursive(5))  # Output: 120
```

Listing 4.6: Calculating Factorial Recursively

While recursion often yields highly elegant and mathematically expressive code, it requires caution. It can sometimes be less efficient regarding memory consumption and execution speed compared to traditional iterative solutions (using `for` or `while` loops) because every single recursive call stacks a new frame in system memory.

4.9 Practical Examples and Exam Questions

This section contains important practical implementations of user-defined functions, representing logic frequently asked in academic examinations. Carefully reviewing these examples will solidify your understanding of how to encapsulate and solve problems using functional programming principles.

PYQ: Define function. Explain user-defined function with suitable example. (Su'23)

Function Definition: A self-contained, formally named, and reusable block of programming code strategically designed to perform a singular, specific task. Functions are defined via the `def` keyword. They empower programmers to logically dismantle complex architectural problems into smaller, manageable, testable modules, thereby strongly promoting code reuse and broad readability.
General Syntax:

```
def function_name(parameters):
    # body of function containing logic
    return value     # optional return statement
```

Example — Function to calculate the area of a mathematical rectangle:

```python
def area_rectangle(length, width):
    """Calculate and return the geometric area of a
   rectangle."""
    area = length * width
    return area

# Function invocations
a1 = area_rectangle(5, 3)
a2 = area_rectangle(10, 7)
print("Area 1:", a1)    # Output: Area 1: 15
print("Area 2:", a2)    # Output: Area 2: 70
```

In this concrete example, `area_rectangle` stands as the user-defined function. It intelligently accepts two parameters (`length` and `width`), executes the multiplication, and seamlessly returns the computational product utilizing the `return` statement. Each subsequent call forces the body to execute with varying arguments, providing highly dynamic outputs.

PYQ: Develop Python program using user-defined function to find factorial of a number. (Su'25, W'25 OR)

Factorial Concept: The mathematical factorial of a non-negative integer n, symbolically denoted by $n!$, represents the product sequence of all positive integers less than or identically equal to n. Mathematically expressed as: $n! = n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1$. By universally accepted convention, the factorial of zero (0!) is 1.

```python
def factorial(n):
    """Calculate and return the factorial of non-negative integer n."""
    if n == 0 or n == 1:
        return 1
    result = 1
    for i in range(2, n + 1):
        result *= i
    return result

# Main procedural program
num = int(input("Enter a non-negative integer: "))
if num < 0:
    print("Error: Factorial logic is strictly undefined for negative integers.")
else:
    print(f"{num}! = {factorial(num)}")
```

Listing 4.7: Factorial computation utilizing an iterative user-defined function

Output

```
Enter a non-negative integer: 6
6! = 720
```

PYQ: Create user-defined function to print Fibonacci series 0 to N. (Su'23, W'24 OR, W'25)

Fibonacci Series Definition: A legendary mathematical sequence wherein each successive number is precisely the sum of the two numbers immediately preceding it. The sequence classically originates with 0 and 1. The numeric progression advances as: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, stretching into infinity.

```python
def fibonacci(n):
    """Iteratively print the Fibonacci series spanning up to n terms."""
    a, b = 0, 1
    print("Fibonacci series:", end=" ")
    for _ in range(n):
        print(a, end=" ")
```

```
        a, b = b, a + b    # Swift, simultaneous tuple updating mechanism
        print()

# Function execution call
terms = int(input("Enter the total number of terms desired: "))
fibonacci(terms)
```

Listing 4.8: Generation of Fibonacci series using a user-defined function

Output

```
Enter the total number of terms desired: 8
Fibonacci series: 0 1 1 2 3 5 8 13
```

PYQ: Write Python code to determine if a number is Armstrong or palindrome using user-defined function. (Su'23 OR, W'23, Su'24, W'24)

Armstrong Number Theory: A mathematically curious number that equates exactly to the mathematical sum of its constituent digits, with each individual digit raised exponentially to the power equivalent to the total count of digits in the initial number. A classic example is 153, proven an Armstrong number because $1^3 + 5^3 + 3^3 = 1 + 125 + 27 = 153$.

Palindrome Logic: A numerical or alphabetical sequence that retains identical structural reading whether traversed chronologically forwards or completely backwards. Classic examples include 121 and 1441.

```
def is_armstrong(n):
    """Boolean check to determine if n mathematically qualifies as an Armstrong number."""
    digits = str(n)
    power = len(digits)
    # Computation: sum of each discrete digit exponentially raised to 'power'
    total = sum(int(d) ** power for d in digits)
    return total == n

def is_palindrome(n):
    """Boolean check to ascertain if sequence n functions as a palindrome."""
    # Conversion to string methodology and aggressive comparative slice reversal
    return str(n) == str(n)[::-1]

# Main analytical program
num = int(input("Please enter a target number for analysis: "))

```

```python
if is_armstrong(num):
    print(num, "is structurally verified as an Armstrong number.")
else:
    print(num, "fails the Armstrong number verification.")

if is_palindrome(num):
    print(num, "is structurally verified as a Palindrome.")
else:
    print(num, "fails the Palindrome sequence verification.")
```

Listing 4.9: Dual validation checking for Armstrong and Palindrome statuses

Output

```
Please enter a target number for analysis: 153
153 is structurally verified as an Armstrong number.
153 fails the Palindrome sequence verification.
```

PYQ: Write program using a function that reverses the entered value. (Su'24 OR)

```python
def reverse_value(val):
    """Invert and reverse the sequential string representation of any incoming value."""
    # Deploying the powerful Python string slice technique [::-1] to enforce reversal
    return str(val)[::-1]

# Example application targeting a numerical integer
num = int(input("Enter an integer number to reverse: "))
print("The reversed numeric output is:", reverse_value(num))

# Example application targeting alphabetical text
text = input("Enter an alphabetical string to reverse: ")
print("The reversed textual output is:", reverse_value(text))
```

Listing 4.10: Versatile value reversal utility utilizing string slicing

PYQ: Create user-defined function which prints cube of all odd numbers between 1 to 7. (W'24)

```python
def cube_odds():
    """Iteratively scan and print the geometric cube of all odd numbers spanning between 1 and 7 (inclusive).
    """
    for i in range(1, 8):
        if i % 2 != 0:
            print(f"Mathematical Cube of {i} translates to = {i**3}")

# Triggering the function execution
cube_odds()
```

Listing 4.11: Targeted computation printing cubes exclusively for odd numbers from 1 to 7

Output

```
Mathematical Cube of 1 translates to = 1
Mathematical Cube of 3 translates to = 27
Mathematical Cube of 5 translates to = 125
Mathematical Cube of 7 translates to = 343
```

PYQ: Develop Python program using user-defined function to check if the number is prime. (Su'25 OR)

Prime Number Concept: A fundamental natural number strictly greater than 1 that mathematically possesses exactly two distinct positive divisors: 1 and itself. By definition, numbers lacking these specific characteristics are classified as composite.

```python
def is_prime(n):
    """Returns boolean True if integer n is a prime number, otherwise definitively False."""
    if n < 2:
        return False
    # Algorithmic optimization: Only check potential divisors from 2 up to the mathematical square root of n
    for i in range(2, int(n ** 0.5) + 1):
        if n % i == 0:
            return False # A divisor was successfully detected, rendering it composite
    return True # Complete iteration without divisor detection validates it as prime

```

```
num = int(input("Input an integer for prime validation:
    "))
if is_prime(num):
    print(num, "is unequivocally a Prime number.")
else:
    print(num, "is conclusively NOT a Prime number.")
```

Listing 4.12: Efficient algorithmic prime validation utilizing a user-defined function

4.10 Scope of Variables

Definition: Scope and Namespaces

The **scope** of a variable refers to the specific region of a program where that variable is accessible, readable, and modifiable. Python uses scopes to avoid name collisions and to manage memory effectively. Behind the scenes, scope is deeply interconnected with the concept of **namespaces**. A namespace is essentially a system—often implemented as a highly optimized dictionary in Python—that maps variable names (keys) to their corresponding objects in memory (values).

To truly understand how Python resolves variable names and manages data, it is absolutely essential to distinguish between a namespace and a scope. A **namespace** is simply a container of names. Its primary purpose is to ensure that names within one namespace do not conflict with the exact same names in another namespace. A **scope**, on the other hand, is the textual region of a Python program where a namespace is directly accessible without needing any special object prefixes.

During the execution of a typical Python program, there are several active namespaces, created at different times and with different lifetimes:

- **Built-in Namespace:** Contains all of Python's intrinsic built-in functions (such as `print()`, `len()`, `type()`, and `id()`) as well as built-in exceptions (like `ValueError` or `NameError`). This namespace is created precisely when the Python interpreter starts up and lives until the interpreter terminates.
- **Global Namespace:** Created for the main module or any imported module. It contains variables, functions, and classes defined at the absolute top level of the script. It lasts until the script finishes execution.
- **Local Namespace:** Created dynamically whenever a function is called. It contains the function's arguments and any locally defined variables. This namespace is highly ephemeral; it is destroyed as soon as the function returns a value or raises an unhandled exception.

4.10.1 The LEGB Rule for Scope Resolution

When you attempt to access a variable in your code, Python must figure out which specific object that variable name refers to. To resolve this, Python searches through scopes in a strict, predefined, and deterministic order known as the **LEGB rule**.

The LEGB acronym stands for **L**ocal, **E**nclosing, **G**lobal, and **B**uilt-in. Python sequentially searches these four levels:

- **L - Local Scope:** Python always starts by looking in the local scope, which refers to the namespace of the current function or method being executed. If the target variable is found here, the search stops immediately, and that variable is used.
- **E - Enclosing Scope:** If the variable is not found locally, Python checks the enclosing scope. This occurs only in the context of nested functions (a function defined intimately inside another function). Python will meticulously search the local scopes of any enclosing functions from the innermost to the outermost.

- **G - Global Scope:** If the variable is still completely absent from both local and enclosing scopes, Python looks in the global scope. This represents the top-most level of the executing script or module.
- **B - Built-in Scope:** If all else fails, Python checks the built-in scope. If the variable name isn't found even among the core built-ins, Python finally throws its hands up and raises a `NameError`, crashing the program if not caught.

4.10.2 Local Variables and Local Scope

A **local variable** is a variable that is defined *inside* the body of a function. It exists exclusively within that function's scope, meaning it is conceptually fenced off from the rest of the program.

Core Characteristics of Local Variables:

1. **Accessibility limits:** They cannot be accessed, read, or modified from outside the function. The outside world is completely unaware of their existence.
2. **Lifetime and Memory:** They are dynamically created when the function is invoked. To optimize memory, they are completely destroyed (garbage collected) the moment the function finishes its execution or returns a value.
3. **Total Isolation:** Two different functions can independently have local variables with the exact same name without any risk of conflict. Because each function maintains its own independent local namespace, `var_x` in function A is entirely distinct from `var_x` in function B.

```
def calculate_area(length, width):
    # 'area', 'length', and 'width' are strictly LOCAL variables
    area = length * width
    print("Inside function: Area =", area)
    return area

calculate_area(10, 5)

# Trying to access 'area' here in the global scope would cause a NameError
# because 'area' is out of its scope and its memory has been reclaimed.
# print(area)    <- ERROR: name 'area' is not defined
```

Listing 4.13: Local variable demonstration

Output

```
Inside function: Area = 50
```

Every time `calculate_area()` is called, a brand new local namespace is generated. If we call it a hundred times, a hundred separate `area` variables are sequentially created and subsequently destroyed, maintaining an incredibly efficient memory footprint.

4.10.3 Global Variables and Global Scope

A **global variable** is defined *outside* of all functions, typically at the very top of a script or module. Because they reside in the overarching global namespace, they are globally accessible from anywhere within that file, including deeply nested inside functions.

Core Characteristics of Global Variables:

1. **Universal Accessibility:** They can be read from any scope (whether local or enclosing) within the same module without any special syntax.
2. **Persistent Lifetime:** They are created when the module is first loaded or run and persist securely in memory until the Python script completely terminates.
3. **State Sharing capability:** They can be used to share state, configuration, or critical data across multiple separate functions that need to communicate.

While reading a global variable inside a function is straightforward and automatic, *modifying* it requires explicit, purposeful instruction.

Reading Global Variables

If you only need to read the value of a global variable, you can simply use its name directly. Python's LEGB rule will fail to find it in the Local and Enclosing scopes, successfully and automatically falling back to the Global scope.

```
app_name = "DataAnalyzer Pro"  # Global variable

def show_welcome_message():
    # Reading (not modifying) global variable -- no 'global'
    keyword needed
    print(f"Welcome to {app_name}!")

show_welcome_message()
```

Listing 4.14: Reading a global variable (no modification)

Output

```
Welcome to DataAnalyzer Pro!
```

Modifying Global Variables: The `global` Keyword

When you attempt to assign a new value to a variable inside a function (e.g., using the `=` operator), Python behaves conservatively: it automatically assumes you are deliberately creating a *new local variable* by default. This is a safety mechanism to prevent functions from accidentally corrupting global state.

If your true intent is to modify an existing global variable rather than creating a local shadow of it, you must explicitly declare the variable using the `global` keyword before you assign to it.

```
# Global variable tracking user logins
login_count = 0

```

```
def track_login():
    global login_count          # Declare explicit intent to modify the global variable
    login_count += 1            # Modifies the global variable directly
    print(f"User logged in. Total logins: {login_count}")

track_login()
track_login()
print("Final global count:", login_count)
```

Listing 4.15: Modifying a global variable using the `global` keyword

Output

```
User logged in. Total logins: 1
User logged in. Total logins: 2
Final global count: 2
```

The UnboundLocalError Trap

A highly common and frustrating pitfall occurs when programmers attempt to logically read and then modify a global variable without using the `global` keyword. It is vital to understand that Python parses functions completely before executing them. If the interpreter sees an assignment to a variable anywhere in the function body, it definitively marks that variable as local for the entire function context.

```
total_score = 50

def add_bonus():
    # Error! Python statically sees the assignment below during parsing
    # and definitively treats 'total_score' as a local variable.
    # However, at runtime, it tries to read the local 'total_score'
    # before it has been assigned any value.
    total_score = total_score + 10
    print(total_score)

# add_bonus()   <- This raises UnboundLocalError
```

Listing 4.16: Without global keyword – UnboundLocalError

Executing the above code raises an `UnboundLocalError: local variable 'total_score' referenced before assignment`. The infallible solution is to add `global total_score` as the very first line inside the function body.

4.10.4 Enclosing Scope and the `nonlocal` Keyword

Python boasts robust support for nested functions, which are simply functions defined entirely inside other functions. This architectural pattern introduces the **Enclosing**

scope. The inner function is permitted to seamlessly access variables from the outer (enclosing) function.

Similar to how the `global` keyword enables the modification of global variables, Python provides the `nonlocal` keyword. This allows an inner nested function to explicitly modify a variable that was defined in its enclosing (but fundamentally non-global) scope.

```
def outer_function():
    count = 0  # Exists in the enclosing scope relative to inner_function

    def inner_function():
        nonlocal count  # Declare explicit intent to modify the enclosing variable
        count += 1
        print("Inner count:", count)

    inner_function()
    inner_function()
    print("Outer count final state:", count)

outer_function()
```

Listing 4.17: Demonstrating enclosing scope and the `nonlocal` keyword

Output

```
Inner count: 1
Inner count: 2
Outer count final state: 2
```

Without the crucial `nonlocal` keyword, `inner_function` would have defensively created its own independent local `count` variable or thrown an `UnboundLocalError`. Crucially, the `nonlocal` keyword specifically searches enclosing scopes moving outward but *never* searches the global scope. If it reaches the global scope without finding the variable, it raises a `SyntaxError`.

4.10.5 Built-in Scope

The Built-in scope is the absolute widest and outermost scope in Python's ecosystem. It natively contains Python's intrinsic functions and core exceptions (like `int`, `str`, `list`, `print`, `TypeError`, `ValueError`, etc.). You can programmatically view all built-in names by inspecting the `__builtins__` module or using `dir(__builtins__)`.

It is perilously possible to accidentally overwrite (or shadow) built-in functions by assigning values to their names, which is considered highly dangerous and terrible practice.

```
def demonstrate_shadowing():
    # SHADOWING: We accidentally name our variable 'list', overriding the built-in
    list = [1, 2, 3]

```

```
    # If we later try to critically use the built-in list()
    constructor, it fails:
    # my_items = list("abc")  <- TypeError: 'list' object is
    not callable

demonstrate_shadowing()
```

Listing 4.18: Shadowing a built-in function (Extremely Bad Practice)

Always meticulously avoid using ubiquitous names like `list`, `str`, `dict`, `min`, `id`, or `max` as variable names to proactively prevent shadowing the Built-in scope and breaking standard language functionality.

4.10.6 Variable Shadowing (Name Masking)

Shadowing happens when a variable defined in a narrower inner scope is given the exact same name as a variable in a broader outer scope. The LEGB rule explicitly dictates that Python will enthusiastically use the innermost variable it finds first, effectively "shadowing" or masking the outer variable from that point onward within that localized scope.

```
x = 100  # Global variable

def shadow_demo():
    x = 50  # Local variable aggressively shadows the global
    variable
    print("Inside function, local x =", x)

shadow_demo()
print("Outside function, global x remains =", x)
```

Listing 4.19: Variable shadowing demonstration

Output

```
Inside function, local x = 50
Outside function, global x remains = 100
```

In this scenario, the localized `x` intercepts the LEGB lookup inside the function, leaving the globally scoped `x` completely untouched, untainted, and temporarily hidden from view.

4.10.7 Lifetime vs. Scope

While intimately related and often confused, **scope** and **lifetime** are distinctly separate programming concepts:

- **Scope:** An inherently spatial and textual concept. It fundamentally determines *where* in your written source code a variable name can be successfully accessed and recognized.

- **Lifetime:** A strictly temporal concept. It fundamentally determines *how long* the variable's physical value resides actively in the computer's RAM during program execution.

A global variable has a vast scope covering the entire module and a prolonged lifetime equal to the program's entire execution runtime. Conversely, a local variable has a tightly constrained scope limited to its function block and an ephemeral lifetime that abruptly begins when the function is called and abruptly ends when it returns.

4.10.8 Local vs. Global vs. Enclosing: Comprehensive Comparison

Property	Local Scope	Enclosing Scope	Global Scope
Where explicitly defined	Inside the body of the innermost executing function.	Inside the body of an outer (enclosing) parent function.	Outside all functions and classes entirely (at the module level).
Where successfully accessible	Only within the specific boundaries of that defining function.	From the outer function and gracefully downwards into any nested inner functions.	Universally anywhere in the entire module, across all functions.
Memory Lifetime	Highly temporary. Created dynamically on call, destroyed mercilessly on return.	Exists safely as long as the outer function is running (or permanently extended via closures).	Highly persistent. Exists continuously until the Python script terminates completely.
Modification Keyword	N/A (can be freely modified directly).	Demands the `nonlocal` keyword from the inner nested function.	Demands the `global` keyword from inside any localized function.
Memory Efficiency	Exceptionally efficient and clean.	Moderately efficient, depending on closure usage.	Least efficient (memory is permanently held globally).

Table 4.1: Detailed Architectural Comparison of Variable Scopes in Python

4.10.9 Architectural Best Practices for Variable Scope

- **Ruthlessly Minimize Global Variables:** Excessive reliance on global variables rapidly makes code difficult to logically trace, hopelessly hard to debug, and nearly

impossible to unit test. Functions should ideally operate as strictly self-contained black boxes that take explicit inputs as parameters and return explicit outputs.

- **Heavily Prefer Local Variables:** Local variables are significantly faster for the Python interpreter to access and guarantee that you will not trigger unintended side effects in widely separated parts of your program.
- **Leverage Global Constants:** If you undeniably must use global variables, try rigorously to restrict them to constants (values that are assigned once and never ever change). By universally accepted convention, global constants are named using exclusively UPPERCASE letters (e.g., `PI = 3.14159`, `MAX_DATABASE_CONNECTIONS = 100`).

4.10.10 Review Questions and Previous Year Questions

PYQ: Explain local variable with example. (Su'25)

Local Variable: A variable logically declared and assigned directly inside a function body is formally known as a local variable. Its operational scope is strictly and unconditionally confined to that specific function, meaning it absolutely cannot be referenced, read, or altered by any arbitrary code existing outside of the function's block. Furthermore, its memory lifetime is deliberately temporary; it is freshly created whenever the function is invoked and is automatically and completely destroyed by Python's background garbage collector the exact moment the function execution successfully completes.

Example:

```
1 def compute_area(radius):
2     pi = 3.14159                    # 'pi' is strictly
    LOCAL to compute_area
3     area = pi * radius * radius   # 'area' is also
    strictly LOCAL
4     print(f"Calculated Area = {area:.2f}")
5
6 compute_area(5)
7 compute_area(10)
8 # Trying to print(pi) here globally would immediately
    result in a fatal NameError.
```

Output

```
Calculated Area = 78.54
Calculated Area = 314.16
```

Each independent, successive call to `compute_area()` flawlessly allocates its own completely separate local memory space for `pi` and `area`. They absolutely never interfere with each other.

PYQ: Explain global variable with example. (Su'25 OR)

Global Variable: A variable logically defined at the very top-most level of a script or module, entirely outside the bounds of any functions or classes, is definitively called a global variable. It inherently resides in the master global namespace and permanently maintains a global scope, meaning it can be effortlessly read from any local or enclosing scope anywhere within that specific module. Its lifetime robustly spans the entire runtime execution of the program. However, to actively update or mutate a global variable's value from inside the restrictive confines of a function, the programmer is forced to explicitly declare their intentions using the `global` keyword.

Example:

```
# Master global variable
DISCOUNT_RATE = 0.10    # Default 10% global discount

def calculate_price(original_price):
    # Reading global variable implicitly - no 'global' keyword is necessary
    return original_price * (1 - DISCOUNT_RATE)

def update_discount(new_rate):
    # Modifying global variable explicitly - keyword is ABSOLUTELY REQUIRED
    global DISCOUNT_RATE
    DISCOUNT_RATE = new_rate
    print(f"System discount permanently updated to {DISCOUNT_RATE * 100}%")

print("Original Price Check:", calculate_price(1000))
update_discount(0.20)
print("New Discounted Price Check:", calculate_price(1000))
```

Output

```
Original Price Check: 900.0
System discount permanently updated to 20.0%
New Discounted Price Check: 800.0
```

PYQ: Explain local and global variables with examples; explain the concept of scope. (Su'23 OR, W'23 OR, W'25)

Concept of Scope: Scope rigorously defines the precise textual region of a Python program where a particular variable name is legally accessible and usable. It inherently dictates both the variable's visibility to other code blocks and its operational lifetime in memory. Python intelligently resolves all variable names using the foundational LEGB rule, systematically searching the Local scope first,

then dynamically expanding to the Enclosing scope, then the Global scope, and finally bottoming out at the Built-in scope.

Local Variables: Variables natively defined within a function block. They are highly temporary, incredibly efficient, and perfectly isolated to that specific function.

```
def greet_new_user():
    msg = "Welcome to the secure system!"    # Pure local variable
    print(msg)

greet_new_user()
# print(msg) -> This rogue attempt would cause a fatal NameError
```

Global Variables: Variables defined expansively outside all functions. They are deeply persistent and globally visible throughout the entire module's execution. Any attempted modification occurring inside a function explicitly mandates the usage of the `global` keyword.

```
user_security_role = "Guest"          # Master global variable

def display_current_role():
    print("Current system role:", user_security_role)
    # Implicitly reading global

def upgrade_security_role():
    global user_security_role             # Explicitly declaring intent to modify
    user_security_role = "Administrator"

display_current_role()                    # Output: Current system role: Guest
upgrade_security_role()
display_current_role()                    # Output: Current system role: Administrator
```

Definitive Scope Summary:

- **Local Scope:** Exists only inside the defining function. Characterized by a highly temporary memory lifetime and zero cross-function interference.
- **Global Scope:** Exists everywhere globally in the module. Characterized by a persistent memory lifetime. Rigorously requires the `global` keyword to be modified locally.

PYQ: Write Python program that explains the scope of a variable. (Su'24)

A highly comprehensive, multi-tiered program deliberately designed to effectively demonstrate Local, Enclosing, and Global scopes interacting simultaneously:

```
# Top-level Global scope variable
scope_var = "I am the original GLOBAL variable"

def outer_function():
    # Enclosing scope variable (conceptually local to outer_function)
    # This immediately shadows the global variable
    scope_var = "I am the ENCLOSING variable"

    def inner_function():
        # Innermost Local scope variable
        # This immediately shadows the enclosing variable
        scope_var = "I am the innermost LOCAL variable"

        # When inner_function attempts to utilize scope_var,
        # it invariably finds the Local one first (dictated by the LEGB rule)
        print("Executing inside inner_function :", scope_var)

    inner_function()

    # outer_function invariably finds the Enclosing one,
    # completely ignorant of the inner local variable
    print("Executing inside outer_function :", scope_var)

outer_function()

# The global module level invariably finds the original Global one
print("Executing at top global level   :", scope_var)
```

Listing 4.20: Complete and comprehensive Scope demonstration program

Output

```
Executing inside inner_function : I am the innermost LOCAL variable
Executing inside outer_function : I am the ENCLOSING variable
Executing at top global level   : I am the original GLOBAL variable
```

Comprehensive Explanation: Although the identical variable name `scope_var` is repeatedly used three distinct times, each individual assignment securely happens

in a completely different, mathematically isolated scope namespace. Python's rigorous LEGB rule ensures that there is absolutely no destructive conflict or state corruption. The innermost (most localized) matching variable is perpetually and predictably prioritized during any name resolution sequence.

4.11 Python Standard Library

Python's standard library is described as "batteries included" — it ships with an enormous collection of modules covering almost every common programming need. No installation is required; they are available as soon as Python is installed.

4.12 Built-in Functions

Built-in functions are always available in Python — no import statement is needed. The most frequently used built-in functions are:

Function	Description	Example
`print()`	Displays output to the screen	`print("Hello")`
`input()`	Reads a line of text from the user	`name = input("Enter name: ")`
`len()`	Returns the length of a sequence	`len("Python")` → 6
`type()`	Returns the data type of an object	`type(3.14)` → float
`int()`	Converts a value to integer	`int("42")` → 42
`float()`	Converts a value to float	`float("3.14")` → 3.14
`str()`	Converts a value to string	`str(100)` → "100"
`abs()`	Returns absolute value	`abs(-15)` → 15
`max()`	Returns the maximum value	`max(3, 9, 1)` → 9
`min()`	Returns the minimum value	`min(3, 9, 1)` → 1
`sum()`	Returns sum of all items in iterable	`sum([1,2,3,4])` → 10
`pow()`	Returns x^y	`pow(2, 8)` → 256
`divmod()`	Returns (quotient, remainder)	`divmod(17, 5)` → (3, 2)
`range()`	Generates a sequence of integers	`range(1, 6)` → 1,2,3,4,5
`round()`	Rounds a number	`round(3.7)` → 4
`sorted()`	Returns sorted list of any iterable	`sorted([3,1,2])` → [1,2,3]
`reversed()`	Returns reversed iterator	`list(reversed([1,2,3]))`

Table 4.2: Commonly Used Python Built-in Functions

```python
# abs, max, min, sum, pow, divmod
numbers = [4, -7, 2, -1, 9, 3]

print("abs(-7)       =", abs(-7))            # 7
print("max(numbers) =", max(numbers))        # 9
print("min(numbers) =", min(numbers))        # -7
print("sum(numbers) =", sum(numbers))        # 10
print("pow(2, 10)    =", pow(2, 10))         # 1024
print("divmod(17,5) =", divmod(17, 5))       # (3, 2)

# input and print
name = input("Enter your name: ")
print("Welcome,", name + "!")
```

Listing 4.21: Built-in functions demonstration

PYQ: Describe built-in functions max(), input(), pow() with example. (W'24)

1. `max()` — Returns the largest value:

```python
print(max(3, 7, 1, 9, 5))          # 9
print(max([10, 20, 5, 15]))        # 20
print(max("apple", "banana"))      # banana (
    lexicographic)
```

2. `input()` — Reads user input as a string:

```python
name = input("Enter your name: ")      # returns string
age = int(input("Enter your age: "))   # convert to int
print(f"Hello {name}, you are {age} years old.")
```

3. `pow()` — Computes power:

```python
print(pow(2, 10))     # 1024   (2^10)
print(pow(3, 4))      # 81     (3^4)
print(2 ** 10)        # same as pow(2, 10)
```

PYQ: Explain usage of built-in functions: print(), min(), sum(), input(). (Su'23 OR)

1. `print()` — Output function: Displays one or more values. Supports `sep` and `end` arguments.

```python
print("Hello", "World", sep="-")    # Hello-World
print("Count:", 1, 2, 3)            # Count: 1 2 3
```

2. `min()` — Minimum value:

```python
print(min(5, 2, 8, 1))              # 1
```

```
scores = [85, 92, 78, 90]
print(min(scores))                    # 78
```

3. `sum()` — **Sum of iterable:**

```
print(sum([1, 2, 3, 4, 5]))           # 15
print(sum(range(1, 101)))             # 5050
```

4. `input()` — **Read user input (always returns string):**

```
num = int(input("Enter a number: "))    # must convert
    for arithmetic
print("Square:", num ** 2)
```

4.13 Python Modules

Definition: Module

A **module** is a file containing Python code — definitions of functions, classes, and variables, plus runnable code. Modules allow organising related code into separate files for reusability and clarity. To use a module, you `import` it.

Import syntax:

```
import module_name              # import full module
from module_name import func    # import specific function
import module_name as alias     # import with alias
```

4.13.1 The `math` Module

The `math` module provides access to mathematical functions and constants defined by the C standard.

Function/Constant	Description	Example
`math.pi`	Value of π (3.14159...)	`math.pi` → 3.14159
`math.e`	Value of e (2.71828...)	`math.e` → 2.71828
`math.sqrt(x)`	Square root of x	`math.sqrt(25)` → 5.0
`math.pow(x, y)`	x^y as float	`math.pow(2, 8)` → 256.0
`math.floor(x)`	Largest integer $\leq x$	`math.floor(4.7)` → 4
`math.ceil(x)`	Smallest integer $\geq x$	`math.ceil(4.2)` → 5
`math.factorial(n)`	$n!$	`math.factorial(5)` → 120
`math.log(x)`	Natural log $\ln(x)$	`math.log(math.e)` → 1.0
`math.log10(x)`	Base-10 logarithm	`math.log10(1000)` → 3.0
`math.sin(x)`	Sine of x radians	`math.sin(math.pi/2)` → 1.0
`math.cos(x)`	Cosine of x radians	`math.cos(0)` → 1.0
`math.fabs(x)`	Absolute value as float	`math.fabs(-5)` → 5.0

Table 4.3: Commonly Used math Module Functions

```
import math

print("pi            =", math.pi)            #
   3.141592653589793
print("sqrt(144)     =", math.sqrt(144))      # 12.0
print("ceil(4.3)     =", math.ceil(4.3))      # 5
print("floor(4.9)    =", math.floor(4.9))    # 4
print("factorial(6)  =", math.factorial(6)) # 720
print("log10(1000)   =", math.log10(1000))  # 3.0
print("sin°(90)      =", math.sin(math.pi / 2))  # 1.0
```

Listing 4.22: math module demonstration

4.13.2 The `random` Module

The `random` module implements pseudo-random number generators and random selection functions.

Function	Description	Example
`random.random()`	Returns float in $[0.0, 1.0)$	0.4857921...
`random.randint(a, b)`	Returns random integer in $[a, b]$ inclusive	`random.randint(1, 6)` → simulates dice
`random.uniform(a, b)`	Returns random float in $[a, b]$	`random.uniform(1.0, 10.0)`
`random.choice(seq)`	Picks a random element from a sequence	`random.choice(["a", "b","c"])`
`random.shuffle(lst)`	Shuffles a list in-place	modifies list
`random.sample(seq, k)`	Returns k unique random elements from sequence	`random.sample(range(100), 5)`

Table 4.4: random Module Functions

4.13.3 The `statistics` Module

The `statistics` module provides functions for mathematical statistics of numeric data.

Function	Description
`statistics.mean(data)`	Arithmetic mean (average) of data
`statistics.median(data)`	Middle value of sorted data
`statistics.mode(data)`	Most common value (raises error if no single mode)
`statistics.stdev(data)`	Sample standard deviation
`statistics.variance(data)`	Sample variance

Table 4.5: statistics Module Functions

PYQ: Explain math module and random module with example. (Su'23, Su'24)

math Module: Provides mathematical functions. Must be imported with `import math`.

```python
import math
# Circle area
r = 7
area = math.pi * r ** 2
print(f"Area of circle (r=7) = {area:.2f}")    # 153.94

# Hypotenuse
a, b = 3, 4
hyp = math.sqrt(a**2 + b**2)
print(f"Hypotenuse = {hyp}")                    # 5.0
```

random Module: Generates pseudo-random numbers. Must be imported with `import random`.

```python
import random
# Roll a die
print("Dice roll:", random.randint(1, 6))           # e.g. 4

# Pick from list
colours = ["red", "blue", "green", "yellow"]
print("Random colour:", random.choice(colours))     # e.g. green

# Random float
print("Random float:", random.uniform(0, 100))      # e.g. 47.23
```

PYQ: List functions of math module and develop a program demonstrating math module functions. (Su'25)

Key math module functions: `sqrt()`, `pow()`, `floor()`, `ceil()`, `factorial()`, `log()`, `log10()`, `sin()`, `cos()`, `tan()`, `pi`, `e`, `fabs()`, `gcd()`.

```python
import math

# Constants
print("pi =", math.pi)              # 3.141592653589793
print("e  =", math.e)               # 2.718281828459045

# Rounding
print("floor(3.9)  =", math.floor(3.9))     # 3
print("ceil(3.1)   =", math.ceil(3.1))      # 4
```

```
10
11 # Powers and roots
12 print("sqrt(81)     =", math.sqrt(81))        # 9.0
13 print("pow(2,10)    =", math.pow(2, 10))      # 1024.0
14
15 # Factorial and log
16 print("6!           =", math.factorial(6))   # 720
17 print("log10(1000) =", math.log10(1000))    # 3.0
18
19 # Trigonometry (radians)
20 angle = math.radians(60)    # convert 60 degrees to
       radians
21 print("sin°(60)     =", round(math.sin(angle), 4))   #
       0.866
22 print("cos°(60)     =", round(math.cos(angle), 4))   # 0.5
```

Listing 4.23: math module – comprehensive program

PYQ: Explain Random module in Python. (W'25)

random Module: A Python standard library module that provides functions for generating pseudo-random numbers and performing random operations.
Import: `import random`

```
1  import random
2
3  # 1. random()  --  float between 0.0 and 1.0
4  print("random():", random.random())           # e.g.
       0.7438
5
6  # 2. randint(a, b)  --  integer between a and b (
       inclusive)
7  print("randint(1,10):", random.randint(1, 10))  # e.g. 7
8
9  # 3. uniform(a, b)  --  float between a and b
10 print("uniform(1,5):", random.uniform(1, 5))    # e.g.
       3.24
11
12 # 4. choice(seq)  --  random element from sequence
13 fruits = ["apple", "mango", "banana"]
14 print("choice():", random.choice(fruits))       # e.g.
       mango
15
16 # 5. shuffle(lst)  --  shuffle list in-place
17 nums = [1, 2, 3, 4, 5]
18 random.shuffle(nums)
19 print("After shuffle:", nums)                   # e.g.
       [3,1,5,2,4]
20
```

```python
# 6. sample(seq, k)  --  k unique random items
lucky = random.sample(range(1, 50), 6)
print("Lottery numbers:", sorted(lucky))        # 6 unique numbers
```

Listing 4.24: random module demonstration

PYQ: Explain statistics module in Python. (W'25 OR)

statistics Module: Provides statistical functions. Available in Python 3.4+.

```python
import statistics

data = [85, 90, 78, 92, 88, 76, 90, 84, 91, 87]

print("Mean      :", statistics.mean(data))        # 86.1
print("Median    :", statistics.median(data))      # 87.5
print("Mode      :", statistics.mode(data))        # 90
print("StdDev    :", round(statistics.stdev(data), 2))
   # 5.49
print("Variance  :", round(statistics.variance(data), 2))
    # 30.1
```

Listing 4.25: statistics module demonstration

Key functions:

- `mean(data)` — arithmetic average.
- `median(data)` — middle value when sorted.
- `mode(data)` — most frequently occurring value.
- `stdev(data)` — sample standard deviation (spread of data).
- `variance(data)` — sample variance (σ^2).

PYQ: List functions of statistics module and develop a program demonstrating statistics module functions. (Su'25 OR)

statistics module functions: `mean()`, `median()`, `median\_low()`, `median\_high()`, `mode()`, `multimode()`, `stdev()`, `variance()`, `pstdev()`, `pvariance()`, `harmonic\_mean()`, `geometric\_mean()`.

```python
import statistics

marks = [72, 85, 90, 72, 88, 95, 72, 80]

print("Data:", marks)
```

```
print("Mean         :", statistics.mean(marks))
    # 81.75
print("Median       :", statistics.median(marks))
    # 82.5
print("Median Low   :", statistics.median_low(marks))
    # 80
print("Median High  :", statistics.median_high(marks))
    # 85
print("Mode         :", statistics.mode(marks))
    # 72
print("Multimode    :", statistics.multimode(marks))
    # [72]
print("Std Dev      :", round(statistics.stdev(marks), 2)
    )   # 8.33
print("Variance     :", round(statistics.variance(marks)
    ,2)) # 69.36
```

Listing 4.26: Complete statistics program

CHAPTER 5

Strings, Lists, Sets, Tuples and Dictionaries

5.1 Strings

Definition: String

A **string** in Python is an ordered, *immutable* sequence of Unicode characters. As one of Python's most fundamental and widely-used built-in data types, strings are essential for representing textual data. From reading user input and formatting text for display to processing files and handling web data, strings form the backbone of text processing in Python.
A string can contain any combination of letters, digits, whitespace, and special characters. Because they are sequences, they share many operations with other sequence types like lists and tuples, such as indexing, slicing, and iteration. However, their defining characteristic is immutability—once a string is created in memory, its contents cannot be altered. Any operation that appears to modify a string actually creates and returns an entirely new string.

5.1.1 Creating Strings

In Python, strings are incredibly flexible to create. They are written by enclosing sequences of characters within quotes. Python provides multiple ways to define strings to accommodate different needs, such as embedding quotes within the text or spanning multiple lines.

- **Single quotes:** `'Hello World'`. Ideal for short strings.
- **Double quotes:** `"Python Programming"`. Extremely useful when your string contains a single quote or an apostrophe, avoiding the need for escape characters.
- **Triple quotes:** `'''Multi-line string'''` or `"""Another multi-line string"""`. These are used for strings that span multiple lines. They are commonly used for docstrings (documentation strings) to document functions, classes, and modules, or for complex text blocks like SQL queries or HTML templates.

```python
# Standard single and double quotes
name = 'Alice'
message = "It's a beautiful day!" # Double quotes allow
    apostrophe

# Multi-line string using triple quotes
poem = """Roses are red,
Violets are blue,
Python is awesome,
And so are you!"""

# Escaping characters
escaped_str = 'It\'s easy to escape quotes.'
newline_str = "First Line\nSecond Line"
tab_str = "Column1\tColumn2"
```

Listing 5.1: Creating strings in various ways

Raw Strings

Python also supports **raw strings**, denoted by prefixing the string literal with an `r` or `R`. In a raw string, escape sequences (like `\n` or `\t`) are not processed but are instead treated as literal characters. This is incredibly useful for regular expressions and Windows file paths.

```python
# Without raw string, \n is treated as a newline
path = "C:\new_folder\test.txt"
print(path) # Will print C: then newline then ew_folder...

# With raw string
raw_path = r"C:\new_folder\test.txt"
print(raw_path) # Prints exactly C:\new_folder\test.txt
```

Listing 5.2: Raw strings

5.1.2 String Indexing and Access

Since a string is a sequence of characters, every individual character has an assigned position, known as its **index**. Python uses **zero-based indexing**, meaning the first character is at index 0.

One of Python's unique and powerful features is **negative indexing**, which allows you to count from the end of the string backwards. The last character is always at index `-1`, the second to last is at `-2`, and so on.

For a string of length n:

- **Valid positive indices:** $0, 1, 2, \ldots, n-1$
- **Valid negative indices:** $-1, -2, \ldots, -n$

Attempting to access an index outside these valid ranges will raise an `IndexError`.

```
word = "PYTHON"
# Character: P   Y   T   H   O   N
# Pos Index: 0   1   2   3   4   5
# Neg Index:-6  -5  -4  -3  -2  -1

print(word[0])     # P (First character)
print(word[3])     # H (Fourth character)
print(word[-1])    # N (Last character)
print(word[-6])    # P (First character via negative indexing)

# Accessing an invalid index:
# print(word[10]) # This raises IndexError: string index out
    of range
```

Listing 5.3: Positive and negative indexing

5.1.3 String Slicing

While indexing allows you to extract a single character, **slicing** lets you extract a substring (a smaller piece of the string). Slicing uses the syntax `s[start:stop:step]`.

- **`start`:** The starting index (inclusive). If omitted, defaults to 0.
- **`stop`:** The ending index (exclusive). The slice goes up to, but does not include, this index. If omitted, defaults to the length of the string.
- **`step`:** The stride or jump between characters. If omitted, defaults to 1. A negative step iterates backwards.

Slicing is a safe operation in Python. Unlike indexing, providing a `start` or `stop` index that is completely out of bounds will not raise an `IndexError`; it will simply return whatever characters fall within the valid overlapping range, or an empty string.

```
s = "Python Programming"

# Basic Slicing
print(s[0:6])        # 'Python' (characters from index 0 to 5)
print(s[7:18])       # 'Programming' (index 7 to 17)

# Omitting Start or Stop
print(s[:6])         # 'Python' (Starts from the beginning)
print(s[7:])         # 'Programming' (Goes up to the end)
print(s[:])          # 'Python Programming' (Creates a shallow
    copy)

# Using Steps
print(s[::2])        # 'Pto rgamn' (Every second character)
print(s[1::2])       # 'yhnPormig' (Every second character
   starting at 1)

# Negative Steps for Reversing
```

```
print(s[::-1])       # 'gnimmargorP nohtyP' (Reverses the entire string)
print(s[5::-1])      # 'nohtyP' (Reverses the word 'Python')
```

Listing 5.4: String slicing comprehensively

5.1.4 String Operations

Strings support several fundamental operations using standard Python operators, making text manipulation intuitive and straightforward.

Operation	Description	Example
Concatenation (+)	Joins two or more strings together to form a new string.	`"Hi" + " " + "All"` → `"Hi All"`
Repetition (*)	Repeats a string a specified number of times.	`"ab" * 3` → `"ababab"`
Membership (in)	Evaluates to `True` if a substring exists within the string.	`"Py" in "Python"` → True
Non-membership (not in)	Evaluates to `True` if a substring does NOT exist within the string.	`"Java" not in "Python"` → True
Comparison (==, !=, <, >)	Lexicographical (alphabetical) comparison based on ASCII/Unicode values.	`"apple" < "banana"` → True `"A" < "a"` → True
Length (len())	Returns the total number of characters in the string.	`len("Hello World")` → 11

Table 5.1: Core String Operations

Immutability in Detail

It is crucial to understand that strings are immutable. You cannot change a character in-place. If you try to assign a new character to an existing index, Python will throw a `TypeError`.

```
greeting = "Hello"
# greeting[0] = "J"  # TypeError: 'str' object does not support item assignment

# To "change" a string, you must create a new one:
greeting = "J" + greeting[1:]
print(greeting)      # Jello
```

Listing 5.5: String Immutability

5.1.5 String Formatting

Formatting allows you to inject variables and dynamically construct strings. Python has evolved to offer several ways to format strings.

1. % Formatting (Legacy)

Borrowing from C's `printf`, Python uses the modulo operator for string interpolation.

```
name = "John"
age = 25
print("My name is %s and I am %d years old." % (name, age))
```

2. The `format()` Method

Introduced in Python 2.6, this method uses curly braces `\{\}` as placeholders.

```
name = "Alice"
score = 98.5
print("Student {} scored {:.2f} marks.".format(name, score))
# Support for positional and keyword arguments:
print("{1} is {0} years old.".format(30, "Bob"))
print("Welcome, {user}!".format(user="Admin"))
```

3. f-Strings (Formatted String Literals)

Introduced in Python 3.6, f-strings are the most modern, readable, and performant way to format strings. By prefixing a string with `f` or `F`, you can embed Python expressions directly inside curly braces.

```
item = "Apples"
price = 1.50
quantity = 4
print(f"Total cost for {quantity} {item}: ${price * quantity
    :.2f}")
# Output: Total cost for 4 Apples: $6.00
```

5.1.6 Comprehensive String Methods

Strings come with a rich set of built-in methods. Since strings are immutable, these methods never modify the original string; instead, they return a new string with the modifications applied.

Case Conversion Methods

These methods are useful for text normalization, especially before comparing inputs.

- **`upper()`:** Converts all lowercase characters to uppercase.
- **`lower()`:** Converts all uppercase characters to lowercase.

- **`capitalize()`:** Capitalizes only the first character of the string, making the rest lowercase.
- **`title()`:** Capitalizes the first letter of every word.
- **`swapcase()`:** Swaps uppercase to lowercase and vice versa.

```
text = "python PROGRAMMING is Fun"
print(text.upper())       # PYTHON PROGRAMMING IS FUN
print(text.lower())       # python programming is fun
print(text.capitalize()) # Python programming is fun
print(text.title())       # Python Programming Is Fun
print(text.swapcase())    # PYTHON programming IS fUN
```

Listing 5.6: Case conversions

Search and Replacement Methods

- **`find(sub[, start[, end]])`:** Returns the lowest index of the substring `sub`. Returns `-1` if not found.
- **`index(sub[, start[, end]])`:** Like `find()`, but raises a `ValueError` if the substring is not found.
- **`count(sub[, start[, end]])`:** Returns the number of non-overlapping occurrences of `sub`.
- **`replace(old, new[, count])`:** Replaces occurrences of `old` with `new`. If `count` is given, only the first `count` occurrences are replaced.

```
sentence = "A dog, a cat, and another dog."
print(sentence.find("dog"))       # 2
print(sentence.rfind("dog"))      # 26 (searches from the
    right)
print(sentence.count("dog"))      # 2

# Replace 'dog' with 'bird'
print(sentence.replace("dog", "bird"))
# Output: A bird, a cat, and another bird.
```

Listing 5.7: Search and replace

Trimming and Stripping

Used to clean up unwanted whitespace or specific characters from the edges of a string.

- **`strip([chars])`:** Removes characters (defaults to whitespace) from both ends.
- **`lstrip([chars])`:** Removes characters from the left end.
- **`rstrip([chars])`:** Removes characters from the right end.

```python
user_input = "   \t  Hello World! \n  "
print(f"[{user_input.strip()}]")  # [Hello World!]

code = "0000456000"
print(code.strip("0"))            # 456
print(code.lstrip("0"))           # 456000
```

Listing 5.8: Stripping whitespace and characters

Splitting and Joining

These methods bridge the gap between strings and lists.

- **`split([sep[, maxsplit]])`:** Splits the string into a list of words using `sep` as the delimiter. If `sep` is omitted, it splits on any whitespace.
- **`join(iterable)`:** Joins elements of an iterable (like a list) into a single string, using the string calling the method as a separator.

```python
csv_data = "apple,banana,cherry"
fruits_list = csv_data.split(",")
print(fruits_list)                # ['apple', 'banana', '
    cherry']

# Joining them back with a hyphen
joined_string = " - ".join(fruits_list)
print(joined_string)              # apple - banana - cherry
```

Listing 5.9: Split and Join

Validation and Character Checks

These methods return `True` or `False` based on the string's content.

Method	Condition to Return True
`startswith(prefix)`	The string starts with the specified prefix.
`endswith(suffix)`	The string ends with the specified suffix.
`isalnum()`	All characters are alphanumeric (letters or numbers) and length ≥ 1.
`isalpha()`	All characters are alphabetic letters.
`isdigit()`	All characters are digits.
`islower()` / `isupper()`	All cased characters are lowercase / uppercase.
`isspace()`	All characters are whitespace (spaces, tabs, newlines).

Table 5.2: Boolean Validation Methods

5.1.7 Important University Questions

PYQ: List various string operations and explain any three with example. (Su'23, W'23)

String Operations: Concatenation (+), Repetition (*), Membership (`in`/`not in`), Comparison, Slicing, and `len()`.

1. Concatenation: Joining two or more strings together using the + operator. It creates a new string containing the combined content of the operands.

```
first_name = "Alan"
last_name = "Turing"
full_name = first_name + " " + last_name
print(full_name)     # Alan Turing
```

2. Repetition: Repeating a string multiple times using the * operator. This is extremely useful for generating padding, dividers, or repeating patterns.

```
divider = "=" * 20
print(divider)       # ====================
echo = "Hello! " * 3
print(echo)          # Hello! Hello! Hello!
```

3. Slicing: Extracting a sub-portion of a string using index ranges `[start:stop:step]`.

```
s = "Python Programming"
print(s[0:6])     # Python (characters from index 0 to 5)
print(s[7:])      # Programming (from index 7 to the end)
print(s[::-1])    # gnimmargorP nohtyP (reverses the
    string completely)
```

PYQ: Explain string methods: count(), upper(), replace() with examples. (W'24)

1. `count(sub)` — Count occurrences of substring: This method counts how many non-overlapping times a specific substring appears in the string. It is helpful for finding the frequency of letters or words.

```
text = "mississippi"
print(text.count("s"))     # 4
print(text.count("iss"))   # 2 (non-overlapping)
```

2. `upper()` — Convert to uppercase: This method returns a completely new string where all alphabetical characters have been converted to their uppercase equivalents. Non-alphabetical characters remain unchanged.

```
msg = "Warning: low battery!"
print(msg.upper())         # WARNING: LOW BATTERY!
```

3. `replace(old, new)` — Replace all occurrences: This method searches the string for the `old` substring and replaces every occurrence of it with the `new` substring. It returns a new string, maintaining the immutability of the original.

```
sentence = "I love Java. Java is fun."
new_sentence = sentence.replace("Java", "Python")
print(new_sentence)        # I love Python. Python is fun.
```

PYQ: Explain slicing of string with suitable example. (W'24)

String Slicing allows for extracting a continuous sequence of characters from a string to form a new substring. It is done using the syntax `string[start:stop:step]`.
Rules and Behaviors:

- **start**: The beginning index (included). Defaults to 0 if omitted.
- **stop**: The ending index (excluded). Defaults to the end of the string if omitted.
- **step**: The jump size between indices. Defaults to 1. A negative step means slicing backwards.
- If `start` is greater than `stop` and the step is positive, slicing returns an empty string. Out-of-bound indices are handled gracefully without errors.

```
s = "ABCDEFGHIJ"

# Standard slicing
print(s[2:6])        # CDEF    (indices 2, 3, 4, 5)
print(s[:4])         # ABCD    (first 4 characters, 0 to 3)
print(s[6:])         # GHIJ    (from index 6 to the very end)

# Slicing with steps
print(s[0:8:2])      # ACEG    (every 2nd char from 0 to 7)

# Reverse slicing using negative step
print(s[::-1])       # JIHGFEDCBA  (reversed entirely)
print(s[8:4:-1])     # IHGF    (backwards from index 8 down to 5)

# Negative index slicing
print(s[-4:-1])      # GHI    (from 4th-last up to 2nd-last character)
```

PYQ: Write Python program to count and display vowels, consonants, uppercase letters and lowercase letters in a string. (Su'24 OR)

```python
text = input("Enter a string: ")

# Initialize counters
vowels = 0
consonants = 0
uppercase = 0
lowercase = 0

# Define a string of vowels for easy membership checking
vowel_set = "aeiouAEIOU"

for char in text:
    # Check if the character is an alphabet letter
    if char.isalpha():
        # Vowel or Consonant check
        if char in vowel_set:
            vowels += 1
        else:
            consonants += 1

        # Case check
        if char.isupper():
            uppercase += 1
        elif char.islower():
            lowercase += 1

print("\n--- String Analysis Results ---")
print(f"Total Vowels     : {vowels}")
print(f"Total Consonants : {consonants}")
print(f"Total Uppercase  : {uppercase}")
print(f"Total Lowercase  : {lowercase}")
```

Listing 5.10: String Character Analysis Program

Output

```
Enter a string: Hello World Python 2024!

--- String Analysis Results ---
Total Vowels     : 5
Total Consonants : 13
Total Uppercase  : 3
Total Lowercase  : 15
```

PYQ: Write Python program to check if a substring is present in a given string. (Su'24)

```python
main_string = input("Enter the main string: ")
search_string = input("Enter the substring to search for: ")

# Method 1: Using the 'in' membership operator (Most Pythonic)
print("\n--- Using 'in' operator ---")
if search_string in main_string:
    print(f"Success: '{search_string}' is present in the main string.")
else:
    print(f"Failure: '{search_string}' is NOT present.")

# Method 2: Using the find() method (Useful if you need the exact index)
print("\n--- Using find() method ---")
position = main_string.find(search_string)

if position != -1:
    print(f"Found! The substring starts at index {position}.")
else:
    print("Substring not found. find() returned -1.")

# Method 3: Using the count() method (Useful if you need frequency)
print("\n--- Using count() method ---")
occurrences = main_string.count(search_string)
if occurrences > 0:
    print(f"The substring appears {occurrences} time(s).")
else:
    print("The substring does not appear.")
```

Listing 5.11: Substring search implementation

PYQ: Explain indexing in string with example. (Su'25 OR)

String Indexing refers to accessing individual characters of a string using their numerical position. Since a string is a sequence, every character is assigned a unique index.

Python utilizes **zero-based indexing**. This means the count starts from 0, not 1. Thus, the first character is at index 0, the second at index 1, and so forth.

Additionally, Python supports **negative indexing**, which is a convenient way to access elements from the end of the string without needing to calculate its length.

The index `-1` refers to the last character, `-2` is the second to last, etc.

```
s = "PYTHON"
# Visualizing Indices:
# Character:   P    Y    T    H    O    N
# Positive :   0    1    2    3    4    5
# Negative :  -6   -5   -4   -3   -2   -1

# 1. Accessing via positive indices
print("First character:", s[0])      # Output: P
print("Fourth character:", s[3])     # Output: H
print("Last character:", s[5])       # Output: N

# 2. Accessing via negative indices
print("Last character:", s[-1])      # Output: N
print("Second to last:", s[-2])      # Output: O
print("First character:", s[-6])     # Output: P

# 3. Iterating using indexing and a while loop
print("\nIterating through string:")
index = 0
while index < len(s):
    print(f"Index {index} -> Character {s[index]}")
    index += 1
```

5.2 Lists

Definition: List

A **list** in Python is a built-in, versatile data structure that serves as an ordered, *mutable* (changeable), and heterogeneous collection of items. Lists are enclosed in square brackets `[]`, and their elements are separated by commas. As one of the most fundamental sequence types in Python, lists can contain elements of any data type, including integers, floating-point numbers, strings, objects, or even other lists (a concept known as nested lists). Lists maintain their insertion order and allow for duplicate values.

Lists are highly dynamic, meaning they can grow or shrink in size automatically as elements are added or removed. This eliminates the need to declare the size of the list ahead of time, a requirement common in arrays found in statically typed languages like C or Java. Behind the scenes, Python lists are implemented as dynamic arrays, offering fast and efficient access to elements via their index.

5.2.1 Key Characteristics of Lists

Understanding the defining traits of lists is crucial for utilizing them effectively in Python programming. Here is a detailed breakdown of their characteristics:

- **Ordered Collection:** Elements in a list have a specific, defined order, and that order is preserved unless explicitly modified. When an item is added to the list, it is placed at the end by default. You can rely on the list to maintain this order, which is why accessing elements by their numerical index is possible.
- **Mutable (Changeable):** Unlike strings and tuples, lists are mutable. This means that you can alter the contents of a list after it has been created. You can reassign individual elements, overwrite entire slices of the list, or add and remove items without needing to create a completely new list object in memory.
- **Heterogeneous Elements:** Lists are not restricted to storing a single data type. A single list can simultaneously hold integers, strings, booleans, floating-point numbers, and complex objects. For instance, `[42, "hello", 3.14159, True]` is a perfectly valid list.
- **Allows Duplicate Values:** Because list elements are indexed by their position, a list can contain the same value multiple times. For example, `[1, 2, 2, 3, 3, 3]` is a valid list with distinct indices for each duplicate item.
- **Dynamic Resizing:** Lists grow and shrink dynamically as you append or remove elements. Python handles the memory allocation and reallocation implicitly, freeing the programmer from manual memory management tasks.

5.2.2 Creating Lists

Python offers several ways to create lists, catering to different scenarios. You can create lists manually, generate them using built-in functions, or convert other data types into lists.

Direct Initialization

The most straightforward way to create a list is by placing comma-separated values inside square brackets.

```
# Empty list
empty_list = []

# List of homogenous elements (all integers)
temperatures = [32, 35, 28, 40, 31]

# List of heterogeneous elements (mixed data types)
student_record = [101, "Alice Smith", 89.5, True, None]

# Nested list (list containing other lists)
grid_matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Listing 5.12: Direct list initialization

Using the `list()` Constructor

The built-in `list()` constructor can be used to create a list from any iterable object, such as a string, tuple, set, or dictionary.

```
# Creating a list from a string
chars = list("Python")
print(chars)  # Output: ['P', 'y', 't', 'h', 'o', 'n']

# Creating a list from a tuple
vowels_tuple = ('a', 'e', 'i', 'o', 'u')
vowels_list = list(vowels_tuple)
print(vowels_list)  # Output: ['a', 'e', 'i', 'o', 'u']

# Creating an empty list using the constructor
another_empty = list()
```

Listing 5.13: Using the list() constructor

Creating Lists with `range()`

When you need a sequence of numbers, the `range()` function paired with the `list()` constructor provides a highly efficient way to generate the list.

```
# Generate a list of numbers from 0 to 9
digits = list(range(10))
print(digits)  # Output: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
4
5 # Generate a list of even numbers from 2 to 20
6 evens = list(range(2, 21, 2))
7 print(evens)    # Output: [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

Listing 5.14: Creating lists using range()

5.2.3 Accessing List Elements: Indexing and Slicing

List elements can be accessed individually or in continuous chunks using indexing and slicing, respectively. These operations work identically to how they function for strings.

Positive and Negative Indexing

Every element in a list has a unique integer index representing its position. Python supports both zero-based positive indexing (starting from the left) and negative indexing (starting from the right).

- **Positive Indexing:** The first element is at index `0`, the second at `1`, and so on.
- **Negative Indexing:** The last element is at index `-1`, the second to last at `-2`, and so forth. This is particularly useful for accessing elements at the end of a long list without needing to calculate its length.

```
1 colors = ["red", "green", "blue", "yellow", "purple"]
2 # Positive:   0        1        2         3         4
3 # Negative:  -5       -4       -3        -2        -1
4
5 print(colors[0])     # Output: red (first element)
6 print(colors[2])     # Output: blue (third element)
7 print(colors[-1])    # Output: purple (last element)
8 print(colors[-3])    # Output: blue (third element from the
    end)
9
10 # Attempting to access an index out of bounds raises an error
11 # print(colors[10]) # Raises IndexError: list index out of
    range
```

Listing 5.15: List indexing examples

Slicing Lists

Slicing allows you to extract a sublist (a specific range of items) from the original list. The syntax for slicing is `list[start:stop:step]`.

- **`start`:** The starting index (inclusive). If omitted, defaults to `0`.
- **`stop`:** The ending index (exclusive). If omitted, defaults to the length of the list.
- **`step`:** The stride or jump between elements. If omitted, defaults to `1`.

```
alphabet = ['A', 'B', 'C', 'D', 'E', 'F', 'G']

# Basic slicing
print(alphabet[1:4])     # Output: ['B', 'C', 'D'] (index 1 to 3)

# Slicing from the start or to the end
print(alphabet[:3])      # Output: ['A', 'B', 'C'] (start to index 2)
print(alphabet[4:])      # Output: ['E', 'F', 'G'] (index 4 to end)

# Using a step value
print(alphabet[::2])     # Output: ['A', 'C', 'E', 'G'] (every second element)
print(alphabet[1::2])    # Output: ['B', 'D', 'F'] (every second element, starting at index 1)

# Reversing a list using slicing
print(alphabet[::-1])    # Output: ['G', 'F', 'E', 'D', 'C', 'B', 'A']
```

Listing 5.16: List slicing examples

5.2.4 Modifying Lists

The mutability of lists is one of their most powerful features. You can modify individual elements or entire slices directly.

Modifying Individual Elements

To change a single item, access it via its index and assign a new value using the assignment operator (=).

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

Listing 5.17: Modifying single elements

Modifying Multiple Elements via Slicing

You can replace multiple elements simultaneously by assigning an iterable (like another list) to a slice. The iterable being assigned does not need to have the same length as the slice it is replacing, which allows you to insert or remove items dynamically.

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

# Replacing a slice with a list of the same length
nums[1:3] = [25, 35]
print(nums)  # Output: [10, 25, 35, 40, 50]
```

```
6
7  # Replacing a slice with a longer list (increases overall
       list size)
8  nums[1:2] = [21, 22, 23]
9  print(nums)  # Output: [10, 21, 22, 23, 35, 40, 50]
10
11 # Replacing a slice with an empty list (removes elements)
12 nums[4:] = []
13 print(nums)  # Output: [10, 21, 22, 23]
```

Listing 5.18: Modifying slices

5.2.5 List Operations

Lists support several operators that make manipulating and combining them intuitive.

- **Concatenation (+):** Joins two or more lists to create a new list.
- **Repetition (*):** Repeats a list a specified number of times to create a new list.
- **Membership (`in`, `not in`):** Checks whether a specific element exists within the list, returning a boolean value.

```
1  list1 = [1, 2, 3]
2  list2 = [4, 5, 6]
3
4  # Concatenation
5  combined = list1 + list2
6  print(combined)  # Output: [1, 2, 3, 4, 5, 6]
7
8  # Repetition
9  repeated = ["Hi"] * 3
10 print(repeated)  # Output: ['Hi', 'Hi', 'Hi']
11
12 # Membership
13 print(2 in list1)        # Output: True
14 print(10 in list1)       # Output: False
15 print(5 not in list2)    # Output: False
```

Listing 5.19: Common list operations

5.2.6 List Methods and Built-in Functions

Python provides a rich set of built-in methods and functions specifically designed to operate on lists. These methods modify the list in-place (meaning they alter the original list directly rather than creating a new one) unless stated otherwise.

Method / Function	Description and Example
`lst.append(x)`	Adds element `x` to the very end of the list. E.g., `[1,2].append(3)` results in `[1,2,3]`.
`lst.insert(i, x)`	Inserts element `x` at the specific index `i`. Existing elements are shifted to the right.
`lst.extend(iterable)`	Appends all items from an iterable (like another list or tuple) to the end of `lst`.
`lst.remove(x)`	Removes the *first* occurrence of element `x`. Raises a `ValueError` if `x` is not found.
`lst.pop([i])`	Removes and returns the element at index `i`. If `i` is omitted, it removes and returns the *last* element.
`lst.clear()`	Removes all items from the list, leaving it empty `[]`.
`lst.index(x)`	Returns the index of the first occurrence of element `x`. Raises a `ValueError` if not found.
`lst.count(x)`	Returns the total number of times element `x` appears in the list.
`lst.sort()`	Sorts the list items in-place in ascending order. Use `sort(reverse=True)` for descending order.
`lst.reverse()`	Reverses the order of the list items in-place.
`lst.copy()`	Returns a shallow copy of the list.
`len(lst)`	A built-in function that returns the total number of elements in the list.
`sum(lst)`	A built-in function that calculates the sum of all numeric elements in the list.
`min(lst)` / `max(lst)`	Built-in functions that return the smallest and largest elements in the list, respectively.
`sorted(lst)`	A built-in function that returns a *new* sorted list, leaving the original list unchanged.

Table 5.3: Python List Methods and Built-in Functions

```
numbers = [3, 1, 4, 1, 5, 9, 2, 6]
print("Original:", numbers)

# Adding elements
numbers.append(7)
print("After append(7):", numbers)        # [3, 1, 4, 1, 5, 9,
    2, 6, 7]

numbers.insert(0, 0)
```

```
print("After insert(0,0):", numbers)      # [0, 3, 1, 4, 1, 5,
    9, 2, 6, 7]

numbers.extend([8, 10])
print("After extend([8, 10]):", numbers) # [0, 3, 1, 4, 1, 5,
    9, 2, 6, 7, 8, 10]

# Removing elements
numbers.remove(1) # Removes the first '1'
print("After remove(1):", numbers)        # [0, 3, 4, 1, 5, 9,
    2, 6, 7, 8, 10]

popped_item = numbers.pop() # Removes the last element
print("Popped item:", popped_item)        # 10
print("After pop():", numbers)            # [0, 3, 4, 1, 5, 9,
    2, 6, 7, 8]

# Finding elements
print("Index of 5:", numbers.index(5))    # 4
print("Count of 1:", numbers.count(1))    # 1 (since the first
    '1' was removed)

# Organizing elements
numbers.sort()
print("After sort():", numbers)           # [0, 1, 2, 3, 4, 5,
    6, 7, 8, 9]

numbers.reverse()
print("After reverse():", numbers)        # [9, 8, 7, 6, 5, 4,
    3, 2, 1, 0]

# Built-in functions
print("Sum:", sum(numbers))               # 45
print("Max:", max(numbers))               # 9
print("Min:", min(numbers))               # 0
print("Length:", len(numbers))            # 10
```

Listing 5.20: Comprehensive demonstration of list methods

5.2.7 Nested Lists

Lists can contain other lists as their elements. This creates a nested list structure, which is extremely useful for representing multi-dimensional data, such as matrices, grids, or tables.

To access elements within a nested list, you use multiple indices. The first index accesses the inner list, and the subsequent indices access elements within that inner list.

```
# A 3x3 matrix represented as a list of lists
matrix = [
    [1, 2, 3],
    [4, 5, 6],
```

```
    [7, 8, 9]
]

# Accessing the entire second row (index 1)
print(matrix[1])        # Output: [4, 5, 6]

# Accessing a specific element: row 1, column 2
print(matrix[1][2])     # Output: 6

# Traversing the nested list using nested loops
for row in matrix:
    for elem in row:
        print(elem, end="\t")
    print()
```

Listing 5.21: Creating and accessing nested lists

Output

```
1 2 3
4 5 6
7 8 9
```

5.2.8 List Comprehension

List comprehension is one of Python's most defining and elegant features. It provides a concise and highly optimized way to create new lists based on existing iterables (like other lists, strings, or ranges). List comprehensions can often replace multi-line `for` loops, making the code more readable and executing faster under the hood.

The general syntax for list comprehension is: `[expression for item in iterable if condition]`

- **`expression`:** The operation to perform on the item, or simply the item itself, which dictates what gets added to the new list.
- **`for item in iterable`:** The standard iteration loop.
- **`if condition`:** (Optional) A filter that determines whether the item should be included in the new list.

```
# Traditional loop to create a list of squares
squares_loop = []
for x in range(1, 6):
    squares_loop.append(x**2)

# Equivalent list comprehension
squares_comp = [x**2 for x in range(1, 6)]
print(squares_comp)     # Output: [1, 4, 9, 16, 25]

# Using conditions to filter data: Extracting even numbers
evens = [x for x in range(1, 21) if x % 2 == 0]
```

```
print(evens)            # Output: [2, 4, 6, 8, 10, 12, 14, 16,
    18, 20]

# List comprehension with strings
words = ["hello", "world", "python", "programming"]
uppercase_words = [word.upper() for word in words]
print(uppercase_words) # Output: ['HELLO', 'WORLD', 'PYTHON',
    'PROGRAMMING']
```

Listing 5.22: List comprehension vs traditional loops

5.2.9 Memory Management and Copying Lists

When working with lists, it is vital to understand how assignment works in memory. Because lists are mutable, assigning an existing list to a new variable does *not* create a new list; it creates a new **reference** (or alias) pointing to the exact same list in memory. Changing the list via one reference will alter the data for all references.

To create an actual, separate copy of a list, you must perform either a shallow copy or a deep copy.

```
# Aliasing: Both variables point to the same list in memory
original = [1, 2, 3]
alias = original
alias[0] = 99
print(original)  # Output: [99, 2, 3] (Original is affected)

# Shallow Copy using the copy() method or slicing [:]
original_list = [1, 2, 3]
copied_list = original_list.copy()  # or original_list[:]
copied_list[0] = 99
print(original_list) # Output: [1, 2, 3] (Original is safe)
print(copied_list)   # Output: [99, 2, 3]
```

Listing 5.23: Aliasing vs Copying

Note: For nested lists, a shallow copy is insufficient because the inner lists are still passed by reference. In such cases, the `copy.deepcopy()` function from the `copy` module is required to ensure independent clones of all nested structures.

5.2.10 Previous Year Questions

PYQ: Write Python code to swap two elements in a list. (Su'23, W'23, Su'24, Su'25 OR)

Swapping elements is a common operation. Python allows for an elegant swapping mechanism using tuple unpacking, which eliminates the need for a temporary variable.

```
lst = [10, 20, 30, 40, 50]
print("Before swap:", lst)

```

```python
# Get positions to swap from the user
i = int(input("Enter index 1: "))
j = int(input("Enter index 2: "))

# Pythonic swap via tuple unpacking
lst[i], lst[j] = lst[j], lst[i]

print("After swap:", lst)
```

Listing 5.24: Swap two elements in a list

Output

```
Before swap: [10, 20, 30, 40, 50]
Enter index 1: 1
Enter index 2: 3
After swap: [10, 40, 30, 20, 50]
```

PYQ: Write Python code to find sum of all elements in a list. (Su'23 OR, W'23 OR, Su'24 OR)

There are multiple ways to find the sum of elements in a list. The most Pythonic way is to use the built-in `sum()` function, but doing it manually via a loop is also fundamentally sound.

```python
# Method 1: Using the built-in sum() function
nums = [10, 20, 30, 40, 50]
print("Sum (built-in):", sum(nums))    # Output: 150

# Method 2: Using a manual for loop
total = 0
for n in nums:
    total += n
print("Sum (loop):", total)            # Output: 150

# Method 3: Taking user input dynamically
n = int(input("How many numbers do you want to enter? ")
    )
dynamic_list = []
for i in range(n):
    num = int(input(f"Enter number {i+1}: "))
    dynamic_list.append(num)
print("Sum of entered numbers:", sum(dynamic_list))
```

Listing 5.25: Sum of list elements

PYQ: Find the smallest number in the given list: List1 = [21, 6, 3, 18, 12, 15]. (Su'25)

Finding the minimum (or maximum) value is a standard array processing technique. Python simplifies this with the `min()` function.

```
List1 = [21, 6, 3, 18, 12, 15]

# Method 1: Utilizing built-in min() function
print("Smallest element (built-in):", min(List1))    # Output: 3

# Method 2: Manual traversal
smallest = List1[0]
for num in List1:
    if num < smallest:
        smallest = num
print("Smallest element (manual loop):", smallest)    # Output: 3
```

Listing 5.26: Find smallest in a list

PYQ: Develop Python program to find multiplication of all elements in the list. (W'25)

Unlike calculating a sum, Python does not have a built-in "product" function for lists (unless importing the `math.prod` module). Iterating through the list is a standard approach.

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

# Initialize product variable to 1 (multiplicative identity)
product = 1
for n in nums:
    product *= n

print("Original List:", nums)
print("Product of all elements:", product)    # Output: 120
```

Listing 5.27: Product of all list elements

PYQ: Explain List Methods and its built-in Functions. (Su'24, Su'25)

List Methods and Built-in Functions: Lists in Python are accompanied by a diverse set of built-in functions (like `len()`, `sum()`, `max()`) and methods bound to the list object (like `.append()`, `.sort()`, `.remove()`). Functions act upon the list,

whereas methods manipulate the list internally.

```
lst = [5, 2, 8, 1, 9, 3, 7]

lst.append(10)
print("append(10):", lst)            # Adds 10 to the end

lst.sort()
print("sort():", lst)                # [1, 2, 3, 5, 7, 8,
    9, 10]

lst.reverse()
print("reverse():", lst)             # [10, 9, 8, 7, 5,
   3, 2, 1]

lst.insert(2, 99)
print("insert(2,99):", lst)          # inserts 99 at
   index 2

print("index(9):", lst.index(9))     # Finds index
   position of 9
print("count(9):", lst.count(9))     # Counts occurrences
    of 9

lst.remove(99)                       # Removes the value
   99
print("remove(99):", lst)

popped = lst.pop()                   # Removes and
   returns the last element
print("pop():", popped, " --  List:", lst)

# Utilizing Built-in functions
print("len:", len(lst), "sum:", sum(lst), "max:", max(
   lst), "min:", min(lst))
```

Listing 5.28: List methods demonstration

PYQ: Develop Python code to create a list of prime and non-prime numbers in range 1 to 50. (W'24 OR)

This problem combines logic control, function creation, loops, and list manipulation (`append()`).

```
# Function to evaluate if a number is prime
def is_prime(n):
    if n < 2:
        return False
    # Check divisors up to the square root of n
```

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

# Initialize empty lists
primes = []
non_primes = []

# Iterate through the specified range
for n in range(1, 51):
    if is_prime(n):
        primes.append(n)
    else:
        non_primes.append(n)

print("Prime numbers (1-50)     :", primes)
print("Non-prime numbers (1-50):", non_primes)
```

Listing 5.29: Separate primes and non-primes 1–50

PYQ: Given a = [1,4,9,16,25,36,49,64,81,100]. Write one line of Python that takes this list and makes a new list that has only the even elements. Also create a list of squares of all odd numbers from range 1 to 10. (Practical 13)

This question specifically tests knowledge of **List Comprehensions**. List comprehensions provide a concise way to generate new lists and filter existing ones in just a single line of code.

```
# Part A: Extract even elements from a given list
a = [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
even_elements = [x for x in a if x % 2 == 0]
print("Even elements:", even_elements)   # Output: [4,
    16, 36, 64, 100]

# Part B: Create a list of squares for odd numbers from
    1 to 10
odd_squares = [x**2 for x in range(1, 11) if x % 2 != 0]
print("Squares of odd numbers 1-10:", odd_squares)   #
    Output: [1, 9, 25, 49, 81]
```

Listing 5.30: List comprehension practical application

5.3 Tuples

5.3.1 Introduction to Tuples

In Python, a tuple is one of the built-in core data types used to store collections of data. A tuple is an ordered, *immutable* sequence of elements enclosed within parentheses `()`. While they appear structurally similar to lists, the defining characteristic of a tuple is its immutability—once a tuple is created, its size and elements cannot be altered. This means you cannot add, remove, or modify elements in a tuple after it has been defined.

Tuples are incredibly versatile and can store heterogeneous data, meaning a single tuple can seamlessly contain integers, strings, floats, booleans, and even other complex data structures like lists or other tuples. Because of their immutability, tuples are generally faster than lists and consume less memory, making them an excellent choice for storing data that should remain constant throughout the program's execution. They protect data against accidental modification.

A helpful analogy for a tuple is a row or a record in a database. For instance, a tuple representing a geographic coordinate `(latitude, longitude)` or a person's basic details `("Alice", 25, "Engineer")` relies on the specific order of elements to convey meaning. Because they have a fixed structure, you can always rely on the first element being the name, the second being the age, and so forth.

Definition: Tuple

A **tuple** is an ordered, *immutable* sequence of elements. They allow duplicate elements and can contain mixed data types. Unlike lists, **tuples cannot be changed after creation**. Tuples provide a safeguard against accidental modification and offer performance benefits over lists due to their fixed memory allocation.

5.3.2 Creating Tuples

Creating a tuple is straightforward. You typically enclose the elements in parentheses, separated by commas. However, it is the commas, rather than the parentheses, that truly define a tuple in Python. The parentheses are often optional but are highly recommended for clarity and readability to group the items visually.

When dealing with single-element tuples, you must include a trailing comma. Without the comma, Python evaluates the expression inside the parentheses mathematically (as a simple grouping) and ignores the tuple structure.

```
# 1. Empty tuple
empty_tuple = ()
print(type(empty_tuple)) # <class 'tuple'>

# 2. Single-element tuple (requires a trailing comma)
# Without the comma, Python treats the parentheses as a
    mathematical grouping operator.
single_item_tuple = (42,)
not_a_tuple = (42)
print(type(single_item_tuple))  # <class 'tuple'>
print(type(not_a_tuple))        # <class 'int'>
```

```python

# 3. Regular tuples with multiple elements
coordinates = (10, 20)
rgb_color = (255, 128, 0)
mixed_data = (1, "Python", 3.14, True)

# 4. Tuple creation without parentheses (Tuple Packing)
packed_tuple = 10, 20, 30, "Hello"
print(type(packed_tuple))       # <class 'tuple'>

# 5. Using the tuple() constructor from other iterables
from_list = tuple([1, 2, 3])
from_string = tuple("Hello")    # ('H', 'e', 'l', 'l', 'o')
```

Listing 5.31: Various ways to create tuples

5.3.3 Accessing Tuple Elements

Since tuples are ordered sequences, each element has a specific index, starting from `0` for the first element. You can access individual elements using positive and negative indexing, and extract subsets of the tuple using slicing, exactly as you would with lists or strings.

- **Positive Indexing:** Starts from `0` (first element) to `len(tuple) - 1` (last element).
- **Negative Indexing:** Starts from `-1` (last element), `-2` (second to last), etc. This is very useful when you want to access elements from the end without knowing the length of the tuple.
- **Slicing:** Syntax is `tuple[start:stop:step]`. It extracts a new tuple containing elements from the `start` index up to, but not including, the `stop` index. Slicing never modifies the original tuple; it always returns a new one.

```python
t = (10, 20, 30, 40, 50)

# Indexing
print(t[0])       # 10 (First element)
print(t[2])       # 30 (Third element)
print(t[-1])      # 50 (Last element)
print(t[-3])      # 30 (Third element from the right)

# Slicing
print(t[1:4])     # (20, 30, 40) - Elements from index 1 to 3
print(t[:3])      # (10, 20, 30) - First 3 elements
print(t[2:])      # (30, 40, 50) - From index 2 to the end
print(t[::-1])    # (50, 40, 30, 20, 10) - Reverses the tuple

# Iterating over a tuple
for item in t:
    print(item, end=" ")
# Output: 10 20 30 40 50
```

Listing 5.32: Tuple indexing, slicing, and iteration

5.3.4 Immutability and Mutable Elements Inside Tuples

The core characteristic of a tuple is immutability. If you try to assign a new value to an existing index (e.g., `t[0] = 100`), Python will immediately raise a `TypeError`. This strictness guarantees that the structure of the tuple remains exactly the same over its lifetime.

However, there is an important nuance: if a tuple contains a *mutable* object, such as a list or a dictionary, the tuple cannot be modified to point to a different object, but the mutable object *itself* can be modified in place. The tuple holds an immutable reference to the list; what happens inside the list does not affect the tuple's structural integrity from Python's perspective.

```
t = (1, 2, [3, 4])

# t[0] = 10          # TypeError: 'tuple' object does not
    support item assignment
# t[2] = [5, 6]      # TypeError: Cannot change the reference
    of the third element

# However, the list INSIDE the tuple can be modified
t[2].append(5)
print(t)             # Output: (1, 2, [3, 4, 5])

# Even replacing an element inside the internal list is
    allowed
t[2][0] = 99
print(t)             # Output: (1, 2, [99, 4, 5])
```

Listing 5.33: Tuple immutability nuance

5.3.5 Tuple Packing and Unpacking

Tuple packing and unpacking are powerful features in Python that make code concise and highly readable. **Packing** refers to placing multiple values into a single tuple variable (as seen earlier when creating tuples without parentheses). **Unpacking** is the reverse process, where you extract elements from a tuple and assign them to multiple independent variables in a single statement.

The number of variables on the left side of the assignment operator must match the exact number of elements in the tuple, otherwise Python will throw a `ValueError`. However, Python 3 introduced extended unpacking with the asterisk `*` operator to gather remaining items into a list.

```
# Basic unpacking
student = ("Alice", 20, "A")
name, age, grade = student
print(name)    # Alice
print(age)     # 20
```

```

# Swapping variables gracefully without a temporary variable
x, y = 10, 20
x, y = y, x
print(x, y)    # 20, 10

# Extended unpacking using the * operator (Python 3+)
# The * gathers remaining elements into a list
numbers = (1, 2, 3, 4, 5)
first, *middle, last = numbers
print(first)  # 1
print(middle) # [2, 3, 4]  <-- Note: packed into a list!
print(last)   # 5
```

Listing 5.34: Tuple unpacking and extended unpacking

5.3.6 Tuple Operations, Built-in Functions, and Methods

Since tuples are immutable, they do not have methods like `append()`, `remove()`, or `sort()` which alter the sequence in place. They only support operations that analyze the tuple or operations that create an entirely new tuple from existing ones.

Operation/Function	Description	Example
`len(t)`	Returns the total number of elements in the tuple.	`len((1,2,3))` → 3
`t1 + t2`	**Concatenation**: Combines two tuples to create a new one.	`(1,2)+(3,4)` → (1,2,3,4)
`t * n`	**Repetition**: Creates a new tuple by repeating it `n` times.	`(1,2)*3` → (1,2,1,2,1,2)
`x in t`	**Membership**: Returns `True` if `x` exists in `t`.	`3 in (1,2,3)` → True
`min(t)` / `max(t)`	Returns the minimum or maximum element.	`max((1,5,3))` → 5
`sum(t)`	Calculates the sum of all numeric elements in the tuple.	`sum((1,2,3))` → 6
`sorted(t)`	Returns a sorted **list** of the elements. Original tuple is untouched.	`sorted((3,1,2))` → [1,2,3]
Tuple Methods		
`t.count(x)`	Returns the number of times `x` appears in the tuple.	`(1,1,2).count(1)` → 2
`t.index(x)`	Returns the index of the *first* occurrence of `x`. Raises ValueError if not found.	`(5,8,3).index(8)` → 1

Table 5.4: Tuple Operations, Built-in Functions, and Methods

```
t = (3, 1, 4, 1, 5, 9, 2, 6, 5)

print("Length  :", len(t))              # 9
print("Count(1):", t.count(1))          # 2
print("Index(9):", t.index(9))          # 5
print("Max     :", max(t))              # 9
print("Min     :", min(t))              # 1
print("Sum     :", sum(t))              # 36
print("Sorted  :", sorted(t))           # Returns a list: [1,
    1, 2, 3, 4, 5, 5, 6, 9]

t2 = (10, 20)
print("t + t2  :", t + t2)              # Creates a new tuple
    combining both
print("t2 * 3  :", t2 * 3)              # (10, 20, 10, 20, 10,
     20)
print("5 in t  :", 5 in t)              # True
```

Listing 5.35: Demonstrating tuple operations and methods

5.3.7 Nested Tuples

A tuple can contain other tuples as its elements, which is known as a nested tuple. This is particularly useful for representing multi-dimensional data, such as matrices, coordinate grids, or hierarchical database records, in an immutable structure. Nesting allows for a robust representation of tree-like structures where data integrity is paramount.

```
# Matrix as a nested tuple (an immutable 2D structure)
matrix = (
    (1, 2, 3),
    (4, 5, 6),
    (7, 8, 9)
)

print(matrix[1])        # (4, 5, 6) -> Accessing the second row entirely
print(matrix[1][2])     # 6 -> Accessing the element in the second row, third column

# Iterating through a nested tuple using nested loops
for row in matrix:
    for item in row:
        print(item, end="\t")
    print()
```

Listing 5.36: Working with nested tuples

5.3.8 List vs Tuple Comparison

Understanding when to use a list versus a tuple is crucial for writing efficient and semantic Python code. Lists are for collections of homogeneous items where the number of items might change dynamically. Tuples are for collections of heterogeneous items where the structure and sequence imply meaning, and the data is meant to remain constant. Using tuples instead of lists signals to other developers that the data sequence shouldn't be altered.

Feature	List	Tuple
Syntax	Enclosed in square brackets `[]`	Enclosed in parentheses `()`
Mutability	Mutable (elements can be added, removed, or changed)	Immutable (cannot be changed after creation)
Methods	Extensive methods: `append()`, `insert()`, `remove()`, `pop()`, `sort()`, etc.	Limited methods: only `count()` and `index()`
Memory	Consumes more memory due to dynamic sizing overhead	Consumes less memory (more space-efficient)
Performance	Slower for iteration and creation	Faster for iteration and creation
Dictionary Key	Cannot be used as a dictionary key (unhashable)	Can be used as a dictionary key if all its elements are immutable
Use Case	Dynamic data that changes over time (e.g., shopping cart, user inputs, task lists)	Fixed, structured data (e.g., geographic coordinates, RGB values, returning multiple values from a function)

Table 5.5: Detailed Comparison: List vs Tuple

PYQ: Explain tuple Operations, Functions and Methods. (Su'23, Su'24)

Creating and accessing:

```
t = (10, 20, 30, 40, 50)
print(t[0])          # 10 (indexing)
print(t[1:4])        # (20, 30, 40) (slicing)
print(t[-1])         # 50 (negative indexing)
```

Operations:

```
t1 = (1, 2, 3)
t2 = (4, 5, 6)

print(t1 + t2)       # (1, 2, 3, 4, 5, 6)  --
    concatenation
print(t1 * 2)        # (1, 2, 3, 1, 2, 3)  --  repetition
print(2 in t1)       # True  --  membership testing
```

Built-in functions:

```
data = (5, 2, 8, 1, 9)
print(len(data))       # 5 (total elements)
print(min(data))       # 1 (smallest element)
print(max(data))       # 9 (largest element)
print(sum(data))       # 25 (sum of all elements)
print(sorted(data))    # [1, 2, 5, 8, 9]  --  returns a
    sorted list
```

Methods:

```
data = (3, 1, 4, 1, 5, 9, 1)
print(data.count(1))     # 3 (counts occurrences of 1)
print(data.index(5))     # 4 (returns the index of the
    first 5)
```

PYQ: Explain concept of nested tuple with example. (Su'25)

Nested Tuple: A tuple that contains other tuples as its elements. It is widely used to represent tabular data, matrices, or hierarchical records safely because of its immutability. Each sub-tuple can represent a row of a dataset.

```
# Student records stored as (name, age, marks)
students = (
    ("Alice", 20, 85),
    ("Bob",   21, 90),
    ("Carol", 19, 78)
)

# Accessing the first outer tuple (Alice's record)
print(students[0])          # ('Alice', 20, 85)

# Accessing inner elements
print(students[1][0])       # Bob (Name of the second
    student)
print(students[2][2])       # 78 (Marks of the third
    student)

# Iterating over the nested tuple and unpacking records
for student in students:
    name, age, marks = student      # unpacking inner
    tuple
    print(f"Name: {name:8} Age: {age} Marks: {marks}")
```

Output

```
Name: Alice    Age: 20 Marks: 85
Name: Bob      Age: 21 Marks: 90
Name: Carol    Age: 19 Marks: 78
```

5.4 Sets

5.4.1 Introduction to Sets

A set is an unordered, unindexed, and mutable collection of unique elements. Sets are inspired by the mathematical concept of sets and are highly optimized for testing membership, eliminating duplicate entries from other collections, and performing mathematical operations such as union, intersection, and difference.

Sets do not record element position or order of insertion. Accordingly, sets do not support indexing, slicing, or other sequence-like behavior. You cannot ask a set for its "first" or "last" element. Every element in a set must be strictly unique; if you attempt to add a duplicate, the set simply ignores it and silently proceeds.

Furthermore, sets place a strict restriction on the *types* of elements they can contain: all elements in a set must be **hashable** (immutable). This means you can add integers, strings, floats, and tuples to a set, but you cannot add lists, dictionaries, or other sets, because those types are mutable and their underlying values could theoretically change, breaking the hash table's integrity. Under the hood, sets use hash tables, which makes checking if an element exists an incredibly fast $O(1)$ operation on average, dramatically faster than searching through a list.

Definition: Set

A **set** is an *unordered*, *mutable* collection of **unique** (non-duplicate), **hashable** elements enclosed in curly braces `\{\}`. Sets are implemented using hash tables, making them extraordinarily fast for membership testing (`in` operator) and for performing classical set mathematics.

5.4.2 Creating Sets

You can create a set by placing a comma-separated sequence of elements within curly braces `\{\}`, or by using the built-in `set()` constructor function.

A common syntax trap for beginners occurs when trying to create an empty set. Because empty curly braces `\{\}` are historically reserved for creating empty dictionaries in Python, you *must* use the `set()` function explicitly to create an empty set.

```
# 1. Creating a set using set literal syntax
fruits = {"apple", "banana", "cherry", "apple"}
print(fruits)        # {'banana', 'apple', 'cherry'} -- order varies, duplicates removed!

# 2. Creating a set from a list (Useful for removing duplicates)
numbers_list = [1, 2, 2, 3, 3, 3, 4, 5, 5]
unique_numbers = set(numbers_list)
print(unique_numbers)    # {1, 2, 3, 4, 5}

# 3. Creating an empty set
empty_dict = {}          # This creates a Dictionary, NOT a set!
empty_set = set()        # This is the correct way to create an empty set
print(type(empty_dict)) # <class 'dict'>
```

```
print(type(empty_set))  # <class 'set'>

# 4. Sets can contain mixed data types (as long as they are
    immutable)
mixed_set = {1, "Python", 3.14, (10, 20)} # Tuple is allowed,
    but a List would cause a TypeError
```

Listing 5.37: Creating sets and eliminating duplicates

5.4.3 Mathematical Set Operations

Sets in Python provide powerful built-in methods and operators to perform mathematical set operations. These operations are essential for comparing datasets, finding commonalities, or isolating differences. You can achieve these operations either by using symbolic operators or by using named methods. The operators require both operands to be sets, while the methods can accept any iterable as an argument.

- **Union (`|` or `union()`):** Returns a new set containing all elements from both sets. Useful for merging collections while keeping items unique.
- **Intersection (`&` or `intersection()`):** Returns a new set containing only the elements that are common to both sets. Useful for finding overlap.
- **Difference (`-` or `difference()`):** Returns a new set with elements that are in the first set but *not* in the second set. The order of sets matters here.
- **Symmetric Difference (`^` or `symmetric_difference()`):** Returns a new set containing elements that are in either of the sets, but *not* in both. It is essentially the union minus the intersection.

```
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}

# Union: All unique elements from A and B
print("Union (A | B)        :", A | B)                    # {1,
    2, 3, 4, 5, 6, 7, 8}
print("Union method         :", A.union(B))

# Intersection: Elements present in both A and B
print("Intersection (A & B):", A & B)                     # {4,
    5}
print("Intersection method :", A.intersection(B))

# Difference: Elements in A that are not in B
print("Difference (A - B)  :", A - B)                     # {1,
    2, 3}
print("Difference (B - A)  :", B - A)                     # {8,
    6, 7} (Order matters!)

# Symmetric Difference: Elements in A or B, but NOT both
```

```
print("Sym Diff (A ^ B)     :", A ^ B)                    # {1,
    2, 3, 6, 7, 8}
print("Sym Diff method      :", A.symmetric_difference(B))
```

Listing 5.38: Set mathematical operations

5.4.4 Set Methods for Modification

Although sets are unordered, they are mutable, allowing you to add or remove elements after the set has been created. There are a few different ways to remove items, and choosing the right one depends on whether you expect the item to exist and how you want to handle errors if it does not.

Method	Description
`s.add(x)`	Adds element `x` to the set. If `x` is already present, it does nothing quietly.
`s.update(iterable)`	Adds multiple elements from an iterable (list, tuple, string, or another set) to the set `s`.
`s.remove(x)`	Removes element `x` from the set. **Raises a `KeyError`** if `x` is not found.
`s.discard(x)`	Removes element `x` from the set if it is present. **Does nothing (no error)** if `x` is not found. Ideal for safe removals.
`s.pop()`	Removes and returns an *arbitrary* (random) element from the set. Raises a `KeyError` if the set is empty.
`s.clear()`	Removes all elements from the set, leaving it completely empty.

Table 5.6: Methods to Modify Sets

```
my_set = {10, 20, 30}

# Adding elements
my_set.add(40)
my_set.update([50, 60, 70]) # Adding multiple elements at
    once
print(my_set) # {70, 40, 10, 50, 20, 60, 30} (Output order is
     unpredictable)

# Removing elements
my_set.remove(10)
# my_set.remove(99)    # This would crash the program with a
    KeyError!

my_set.discard(20)
my_set.discard(99)     # Safe: No error is raised even though
    99 is not in the set
```

```
14
15 removed_item = my_set.pop()
16 print(f"Popped item: {removed_item}")
17 print("Final set:", my_set)
```

Listing 5.39: Adding and removing elements

5.4.5 Set Relations

Sets can also be compared to check their relationships—whether one set encompasses another completely, or if they share no elements at all.

- **`issubset()`:** Returns `True` if all elements of the calling set are present in the argument set. Operator equivalent: `<=`.
- **`issuperset()`:** Returns `True` if the calling set contains all elements of the argument set. Operator equivalent: `>=`.
- **`isdisjoint()`:** Returns `True` if both sets have null intersection (absolutely no common elements).

```
1 X = {1, 2, 3}
2 Y = {1, 2, 3, 4, 5}
3 Z = {7, 8, 9}
4
5 print("Is X a subset of Y?     :", X.issubset(Y))        #
    True
6 print("Is Y a superset of X?   :", Y.issuperset(X))      #
    True
7 print("Are X and Z disjoint?   :", X.isdisjoint(Z))      #
    True
8 print("Are X and Y disjoint?   :", X.isdisjoint(Y))      #
    False (They share 1, 2, 3)
```

Listing 5.40: Set relational testing

5.4.6 Frozensets (Immutable Sets)

While a standard `set` is mutable, Python also provides a special variant called `frozenset`. A `frozenset` is exactly like a set, maintaining all the benefits of fast membership testing and uniqueness, except it is strictly *immutable*. Once created, you cannot add or remove elements from it.

Because they are immutable, `frozenset`s are hashable. This solves a crucial architectural problem: standard sets cannot be used as dictionary keys or as elements inside another set, but `frozenset`s can! This is heavily utilized in complex algorithms that need to cache sets or map sets of properties to certain values.

```
1 # Creating a frozenset
2 vowels = frozenset(['a', 'e', 'i', 'o', 'u'])
3
```

```
# vowels.add('y')  # AttributeError: 'frozenset' object has
    no attribute 'add'

# Frozensets can be used as dictionary keys
dictionary = {
    frozenset([1, 2]): "Group A",
    frozenset([3, 4]): "Group B"
}
print(dictionary[frozenset([1, 2])])  # Output: Group A
```

Listing 5.41: Using frozenset

5.4.7 Set Comprehensions

Similar to list comprehensions and dictionary comprehensions, Python supports set comprehensions. It is a concise, expressive way to dynamically create sets using a single line of code, naturally ensuring that the generated elements are unique and efficiently stored.

```
# Create a set of squares for even numbers between 0 and 9
# Notice the curly braces used here instead of square
    brackets
squares = {x**2 for x in range(10) if x % 2 == 0}
print(squares)  # {0, 64, 4, 36, 16}
```

Listing 5.42: Set comprehension

PYQ: List set functions and operations; develop a Python program to demonstrate set functions and operations. (Su'23 OR, Su'25, W'25)

```
# Create two sets
A = {10, 20, 30, 40, 50}
B = {30, 40, 50, 60, 70}

print("Set A:", A)
print("Set B:", B)
print("-" * 30)

# 1. Mathematical operations
print("Union (A|B)          :", A | B)
print("Intersection (A&B)   :", A & B)
print("Difference (A-B)     :", A - B)
print("Sym Diff (A^B)       :", A ^ B)
print("-" * 30)

# 2. Set methods for modification
A.add(100)
print("After add(100) to A  :", A)

A.discard(10)
```

```
print("After discard(10) A  :", A)

# 3. Built-in functions & Relations
print("Is {30,40} subset of A?", {30, 40}.issubset(A))
print("Is A superset of {30,40}?", A.issuperset({30,
    40}))
print("Length of Set A (len):", len(A))
print("Is 70 present in B?  :", 70 in B)
```

Listing 5.43: Set operations comprehensive program

PYQ: Develop code for two sets and perform union, intersection, difference, and symmetric difference operations. (W'24)

```
# Define two sets with some overlapping elements
set1 = {1, 2, 3, 4, 5, 6}
set2 = {4, 5, 6, 7, 8, 9}

print(f"Set 1: {set1}")
print(f"Set 2: {set2}\n")

# Perform and display operations
print("Union              (set1 | set2):", set1 | set2)
      # {1,2,3,4,5,6,7,8,9}
print("Intersection       (set1 & set2):", set1 & set2)
      # {4,5,6}
print("Difference         (set1 - set2):", set1 - set2)
      # {1,2,3}
print("Difference         (set2 - set1):", set2 - set1)
      # {7,8,9}
print("Symmetric Diff     (set1 ^ set2):", set1 ^ set2)
      # {1,2,3,7,8,9}
```

Listing 5.44: All four core set mathematical operations

PYQ: Write a program to demonstrate set functions and operations. (Su'24 OR)

```
colours = {"red", "blue", "green", "yellow"}

# Demonstrating Addition and Removal
colours.add("purple")
print("After adding 'purple' :", colours)

colours.remove("yellow")
print("After removing 'yellow':", colours)

```

```
# Using discard avoids errors if the element doesn't exist
colours.discard("orange")
print("After discarding 'orange' (not present):", colours)

# Demonstrating Fast Membership Testing
print("\nMembership Test:")
print("Is 'red' in set?    :", "red" in colours)
print("Is 'yellow' in set? :", "yellow" in colours)

# Iterating over a Set (Order is not guaranteed)
print("\nNumber of colours:", len(colours))
print("Listing all colours:")
for c in colours:
    print(" -", c)
```

Listing 5.45: Set functions and membership

5.5 Dictionaries

5.5.1 Introduction to Dictionaries

Definition: Dictionary

A **dictionary** (`dict`) in Python is a fundamental, highly optimized, built-in data structure that stores data as a collection of **key-value pairs**. It is an *ordered* (as of Python 3.7), *mutable*, and dynamic collection. Each element in a dictionary is a pair consisting of a unique **key** that maps to a specific **value**. They are formally known in computer science as associative arrays, hash maps, or hash tables.

To understand dictionaries intuitively, think of a real-world language dictionary or a telephone directory. In a language dictionary, you look up a word (the *key*) to find its definition (the *value*). In a telephone directory, you search for a person's name (the *key*) to find their phone number (the *value*). You do not search by a numerical index (like "page 50"), but rather by the unique identifier (the word or the name).

Similarly, in Python, whereas lists and tuples use sequential, zero-based integer indices (0, 1, 2, ...) to access their elements, dictionaries use arbitrary keys. This provides a direct "mapping" from a meaningful key to its associated value.

Under the hood, Python dictionaries are implemented using highly optimized **hash tables**. This implementation ensures that core dictionary operations—like looking up a value, adding a new key-value pair, or deleting an existing pair—are exceptionally fast. In Big O notation, these operations generally take $O(1)$ (constant) average time complexity. This performance characteristic makes dictionaries the absolute ideal data structure when dealing with large datasets where rapid retrieval based on a unique identifier is crucial, far outperforming lists which would require $O(N)$ linear scans.

5.5.2 Key Properties of Dictionaries

Before diving into the syntax and usage of dictionaries, it is absolutely essential to understand their core characteristics and the rules governing them:

- **Key-Value Structure:** Data is strictly stored as pairs. A key acts as a unique identifier, and the value is the actual data associated with that identifier. You cannot have a key without a value, or a value without a key.
- **Key Uniqueness:** All keys within a single dictionary must be strictly **unique**. A dictionary cannot contain duplicate keys. If you attempt to assign a value to an already existing key, the dictionary will not create a second pair; instead, the old value will be silently overwritten and replaced by the new value.
- **Key Immutability (Hashability):** Keys must be of an **immutable** (unchangeable) data type. Valid types for keys include strings, integers, floats, booleans, and tuples (provided the tuple contains only immutable elements). This rule exists because dictionaries use the key's memory hash to determine where to store the value. If a key were a mutable type like a list, changing the list later would change its hash, making the dictionary completely lose track of where the associated

value is stored. Thus, mutable types like lists, sets, or other dictionaries trigger a `TypeError` if used as keys.

- **Value Flexibility:** Values, unlike keys, are incredibly flexible and can be of **any** Python data type whatsoever. A value can be a primitive type like an integer or a string, but it can also be a complex data structure like a list, a set, another nested dictionary, a custom object, or even a function. Furthermore, values do not need to be unique; multiple entirely different keys can point to the exact same value.
- **Mutability:** Dictionaries themselves are completely mutable objects. This means their size and contents can change dynamically throughout the execution of a program. You can freely add new key-value pairs, update the values of existing keys, and remove pairs entirely after the dictionary has initially been constructed.
- **Ordering (Insertion Order):** The behavior of dictionary ordering has changed over Python's history. Prior to Python 3.6, dictionaries were fundamentally unordered collections. Iterating over a dictionary would yield items in an unpredictable, semi-random order based on internal hash tables. However, starting with an internal memory optimization in CPython 3.6, and becoming an officially guaranteed language specification in Python 3.7+, dictionaries now strictly maintain **insertion order**. This means that when you iterate through a dictionary, the key-value pairs will always be returned in the exact chronological order in which they were initially added to the dictionary.

5.5.3 Creating Dictionaries

Python offers several different syntactical approaches to create dictionaries, catering to various programming scenarios and data sources.

Using Curly Braces `{}`

The most common, direct, and readable way to instantiate a dictionary is by using curly braces. You enclose a comma-separated sequence of key-value pairs inside the braces. A colon (`:`) separates each individual key from its associated value.

```
# Creating a completely empty dictionary
empty_dict = {}

# Creating a dictionary with string keys and mixed value
    types
student = {
    "first_name": "Alice",
    "last_name": "Smith",
    "age": 20,
    "roll_no": "101A",
    "is_graduated": False,
    "marks": [85, 90, 78] # The value is a list of integers
}

# Creating a dictionary mapping integers to their squares
squares_map = {1: 1, 2: 4, 3: 9, 4: 16}
```

```
16
17 print(student)
```

Listing 5.46: Creating dictionaries with curly braces

Using the `dict()` Constructor Function

The built-in `dict()` constructor function provides alternative, programmatic ways to create dictionaries. It is particularly elegant when your keys are simple strings, as it allows you to construct the dictionary using keyword arguments, omitting the need for quotation marks around the keys.

```
1  # Creating an empty dictionary
2  another_empty = dict()
3
4  # 1. Using keyword arguments (keys must be valid Python
       variable names)
5  db_config = dict(host="localhost", port=5432, user="admin",
       password="pwd")
6
7  # 2. From an iterable of tuples (each tuple must contain
       exactly two items: a key and a value)
8  pairs_list = [("apple", 50), ("banana", 30), ("cherry", 100)]
9  fruit_inventory = dict(pairs_list)
10
11 # 3. Using zip() to magically combine two separate parallel
       lists into a single dictionary
12 columns = ["id", "name", "email"]
13 row_data = [1, "John Doe", "john@example.com"]
14 user_record = dict(zip(columns, row_data))
15
16 print(db_config)
17 print(user_record)
```

Listing 5.47: Creating dictionaries with the dict() constructor

Output

```
{'host': 'localhost', 'port': 5432, 'user': 'admin', 'password': 'pwd'}
{'id': 1, 'name': 'John Doe', 'email': 'john@example.com'}
```

5.5.4 Accessing Dictionary Items

Once a dictionary is populated with data, you require mechanisms to retrieve that data. Python provides two primary techniques for accessing values, each handling the scenario of "missing keys" very differently.

Square Bracket Notation (`dict[key]`)

The most straightforward and widely used method to access a value is by placing its corresponding key inside square brackets `[]`, immediately following the dictionary's variable name.

While fast and readable, this method is "unsafe". If you attempt to access a key that does not currently exist within the dictionary, Python will instantly raise a `KeyError` exception. If this exception is not caught and handled (using a `try-except` block), it will fatally crash your program.

```
employee_record = {"emp_id": 105, "name": "John Doe", "
    department": "Finance"}

# Accessing existing keys works perfectly
print("Employee Name:", employee_record["name"])
print("Employee ID:", employee_record["emp_id"])

# The following line is commented out because it would crash
    the program
# Python raises a KeyError because the exact string 'salary'
    is not a registered key
# print(employee_record["salary"])
```

Listing 5.48: Accessing values using square brackets

The `get()` Method (`dict.get(key, default)`)

To write robust, crash-resistant code, it is highly recommended to use the `get()` method when dealing with dynamic data where you are unsure if a key exists.

The `get()` method attempts to retrieve the value for a given key. If the key exists, it returns the value exactly like the square bracket notation. However, if the key is completely absent, it gracefully returns a default value instead of raising an error. By default, this fallback value is the special Python object `None`. You can optionally provide a second argument to specify a custom fallback value.

```
settings = {"theme": "dark", "font_size": 14}

# Key exists: returns 'dark'
print("Theme:", settings.get("theme"))

# Key missing (no custom default): returns None
print("Notifications enabled:", settings.get("notifications")
    )

# Key missing (with custom default): returns 'Arial'
print("Font Family:", settings.get("font_family", "Arial"))
```

Listing 5.49: Safely accessing values using the get() method

5.5.5 Adding and Updating Items

Because dictionaries are mutable, their state can evolve. The operations for inserting a brand new pair and modifying an existing pair share the exact same syntax.

Using the Assignment Operator

You assign a value to a key using the standard square bracket notation combined with the equals sign (=). The dictionary handles the logic automatically:

- **Update:** If the key already exists in the dictionary, its current value is destroyed and overwritten by the newly assigned value.
- **Add:** If the key does not currently exist, a brand new key-value pair is created and appended to the dictionary.

```
user_profile = {"username": "cyber_ninja", "level": 15}

# 1. Adding a completely new key-value pair
user_profile["email"] = "ninja@example.com"
print("After addition:", user_profile)

# 2. Updating the value of an existing key
user_profile["level"] = 16
print("After level up:", user_profile)
```

Listing 5.50: Adding and updating elements with direct assignment

Using the `update()` Method

While direct assignment is great for single elements, the `update()` method is a powerful tool allowing you to bulk-add or bulk-update multiple key-value pairs simultaneously. It takes another dictionary (or an iterable of key-value tuples, or keyword arguments) and merges it into the original dictionary. Any overlapping keys in the original dictionary will be overwritten by the new values from the incoming dictionary.

```
user_profile = {"username": "cyber_ninja", "level": 15}

# Updating with a dictionary of new incoming data
incoming_data = {"level": 20, "score": 5000, "status": "
    online"}
user_profile.update(incoming_data)

# Updating using keyword arguments directly inside the
    function call
user_profile.update(region="North America", is_premium=True)

print("Fully updated profile:", user_profile)
```

Listing 5.51: Updating multiple items using the update() method

5.5.6 Removing Items

Depending on whether you need to use the data being removed or just want to destroy it, Python provides several different deletion mechanisms.

- **The `del` keyword:** A built-in Python keyword that removes the pair completely. It does not return the deleted value. It is strictly fatal and raises a `KeyError` if you attempt to delete a non-existent key.
- **The `pop(key[, default])` method:** Removes the pair corresponding to the given key and profoundly **returns its value**. This allows you to capture the removed data into a variable. If the key is missing, it raises a `KeyError` unless you provide a `default` parameter, which allows it to fail gracefully.
- **The `popitem()` method:** Removes and returns the **last inserted** key-value pair as a tuple `(key, value)`. This implements a LIFO (Last-In, First-Out) behavior. Prior to Python 3.7, it removed a seemingly random, arbitrary pair. It is useful for destructively iterating through a dictionary from back to front.
- **The `clear()` method:** An aggressive method that instantly wipes out the entire contents of the dictionary, leaving behind a completely empty dictionary object \{\}. The dictionary structure remains in memory, but its size becomes zero.

```
stock = {"laptops": 50, "mice": 150, "keyboards": 100, "
    monitors": 30}

# 1. Using del keyword (Destructive, returns nothing)
del stock["mice"]
print("After del:", stock)

# 2. Using pop() (Removes and returns the value)
# Let's say a customer buys all the monitors
sold_monitors = stock.pop("monitors")
print(f"Sold {sold_monitors} monitors. Remaining stock:",
    stock)

# Using pop() with a default value to prevent program crashes
tablets = stock.pop("tablets", 0) # Tablets don't exist,
    safely returns 0
print(f"Tried to pop tablets, got: {tablets}")

# 3. Using popitem() (Removes the very last inserted element)
last_item_tuple = stock.popitem()
print(f"Popped last item {last_item_tuple}. Remaining stock:"
    , stock)

# 4. Using clear() (Purges everything)
stock.clear()
print("After calling clear():", stock) # Output: {}
```

Listing 5.52: Various techniques for removing items from a dictionary

5.5.7 Iterating Over a Dictionary

Iteration involves programmatically visiting every element in a data structure. Because dictionaries are inherently a composite of two distinct parts (keys and values), you have maximum flexibility over how you traverse them.

Iterating Over Keys

When you place a dictionary directly in a `for` loop header, Python implicitly iterates over the dictionary's **keys**. You can also achieve this explicitly by calling the `keys()` method, though most Python developers prefer the implicit approach for brevity.

```
student_grades = {"Alice": "A", "Bob": "B+", "Carol": "A-"}

# Implicit iteration (Pythonic, preferred)
print("Students:")
for name in student_grades:
    print(f"- {name}")
```

Listing 5.53: Looping strictly over dictionary keys

Iterating Over Values

If you care only about the data payload and not the identifiers, you can iterate strictly over the values by using the `values()` method.

```
student_grades = {"Alice": "A", "Bob": "B+", "Carol": "A-"}

print("Grades awarded:")
for grade in student_grades.values():
    print(grade)
```

Listing 5.54: Looping strictly over dictionary values

Iterating Over Key-Value Pairs

The most robust, common, and idiomatic way to iterate over a dictionary is by using the `items()` method. This method yields a tuple containing both the key and the value on each iteration. You can use Python's "tuple unpacking" feature to assign these tuple elements directly into two separate loop variables simultaneously.

```
student_grades = {"Alice": "A", "Bob": "B+", "Carol": "A-"}

# Unpacking the tuple into 'student_name' and 'letter_grade'
for student_name, letter_grade in student_grades.items():
    print(f"{student_name} achieved a grade of: {letter_grade
   }")
```

Listing 5.55: Simultaneously looping over keys and values

5.5.8 Dictionary Views

The specialized methods `keys()`, `values()`, and `items()` do not return standalone lists. Instead, they return **dictionary view objects** (`dict_keys`, `dict_values`, `dict_items`).

These view objects are highly memory efficient and, crucially, they are **dynamic**. This means they are inherently linked to the original dictionary. If the original dictionary is modified (items added or removed), the view object will immediately and automatically reflect these changes without needing to be recreated.

While views are iterable (you can run a `for` loop over them), they are not subscriptable (they do not support integer indexing like `view[0]`). If you strictly require index-based access, you must force the view to consume memory and convert it into a static list using the `list()` constructor.

```
config = {"theme": "light", "language": "en"}
keys_view = config.keys()

print("Initial keys view:", keys_view)

# We modify the original dictionary by adding a totally new pair
config["auto_save"] = True

# The keys_view variable magically updates itself to include 'auto_save'!
print("Updated keys view:", keys_view)

# To use indexing, we convert the dynamic view into a static list
static_keys_list = list(keys_view)
print("The first key in the list is:", static_keys_list[0])
```

Listing 5.56: Demonstrating the dynamic nature of dictionary views

5.5.9 Advanced Dictionary Operations and Merging

Membership Testing (`in` Operator)

You can definitively verify whether a specific key exists within a dictionary using the `in` and `not in` logical operators. Because dictionaries are powered by hash tables, this membership test is blindingly fast, operating in constant $O(1)$ time regardless of whether the dictionary contains ten items or ten million items.

It is critical to note that the `in` operator only searches the **keys** by default. It does not search the values.

```
server_settings = {"port": 443, "ssl": True, "max_connections": 500}

# Testing if keys exist (Extremely fast O(1))
print("Is 'ssl' configured?", "ssl" in server_settings)       # True
print("Is 'timeout' configured?", "timeout" in server_settings)  # False

# If you MUST check for a value, you have to search the values view.
# Warning: This is a slow O(N) operation as it must scan every value sequentially.
print("Is the value 443 present?", 443 in server_settings.values()) # True
```

Listing 5.57: High-performance membership testing

Getting Dictionary Length

To calculate the total volume of data in the dictionary, pass it into the built-in `len()` function. This will return the exact total count of key-value pairs. Like membership testing, length checking is an $O(1)$ operation.

```
dataset = {"x": 10.5, "y": -3.2, "z": 8.0, "w": 0.0}
print("Total number of coordinates:", len(dataset)) # Output: 4
```

Modern Dictionary Merging (Python 3.9+)

Historically, merging dictionaries required using the `update()` method or complex unpacking syntax like `\{**dict1, **dict2\}`. To improve code readability and mathematical elegance, Python 3.9 formally introduced the merge operator (`|`) and the augmented assignment update operator (`|=`).

These operators work similarly to set union. If conflicting, identical keys exist in both dictionaries, the value from the dictionary on the right-hand side of the operator "wins" and overwrites the value from the left.

```
defaults = {"theme": "light", "font": "Arial", "volume": 50}
user_overrides = {"theme": "dark", "volume": 80, "notifications": True}

# The Merge operator (|) creates a brand new, third dictionary
# The original 'defaults' and 'user_overrides' remain completely untouched
final_config = defaults | user_overrides
print("Final config:", final_config)
# Result: {'theme': 'dark', 'font': 'Arial', 'volume': 80, 'notifications': True}

# The Update operator (|=) modifies the dictionary on the left directly in-place
defaults |= user_overrides
print("Modified defaults:", defaults)
# Result: 'defaults' now contains the merged data.
```

Listing 5.58: Merging dictionaries using modern pipe operators

5.5.10 Dictionary Methods Summary

The following reference table outlines the comprehensive suite of built-in methods designed specifically for interacting with dictionary objects.

Method	Description
`d.clear()`	Aggressively purges and removes all items, leaving an empty dictionary.
`d.copy()`	Returns a fresh shallow copy of the dictionary. Changes to the copy's top-level structure won't affect the original.
`d.fromkeys(seq[, val])`	Class method that constructs a new dictionary using elements from sequence `seq` as keys, all initialized to `val` (which defaults to `None`).
`d.get(key[, default])`	Safely retrieves the value for `key`. If the key is totally absent, it intercepts the error and returns `default` instead.
`d.items()`	Generates and returns a dynamic view object of the dictionary's `(key, value)` tuple pairs.
`d.keys()`	Generates and returns a dynamic view object of the dictionary's underlying keys.
`d.pop(key[, default])`	Locates `key`, permanently removes it from the dictionary, and returns its data value.
`d.popitem()`	Removes and returns the very last inserted `(key, value)` pair (LIFO queue behavior).
`d.setdefault(key[, def])`	A hybrid method. If `key` exists, it acts like `get()` and returns the value. If `key` is absent, it automatically inserts `key` mapped to `def`, and then returns `def`.
`d.update([other])`	Performs a bulk update, merging key/value pairs from `other` iterable or dictionary, ruthlessly overwriting existing keys.
`d.values()`	Generates and returns a dynamic view object exposing only the dictionary's values.

Table 5.7: Comprehensive Reference of Built-in Python Dictionary Methods

5.5.11 Dictionary Comprehensions

Just as list comprehensions provide a highly optimized, single-line syntax for generating lists, Python provides **dictionary comprehensions** for dynamically assembling dictionaries. They represent the "Pythonic" approach, replacing bulky, multi-line `for` loops and `.update()` calls with a single, elegant line of code that executes significantly faster under the CPython interpreter.

A dictionary comprehension allows you to iterate over any existing sequence or iterable, process the items, and construct a new key-value pair for each iteration. You can also append an `if` clause to filter which items are allowed into the final dictionary.

General Syntax:

```
{key_expression: value_expression for item in iterable if condition}
```

```python
# 1. Mathematical Generation: Creating a mapping of numbers to their cubes
cubes = {x: x**3 for x in range(1, 6)}
print("Cubes mapping:", cubes)

# 2. Data Filtering: Removing failing students from a grading system
exam_scores = {"Alice": 45, "Bob": 85, "Carol": 30, "Dave": 90, "Eve": 72}
# Only include students who scored 50 or higher
passing_students = {name: score for name, score in exam_scores.items() if score >= 50}
print("Passing students:", passing_students)

# 3. String Manipulation: Creating a character frequency/ASCII map
word = "COMPREHENSION"
# Map each unique character to its mathematical ASCII integer value
ascii_map = {char: ord(char) for char in set(word)}
print("ASCII values for characters in word:", ascii_map)

# 4. Complex Logic: Using conditional (ternary) expressions for values
numbers = {"A": 10, "B": 15, "C": 20, "D": 25}
# Categorize numbers as 'Even' or 'Odd'
parity_map = {key: ("Even" if val % 2 == 0 else "Odd") for key, val in numbers.items()}
print("Parity Categorization:", parity_map)
```

Listing 5.59: Powerful data generation using dictionary comprehensions

5.5.12 Nested Dictionaries

Dictionaries are not limited to storing simple text or numbers. Because dictionary values can be of any data type whatsoever, a dictionary value can itself be another fully-featured dictionary. This concept is referred to as a **nested dictionary**.

Nested dictionaries are incredibly potent and are the standard structural choice for representing deep, multi-level, structured, hierarchical data. They behave virtually identically to JSON (JavaScript Object Notation), which is the universal language of modern web APIs. Understanding nested dictionaries is a prerequisite for advanced web development and data science in Python.

```python
# A complex database representing employees, their personal info, and contact details
company_directory = {
    "emp_001": {
        "first_name": "Sarah",
        "last_name": "Connor",
        "role": "Security Analyst",
```

```
        "contact": {
            "email": "sarah.c@company.com",
            "phone": "555-0100"
        }
    },
    "emp_002": {
        "first_name": "John",
        "last_name": "Smith",
        "role": "Software Engineer",
        "contact": {
            "email": "john.s@company.com",
            "phone": "555-0200"
        }
    }
}

# 1. Deep Access: Reaching into multiple nested levels using chained brackets
# Let's extract John Smith's phone number
johns_phone = company_directory["emp_002"]["contact"]["phone"]
print(f"John's Direct Line: {johns_phone}")

# 2. Deep Iteration: Using nested loops to unpack the data structure
print("\n--- Full Company Directory ---")
for emp_id, details in company_directory.items():
    print(f"[{emp_id}] {details['first_name']} {details['last_name']} - {details['role']}")
    # Iterate through the nested 'contact' dictionary
    for contact_method, contact_value in details["contact"].items():
        print(f"    -> {contact_method.capitalize()}: {contact_value}")
```

Listing 5.60: Constructing and traversing complex nested dictionaries

5.5.13 Practical Application Scenarios

Because of their immense flexibility and high performance, dictionaries are heavily utilized across all domains of Python programming. Common scenarios include:

1. **Counting Frequency Maps:** Dictionaries are the perfect tool for calculating the frequency of elements in an array or characters in a document. You iterate through the data, using the data element as the key, and incrementing an integer counter as the value.

2. **Data Grouping and Aggregation:** You can categorize raw data points by a shared attribute. The shared attribute becomes the key, and the value is a list of all data points belonging to that category.

3. **High-Speed Lookups (Memoization/Caching):** In complex algorithms, you can use dictionaries to cache the results of incredibly expensive, time-consuming mathematical function calls. If the function is called again with the identical inputs (the key), you simply return the pre-calculated result (the value) from the dictionary instantly.

4. **Configuration Management:** Large software applications use nested dictionaries to load, store, and modify application settings, user preferences, and environment variables.

5. **Web Development & API Integration:** As mentioned, dictionaries map perfectly to JSON format. When your Python code interacts with external web services (like fetching weather data, processing financial transactions, or sending text messages), the incoming and outgoing data payloads are heavily processed using dictionaries via the `json` module.

5.5.14 Previous Year Questions (PYQs)

PYQ: Write a program to demonstrate dictionary functions and operations. (Su'24 OR, Su'25 OR)

This program serves as a comprehensive overview, demonstrating the entire lifecycle of a dictionary from creation to accessing, dynamic updating, safe deletion, and iterative inspection.

```
# Step 1: Initialize an empty dictionary
student_record = {}

# Step 2: Dynamically add varied data types
student_record["name"] = "Raj Patel"        # String
student_record["roll_no"] = 42              # Integer
student_record["enrolled"] = True           # Boolean
student_record["grades"] = [85, 90, 88]     # List

# Step 3: Iterate and cleanly display contents
print("=== Initial Student Record ===")
for attr, value in student_record.items():
    print(f"{attr.upper():<10}: {value}")

# Step 4: Safely access potentially missing elements
   using get()
print("\n[Accessing Data]")
print("Student Name:", student_record["name"])
# 'scholarship' key doesn't exist, so get() uses our
   fallback string
print("Scholarship Status:", student_record.get("
   scholarship", "Not Applied"))

# Step 5: Update an existing attribute
student_record["roll_no"] = 1042
```

```python
print("\n[Updating Data]")
print("Updated Roll No:", student_record["roll_no"])

# Step 6: Use pop() to cleanly remove and capture data
removed_grades = student_record.pop("grades")
print("\n[Removing Data]")
print(f"Extracted and removed grades: {removed_grades}")
print(f"Remaining attributes in record: {list(
    student_record.keys())}")

# Step 7: Verify presence using membership operators
print("\n[Operations]")
print("Is 'name' an active attribute?", "name" in
    student_record)
print("Total number of stored attributes:", len(
    student_record))
```

Listing 5.61: Comprehensive demonstration of dictionary operations

PYQ: Explain Dictionary Built-in Functions with examples. (W'25, W'24 OR)

Dictionaries feature a robust, built-in standard library of functions allowing profound manipulation of their underlying data structures. The most critical built-in methods include:
1. View Generation (`keys()`, `values()`, `items()`): These methods create highly efficient, dynamic, real-time "windows" into the dictionary's structure, returning iterable sequences of keys, data payloads, or tuple pairs respectively. **2. Safe Retrieval (`get(key, default)`):** A crash-resistant extraction method that attempts to fetch the value for a key, falling back to an optional default if the key is structurally missing. **3. Bulk Modification (`update(dict2)`):** Aggressively fuses a secondary dictionary into the primary one, enforcing an overwrite policy for any conflicting overlapping keys. **4. Fallback Insertion (`setdefault(key, default)`):** A conditional hybrid method. It functions identical to `get()` if the key exists. If missing, it physically inserts the new key-value pair before returning the newly placed value. **5. Controlled Deletion (`pop(key)`, `popitem()`):** `pop()` precisely targets a specific key for deletion, returning its value. `popitem()` indiscriminately removes the absolute last element chronologically added.

```python
# Initialize base dictionary
configuration = {"alpha": 100, "beta": 200, "gamma":
    300}

# 1. View Generation
print("Dictionary Keys   :", list(configuration.keys()))
print("Dictionary Values :", list(configuration.values()
    ))
print("Dictionary Pairs  :", list(configuration.items())
```

```
    )

# 2. Safe Retrieval
print("Fetch Delta       :", configuration.get("delta",
    "Missing Value!"))

# 3. Bulk Modification
# We add 'delta' and overwrite the value of 'alpha'
configuration.update({"delta": 400, "alpha": 999})
print("Post-Update State :", configuration)

# 4. Fallback Insertion
# 'epsilon' is completely new, so it gets injected into
    the dictionary
configuration.setdefault("epsilon", 500)
# 'beta' already already exists and is 200, so it
    completely ignores the 999
configuration.setdefault("beta", 999)
print("Post-SetDefault   :", configuration)

# 5. Controlled Deletion
val = configuration.pop("alpha")
print(f"Popped Key 'alpha', retrieved value: {val}")

last_entry = configuration.popitem()
print(f"Popped the most recent chronological entry: {
    last_entry}")

# 6. Total Annihilation
configuration.clear()
print("Post-Clear State  :", configuration) # Output is
    totally empty {}
```

Listing 5.62: Demonstrating critical built-in dictionary functions

PYQ: Develop a Python program to create a Dictionary with the roll number, name and marks of n students; display the names of students who have scored marks above 70/75. (W'25, Su'24)

This program effectively integrates dictionaries, nested dictionary structures, and algorithmic loops. It establishes a centralized database where a student's unique Roll Number acts as the primary key, mapping to a secondary nested dictionary containing their human-readable name and academic marks.

```
# Initialize the master database to harbor student
    profiles
academic_database = {}

# Interactively determine the size of the dataset
```

```
total_students = int(input("Enter the total number of students to process: "))

# Phase 1: Data Ingestion
for index in range(total_students):
    print(f"\n--- Constructing Profile for Student {index + 1} ---")
    roll_number = int(input("Assign Unique Roll Number: "))
    student_name = input("Enter Full Legal Name: ")
    academic_marks = float(input("Enter Academic Marks Achieved: "))

    # Store the complex profile payload as a nested dictionary mapped to the Roll Number
    academic_database[roll_number] = {
        "name": student_name,
        "marks": academic_marks
    }

# Phase 2: Complete Database Dump
print("\n" + "="*45)
print(f"{'ROLL NO':<12} | {'STUDENT NAME':<18} | {'MARKS':<8}")
print("="*45)
for roll, profile in academic_database.items():
    # Use formatted string literals to perfectly align the output columns
    print(f"{roll:<12} | {profile['name']:<18} | {profile['marks']:<8.1f}")

# Phase 3: Targeted Algorithmic Filtering (Threshold > 75)
print("\n--- Identifying Elite Performers (Marks strictly > 75) ---")
performers_found = False

# Iterate through every profile looking for qualifying marks
for roll, profile in academic_database.items():
    if profile["marks"] > 75:
        print(f"[*] Awarding Distinction to: {profile['name']} "
              f"(Roll: {roll}, Score: {profile['marks']})")
        performers_found = True

# Fallback mechanism if the entire cohort fails to meet the threshold
if not performers_found:
```

```
41      print("[!] Alert: No students in this batch
        successfully crossed the 75-mark threshold.")
```

Listing 5.63: Algorithmic filtering of a nested student database

Output

```
Enter the total number of students to process: 3

--- Constructing Profile for Student 1 ---
Assign Unique Roll Number: 101
Enter Full Legal Name: Alice Smith
Enter Academic Marks Achieved: 82.5

--- Constructing Profile for Student 2 ---
Assign Unique Roll Number: 102
Enter Full Legal Name: Bob Johnson
Enter Academic Marks Achieved: 68.0

--- Constructing Profile for Student 3 ---
Assign Unique Roll Number: 103
Enter Full Legal Name: Charlie Brown
Enter Academic Marks Achieved: 91.0

=============================================
ROLL NO      | STUDENT NAME       | MARKS
=============================================
101          | Alice Smith        | 82.5
102          | Bob Johnson        | 68.0
103          | Charlie Brown      | 91.0

--- Identifying Elite Performers (Marks strictly > 75) ---
[*] Awarding Distinction to: Alice Smith (Roll: 101, Score: 82.5)
[*] Awarding Distinction to: Charlie Brown (Roll: 103, Score: 91.0)
```