Python (DI01016011) - Problem Solver

Antigravity AI

June 4, 2026

Python

First Edition: 2026

Author: Antigravity AI
Website: milav.in
YouTube: @milav.dabgar

To my students,
who constantly inspire me to find new analogies,
break down complex theories, and make learning
an engineered science.

Preface

About this Book

Welcome to **Computer Networks**! This textbook is meticulously designed to serve as a comprehensive problem solver and study companion. It provides a robust and intuitive foundation in the core principles of this subject, bridging the gap between theoretical models and practical applications. A unique feature of this book is its focus on decoding complex concepts using clear, step-by-step methodologies and real-world analogies.

Target Audience

This book is intended for students and professionals looking to master the concepts of Computer Networks. Whether you are revising for exams, seeking to solidify your fundamental understanding, or looking for practical examples to clarify abstract concepts, this book has been structured to support your learning journey. It serves as an accessible guide designed to remove the fear of complex topics, replacing it with an appreciation for the underlying mechanics.

Scope and Coverage

The coverage is divided logically to follow standard academic curriculums. Each chapter focuses on key topics, breaking them down into digestible sections:

- **Theoretical Insights** – Core concepts explained intuitively.
- **Method Blocks** – Standardized, step-by-step procedures for solving specific types of problems.
- **Worked Examples** – Fully solved problems that apply the methods in practice.

We hope this resource proves invaluable in your studies and helps you achieve excellence in **Computer Networks**. Happy learning!

Acknowledgments

Writing a comprehensive book that connects the fundamental forces of Chemistry with practical engineering applications requires more than just academic knowledge—it requires a community of support. I would like to express my sincere gratitude to everyone who contributed to the realization of this project.

Specially, I extend my heartfelt gratitude to the **Department of Technical Education (DTE), Government of Gujarat**, and **Government Polytechnic, Palanpur**, for providing an environment that fosters innovation, academic rigor, and continuous professional development.

I am deeply thankful to my family for their unwavering patience and support during the countless hours spent researching, formatting, and refining these chapters.

Finally, my deepest appreciation goes to my students. Your curiosity, challenging questions, and continuous feedback are the true driving force behind the unique analogies and streamlined structure of this book. This textbook was engineered for you.

Antigravity AI

About the Author

Milav Jayeshkumar Dabgar is an engineering educator and R&D professional with over 9 years of experience in electronics hardware development, embedded systems, AI/ML, and full-stack software engineering. He currently serves as a **Lecturer (Class - II) & Innovation Leader** at Government Polytechnic, Palanpur, under the Education Department of the Government of Gujarat.

Milav holds a Master of Engineering (ME) in Communication Systems from L.D. College of Engineering and a Bachelor of Engineering (BE) in Electronics & Communication. He is also currently pursuing a **BS in Data Science and Applications from IIT Madras**, where he has completed both the **Diploma in Programming** and the **Diploma in Data Science** with distinction.

Most recently, he completed the **AICTE QIP PG Certificate Program in Deep Learning from SVNIT Surat** with a **perfect 10/10 CGPA**, topping his batch.

A dedicated mentor, Milav has guided numerous student teams in securing over **INR 45 lakhs** in innovation funding, including INR 25 lakhs from Shark Tank India and INR 20 lakhs through iHub Gujarat, resulting in two student patents. He is recognized as an **NPTEL Evangelist** and **NPTEL Discipline Star** in Computer Science, reflecting his commitment to continuous learning and academic excellence.

His technical expertise spans from embedded systems (8051, STM32, Arduino) to modern web technologies (Next.js, Python, PostgreSQL) and Data Science (TensorFlow, PyTorch). Through this textbook, he aims to simplify complex Engineering Chemistry concepts by integrating relatable, real-world analogies drawn from modern tech and engineering landscapes.

Milav is also an active content creator, running a popular **YouTube channel** (@milav.dabgar) where he shares technical tutorials and academic resources. He maintains a personal blog and resource hub at milav.in and can be reached for professional collaborations on LinkedIn.

Contents

List of Figures

List of Tables

CHAPTER 1

Problem Solving using Flowchart and Algorithm

1.1 Introduction, Steps for problem-solving, Algorithm

Problem solving is the systematic process of identifying a problem and developing a clear, step-by-step computational solution. In computer science, this process is essential before writing any code to ensure accuracy and efficiency.

Methodology:

Steps for Problem Solving 1. **Understand the Problem:** Clearly identify the inputs required and the expected outputs. 2. **Devise a Plan:** Determine the logical steps, formulas, or conditions needed to transform the inputs into the correct outputs. 3. **Execute the Plan:** Formulate the solution using an Algorithm, Flowchart, or Pseudocode. 4. **Review and Refine:** Trace the logic with sample test cases to verify correctness and handle edge cases.

Methodology:

Writing an Algorithm An algorithm is a finite sequence of precise, step-by-step instructions to solve a problem. 1. Start the algorithm with the word "START" or "Step 1: Start". 2. Read the necessary inputs using keywords like "READ" or "INPUT". 3. Write formulas or logical operations sequentially to process the data. 4. Output the result using keywords like "PRINT", "WRITE", or "OUTPUT". 5. End the algorithm with the word "STOP" or "END".

Worked Example:

Algorithm to Calculate Area of a Circle **Problem:** Write an algorithm to compute the area of a circle given its radius.
Step 1: Start
Step 2: Read radius R
Step 3: Calculate Area $= 3.14159 \times R \times R$
Step 4: Print Area
Step 5: Stop

Worked Example:

Algorithm to Check if a Number is Even or Odd **Problem:** Write an algorithm to check whether an integer is even or odd.
Step 1: Start
Step 2: Read integer N
Step 3: If $N \pmod 2 == 0$ then
Print "N is Even"
Else

Print "N is Odd"
Step 4: Stop

1.2 Flowchart

A flowchart is a visual and graphical representation of an algorithm. It uses standard geometric shapes to represent different types of instructions and arrows to define the execution flow.

Methodology:

Flowchart Symbols and Usage

- **Oval (Terminal):** Indicates the Start or Stop points of the flowchart.
- **Parallelogram (Input/Output):** Represents reading inputs from the user or displaying output results.
- **Rectangle (Process):** Represents arithmetic operations, data assignments, and calculations.
- **Diamond (Decision):** Represents a conditional test (e.g., $A > B$). It has one entry point and typically two exit points (Yes/No or True/False).
- **Arrows (Flow lines):** Connect symbols to indicate the logical sequence of operations.

Worked Example:

Flowchart to Find the Largest of Two Numbers **Problem:** Determine the logic for a flowchart that inputs two distinct numbers and prints the larger one.
Graphical Logic Sequence:

1. **Terminal (Oval):** Start
2. **Input (Parallelogram):** Read values for A and B
3. **Decision (Diamond):** Check condition $A > B$?
   - **If Yes path:** Flow to Output (Parallelogram) $\rightarrow$ Print "A is largest"
   - **If No path:** Flow to Output (Parallelogram) $\rightarrow$ Print "B is largest"
4. **Terminal (Oval):** Stop (Both paths merge to this terminal)

Worked Example:

Flowchart for Simple Interest Calculation **Problem:** Calculate Simple Interest for given Principal (P) Rate (R) and Time (T).
Graphical Logic Sequence:

1. **Terminal (Oval):** Start
2. **Input (Parallelogram):** Read P, R, T
3. **Process (Rectangle):** Compute $SI = (P \times R \times T)/100$
4. **Output (Parallelogram):** Print SI
5. **Terminal (Oval):** Stop

1.3 Pseudocode

Pseudocode is an informal, language-independent way of writing algorithms. It blends natural language with structural conventions of programming languages without requiring strict syntax adherence.

Methodology:

Writing Pseudocode

- Use clear, descriptive action words (e.g., `INPUT`, `COMPUTE`, `SET`, `PRINT`).
- Use standard mathematical operators ($+$, $-$, $\times$, $\div$, $\pmod{}$).
- Use capitalization for keywords to distinguish them from variables.
- Use indentation to clearly show the scope of loops (`WHILE...END WHILE`) and conditional statements (`IF...ELSE...END IF`).

Worked Example:

Pseudocode to Find the Factorial of a Number **Problem:** Write pseudocode to compute the factorial of a given positive integer N ($N! = 1 \times 2 \times \cdots \times N$).

```
START
  INPUT N
  SET Factorial = 1
  SET i = 1

  WHILE i <= N DO
    COMPUTE Factorial = Factorial * i
    COMPUTE i = i + 1
  END WHILE

  PRINT Factorial
END
```

Worked Example:

Algorithm and Pseudocode for Prime Number Check **Problem:** Determine whether a given integer $N > 1$ is a prime number.

Pseudocode:

```
START
  INPUT N
  SET IsPrime = True
  SET i = 2

  WHILE i < N DO
    IF N MOD i == 0 THEN
      SET IsPrime = False
      BREAK
    END IF
    SET i = i + 1
  END WHILE

  IF IsPrime == True THEN
    PRINT "N is a Prime Number"
  ELSE
```

```
    PRINT "N is Not a Prime Number"
  END IF
END
```

CHAPTER 2

Basics of Python

2.1 Introduction to Python

Python is a high-level interpreted programming language renowned for its clear syntax and readability. It is widely used for computational problem-solving, automation, data analysis, and web development.

Methodology:

Executing a Python Program

1. Write the source code in a plain text editor and save the file with a `.py` extension (e.g. `program.py`).
2. Open the system terminal or command prompt.
3. Navigate to the directory containing the file.
4. Run the program using the command `python program.py`.

Worked Example:

Hello World Program Write a basic program to display a greeting message on the screen.
Solution:

```
# This is a basic Python program
print("Hello World")
```

Output:

```
Hello World
```

2.2 Python Installation

Methodology:

Installing Python on a Computer

1. Open a web browser and navigate to the official Python website downloads page.
2. Download the installer executable corresponding to your operating system.
3. Run the downloaded installer.
4. **Crucial Step**: Ensure the checkbox labeled "Add Python to PATH" or "Add python.exe to PATH" is checked before proceeding.
5. Click "Install Now" and wait for the setup wizard to finish.
6. Verify the installation by opening a terminal and typing `python -version`.

2.3 Basic Structure Comments Keywords Identifiers and Variables

A Python script is a sequence of statements executed sequentially. It utilizes indentation to define code blocks instead of curly braces.

Methodology:

Variables Identifiers and Keywords

1. **Variables**: A variable is created the moment you first assign a value to it using the `=` operator. Explicit type declaration is not required.
2. **Identifiers**: Names given to variables, functions, or classes. They must start with a letter (a-z A-Z) or an underscore (_) and can be followed by letters, numbers, or underscores.
3. **Keywords**: Reserved words (such as `if`, `else`, `while`, `for`, `def`, `return`) that have special meaning and cannot be used as identifiers.
4. **Comments**: Text ignored by the interpreter. Use the `#` symbol for single-line comments.

Worked Example:

Variable Assignment and Printing Write a program to assign the length and width of a rectangle to variables and print them.
Solution:

```
# Assign values to valid identifiers
rectangle_length = 10
rectangle_width = 5.5

# Print the values using the print() function
print("Length:", rectangle_length)
print("Width:", rectangle_width)
```

Output:

```
Length: 10
Width: 5.5
```

2.3.1 Data Types

Methodology:

Basic Built-in Data Types

1. **Numeric Types**: `int` (whole numbers), `float` (floating-point decimal numbers), and `complex` (numbers with real and imaginary parts).
2. **Sequence Types**: `str` (strings representing text data), `list` (ordered and mutable collection of items), and `tuple` (ordered and immutable collection).
3. **Boolean Type**: `bool` (logical values representing `True` or `False`).
4. **Dictionary**: `dict` (unordered collection of key-value pairs).

2.3.2 Operators

Methodology:

Applying Basic Operators

1. **Arithmetic Operators**: Used for mathematical calculations. `+` (Addition), `-` (Subtraction), `*` (Multiplication), `/` (Division yielding a float), `//` (Floor Division yielding an integer), `%` (Modulus or remainder), `**` (Exponentiation).
2. **Relational Operators**: Used to compare values, returning a boolean. `==` (Equal to), `!=` (Not equal to), `>` (Greater than), `<` (Less than), `>=` (Greater than or equal to), `<=` (Less than or equal to).
3. **Logical Operators**: Used to combine conditional statements. `and` (True if both operands are true), `or` (True if at least one operand is true), `not` (Reverses the logical state).
4. **Assignment Operators**: Used to assign values to variables. `=`, `+=`, `-=`, `*=`, `/=`.

Worked Example:

Arithmetic Operations Write a program to calculate the area and perimeter of a rectangle using arithmetic operators.

Solution:

```
length = 10
width = 5

# Arithmetic calculations
area = length * width
perimeter = 2 * (length + width)

print("Area of rectangle:", area)
print("Perimeter of rectangle:", perimeter)
```

Output:

```
Area of rectangle: 50
Perimeter of rectangle: 30
```

Worked Example:

Relational and Logical Operations Write a program to check if a student has passed an exam. A student passes only if marks in both Math and Science are greater than or equal to 40.
Solution:

```
math_marks = 85
science_marks = 35

# Relational and Logical evaluation
has_passed = (math_marks >= 40) and (science_marks >= 40)

print("Has the student passed?", has_passed)
```

Output:

```
Has the student passed? False
```

2.4 Type Conversion

Methodology:

Implicit and Explicit Type Conversion

1. **Implicit Conversion**: Python automatically converts a smaller or simpler data type to a larger or more complex data type to prevent data loss. For example, adding an `int` and a `float` automatically results in a `float`.
2. **Explicit Conversion**: Also known as Type Casting. The programmer manually converts the data type of an object to a required data type using built-in functions like `int()`, `float()`, or `str()`.

Worked Example:

Implicit Type Conversion Write a program to add an integer and a float, and observe the data type of the resulting value.
Solution:

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

# Implicit conversion of num_int to float during addition
result = num_int + num_float

print("Result:", result)
print("Data type of result:", type(result))
```

Output:

```
Result: 12.5
Data type of result: <class 'float'>
```

Worked Example:

Explicit Type Conversion Write a program to take a numerical value stored as a string, convert it to an integer, add another integer to it, and print the total.
Solution:

```
string_val = "50"
```

```
integer_val = 20

# Explicit conversion of string to integer
converted_val = int(string_val)

# Now addition can be performed mathematically
total = converted_val + integer_val

print("Total sum:", total)
```

Output:

```
Total sum: 70
```

CHAPTER 3
Flow of Control

3.1 Introduction to Flow of Control

Flow of control determines the order in which statements are executed in a program. By default, Python executes code sequentially from top to bottom. However, control flow statements allow you to alter this sequential execution based on conditions or to repeat a block of code multiple times. The primary categories of control flow are Selection (decision-making) and Repetition (looping).

3.2 Selection Statements

Selection statements allow the program to choose different execution paths based on boolean conditions.

3.2.1 If Elif and Else

Methodology:

Using If Elif and Else Statements To execute code blocks based on multiple conditions:

1. Start with an `if` statement followed by a condition and a colon.
2. Indent the block of code to be executed if the condition is `True`.
3. Optionally add one or more `elif` (else if) statements to test additional conditions if previous ones are `False`.
4. Optionally end with an `else` statement to provide a default block of code if all preceding conditions evaluate to `False`.

Worked Example:

Grade Evaluation **Problem Statement:** Write a Python snippet to evaluate a student's grade based on their marks. Marks ≥ 80 is 'A', ≥ 60 is 'B', ≥ 40 is 'C', and otherwise 'F'. Assume `marks = 75`.
Step 1: Define the marks variable.

```
marks = 75
```

Step 2: Apply the if-elif-else logic.

```
if marks >= 80:
    grade = 'A'
elif marks >= 60:
    grade = 'B'
elif marks >= 40:
    grade = 'C'
else:
    grade = 'F'
```

Step 3: Execution trace.

- Condition `marks >= 80` ($75 \geq 80$) is `False`.
- Condition `marks >= 60` ($75 \geq 60$) is `True`.
- The variable `grade` is assigned the string `'B'`.
- The rest of the block is skipped.

Final Result: The grade evaluated is 'B'.

3.2.2 Nested If Statements

Methodology:

Using Nested If Statements To perform multi-level decision making:

1. Place an `if` statement inside another `if` or `else` block.
2. Ensure correct indentation for the inner `if` statements to define their scope.
3. The inner `if` condition is evaluated only if the outer `if` condition is `True`.

Worked Example:

Largest of Three Numbers **Problem Statement:** Find the largest among three numbers: $a = 15$, $b = 25$, and $c = 20$ using nested `if` statements.
Step 1: Define the variables.

```
a = 15
b = 25
c = 20
```

Step 2: Apply nested if logic.

```
if a >= b:
    if a >= c:
        largest = a
    else:
        largest = c
else:
    if b >= c:
        largest = b
    else:
        largest = c
```

Step 3: Execution trace.

- Outer condition `a >= b` ($15 \geq 25$) is `False`.
- Control jumps to the outer `else` block.
- Inner condition `b >= c` ($25 \geq 20$) is `True`.
- The variable `largest` is assigned the value of `b` (25).

Final Result: The largest number is 25.

3.2.3 Match Case Statement

Introduced in Python 3.10, the `match` statement provides a cleaner way to write multiple condition checks against a single variable, similar to a switch-case statement in other languages.

Methodology:

Using Match Case Statement To perform pattern matching:

1. Use the `match` keyword followed by the variable or expression to evaluate, and a colon.
2. Write `case` blocks for each pattern you want to check, followed by a colon.
3. Indent the code to execute for a matching case.
4. Use `case _:` as the default fallback (wildcard) if no other cases match.

Worked Example:

Simple Arithmetic Calculator **Problem Statement:** Evaluate an arithmetic operation based on an operator character. `num1 = 10`, `num2 = 5`, and `operator = '*'`.
Step 1: Write the match-case logic.

```
num1 = 10
num2 = 5
operator = '*'

match operator:
    case '+':
        result = num1 + num2
    case '-':
        result = num1 - num2
    case '*':
        result = num1 * num2
    case '/':
        result = num1 / num2
    case _:
        result = "Invalid Operator"
```

Step 2: Execution trace.

- The variable `operator` is matched against the patterns.
- `'+'` does not match `'*'`.
- `'-'` does not match `'*'`.
- `'*'` matches exactly.
- `result = 10 * 5 = 50` is executed.

Final Result: The result is 50.

3.3 Repetition Statements

Repetition statements (loops) allow executing a block of code multiple times.

3.3.1 For Loop

Methodology:

Using For Loop with Range To iterate a specific number of times:

1. Use the syntax: `for variable in range(start, stop, step):`.
2. `start` (inclusive, default 0) is the starting integer.
3. `stop` (exclusive) is the ending boundary.
4. `step` (default 1) determines the increment or decrement.
5. The indented code block executes once for each value generated by `range()`.

Worked Example:

Sum of First N Natural Numbers **Problem Statement:** Calculate the sum of the first 5 natural numbers using a `for` loop.
Step 1: Initialize the sum variable.

```
total_sum = 0
```

Step 2: Write the for loop.

```
for i in range(1, 6):
    total_sum = total_sum + i
```

Step 3: Execution trace.

- Iteration 1: `i = 1`, `total_sum = 0 + 1 = 1`
- Iteration 2: `i = 2`, `total_sum = 1 + 2 = 3`
- Iteration 3: `i = 3`, `total_sum = 3 + 3 = 6`
- Iteration 4: `i = 4`, `total_sum = 6 + 4 = 10`
- Iteration 5: `i = 5`, `total_sum = 10 + 5 = 15`
- Loop terminates as the range ends before 6.

Final Result: The `total_sum` is 15.

3.3.2 While Loop

Methodology:

Using While Loop To loop as long as a condition remains true:

1. Initialize a loop control variable before the loop.
2. Use the `while` keyword followed by a boolean condition and a colon.
3. Write the indented code block to execute.
4. Update the loop control variable inside the loop to avoid infinite loops.

Worked Example:

Reversing an Integer **Problem Statement:** Reverse the digits of the number `n = 456`.
Step 1: Initialize variables.

```
n = 456
reversed_n = 0
```

Step 2: Write the while loop.

```
while n > 0:
    digit = n % 10
    reversed_n = (reversed_n * 10) + digit
    n = n // 10
```

Step 3: Execution trace.

- **Iteration 1:** `n > 0` $(456 > 0)$ is True.
  - `digit = 456 % 10 = 6`
  - `reversed_n = (0 * 10) + 6 = 6`
  - `n = 456 // 10 = 45`
- **Iteration 2:** `n > 0` $(45 > 0)$ is True.
  - `digit = 45 % 10 = 5`
  - `reversed_n = (6 * 10) + 5 = 65`
  - `n = 45 // 10 = 4`
- **Iteration 3:** `n > 0` $(4 > 0)$ is True.
  - `digit = 4 % 10 = 4`
  - `reversed_n = (65 * 10) + 4 = 654`
  - `n = 4 // 10 = 0`
- **Iteration 4:** `n > 0` $(0 > 0)$ is False. Loop terminates.

Final Result: The reversed number is 654.

3.3.3 Nested Loops

Methodology:

Using Nested Loops To execute a loop inside another loop:

1. Write an outer loop (`for` or `while`).
2. Inside the outer loop's block, write an inner loop.
3. The inner loop runs completely from start to finish for every single iteration of the outer loop.

Worked Example:

Printing a Number Triangle **Problem Statement:** Print a right-angled triangle of numbers with 3 rows.

```
1
1 2
1 2 3
```

Step 1: Write the nested loops.

```
for i in range(1, 4):          # Outer loop for rows
    for j in range(1, i + 1): # Inner loop for columns
        print(j, end=" ")
    print()                    # Move to the next line
```

Step 2: Execution trace.

- **Outer Iteration 1 (`i=1`):**
  - Inner loop runs for `j` in `range(1, 2)` → `j=1`. Prints "1 ".
  - Inner loop ends. Prints newline.
- **Outer Iteration 2 (`i=2`):**
  - Inner loop runs for `j` in `range(1, 3)` → `j=1, 2`. Prints "1 2 ".
  - Inner loop ends. Prints newline.
- **Outer Iteration 3 (`i=3`):**
  - Inner loop runs for `j` in `range(1, 4)` → `j=1, 2, 3`. Prints "1 2 3 ".
  - Inner loop ends. Prints newline.

3.4 Loop Control Statements

Python provides statements to alter the regular flow of loops.

3.4.1 Break Continue and Pass

Methodology:

Using Break and Continue To control loop execution dynamically:

1. **`break`:** Instantly terminates the nearest enclosing loop. Execution resumes at the first statement following the loop.
2. **`continue`:** Skips the remaining statements in the current iteration of the nearest enclosing loop and immediately jumps to the next iteration's condition check or loop update.

Worked Example:

Filtering and Searching in a Loop **Problem Statement:** Iterate through numbers 1 to 5. If the number is 2, skip it. If the number is 4, stop the loop completely.
Step 1: Write the loop with continue and break.

```
for num in range(1, 6):
    if num == 2:
        continue
    if num == 4:
        break
    print(num)
```

Step 2: Execution trace.

- **num = 1:** Conditions are False. Prints `1`.
- **num = 2:** `num == 2` is True. `continue` executes. The print statement is skipped. Moves to next iteration.

- **num = 3:** Conditions are False. Prints `3`.
- **num = 4:** `num == 2` is False. `num == 4` is True. `break` executes. The loop is immediately terminated.

Final Output:

```
1
3
```

Methodology:

Using Pass Statement To write empty blocks of code:

1. Place the `pass` keyword where a statement is syntactically required but no action is needed.
2. It acts as a null operation; nothing happens when it is executed.

Worked Example:

Placeholder with Pass **Problem Statement:** Write a loop that iterates through numbers 1 to 3, but leave the code block empty to be implemented later.
Step 1: Use the pass keyword.

```
for i in range(1, 4):
    pass
```

Step 2: Execution trace.

- The loop executes 3 times.
- Each time, the `pass` statement does absolutely nothing.
- No error is thrown for having an empty block.

CHAPTER 4

Functions

4.1 Introduction to Functions

A function is a reusable block of code designed to perform a specific task. Functions help in breaking our program into smaller and modular chunks. As our program grows larger and more complex, functions make it more organized and manageable. Python has many built-in functions like `print()` and `len()`, but you can also create your own functions.

4.2 User Defined Functions

Methodology:

Defining and Calling a Function To create a function in Python, follow these steps:

1. Use the `def` keyword followed by the function name and parentheses `()`.
2. Inside the parentheses, specify any parameters (arguments) the function should accept.
3. End the line with a colon `:`.
4. Indent the block of code that makes up the function body.
5. Use the `return` statement to send a result back to the caller (optional).
6. To execute the function, call it by its name followed by parentheses containing any required arguments.

Worked Example:

Calculate Factorial of a Number Write a user-defined function to calculate the factorial of a given integer.
Solution:

1. Define a function `factorial(n)`.
2. Initialize a variable `result = 1`.
3. Loop from 1 to `n` (inclusive) and multiply `result` by the loop variable.
4. Return `result`.

```
def factorial(n):
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result

# Calling the function
num = 5
```

```
print("Factorial of", num, "is", factorial(num))
```

Output:

```
Factorial of 5 is 120
```

Worked Example:

Check for Prime Number Write a function that takes an integer and returns `True` if it is a prime number and `False` otherwise.
Solution:

1. Define `is_prime(n)`.
2. If `n <= 1`, return `False`.
3. Loop `i` from 2 to the integer part of the square root of `n`.
4. If `n % i == 0`, return `False`.
5. If the loop finishes without returning, return `True`.

```
import math

def is_prime(n):
    if n <= 1:
        return False
    for i in range(2, int(math.sqrt(n)) + 1):
        if n % i == 0:
            return False
    return True

# Calling the function
print(is_prime(11))  # True
print(is_prime(15))  # False
```

Output:

```
True
False
```

4.3 Scope of a Variable

The scope of a variable determines the portion of the program where that variable can be accessed.

Methodology:

Variable Scope Rules

1. **Local Scope:** Variables defined inside a function are local to that function. They cannot be accessed outside of it.
2. **Global Scope:** Variables defined outside of all functions are global. They can be accessed anywhere in the file.
3. **Global Keyword:** To modify a global variable inside a function, you must declare it using the `global` keyword before modifying it.

Worked Example:

Understanding Local and Global Scope Demonstrate the difference between local and global variables, and how to modify a global variable inside a function.
Solution:

```
count = 10  # Global variable

def my_function():
    local_var = 5  # Local variable
    print("Inside function local_var =", local_var)
    print("Inside function global count =", count)

def update_global():
    global count
    count += 5
    print("Inside update_global count is updated to", count)

my_function()
update_global()
print("Outside functions count =", count)
```

Output:

```
Inside function local_var = 5
Inside function global count = 10
Inside update_global count is updated to 15
Outside functions count = 15
```

4.4 Python Standard Library

Python comes with a rich standard library containing modules that provide pre-written code for various tasks, such as mathematical operations, file I/O, and random number generation.

Methodology:

Importing and Using Modules To use functions from a standard library module:

1. **Import entire module:** `import module_name`. Access functions using `module_name.function_name()`.
2. **Import specific functions:** `from module_name import func1 func2`. Access functions directly as `func1()`.
3. **Import with alias:** `import module_name as alias`. Access functions using `alias.function_name()`.

Worked Example:

Using Math and Random Modules Write a program that uses the `math` module to compute the greatest common divisor (GCD) of two numbers and the `random` module to generate a random number between 1 and 100.
Solution:

```
import math
import random
```

```
# Using math module
num1 = 48
num2 = 18
gcd_value = math.gcd(num1, num2)
print("GCD of", num1, "and", num2, "is:", gcd_value)

# Using random module
rand_num = random.randint(1, 100)
print("Random number between 1 and 100:", rand_num)
```

Output:

```
GCD of 48 and 18 is: 6
Random number between 1 and 100: 42
```

(Note: The random number will vary on each execution).

Worked Example:

Using Datetime Module Write a program to display the current date and time using the `datetime` module.
Solution:

```
from datetime import datetime

# Get current date and time
now = datetime.now()

# Format the output
formatted_time = now.strftime("%Y-%m-%d %H:%M:%S")
print("Current Date and Time:", formatted_time)
```

Output:

```
Current Date and Time: 2026-06-04 08:45:59
```

CHAPTER 5
Strings and Lists

5.1 Introduction to Strings

In Python, strings are ordered sequences of character data. They are immutable, meaning that once created, their elements cannot be changed. Strings can be enclosed in single, double, or triple quotes. Understanding how to index and slice strings is fundamental for text processing.

Methodology:

String Indexing and Slicing 1. **Indexing:** Elements are accessed using square brackets `[]`. Python supports zero-based positive indexing (0 to length-1) and negative indexing (-1 for the last character, -2 for the second last, etc.). 2. **Slicing:** Slicing extracts a substring using the syntax `string[start:stop:step]`. 3. `start` is the starting index (inclusive). If omitted, it defaults to 0. 4. `stop` is the ending index (exclusive). If omitted, it defaults to the length of the string. 5. `step` defines the stride length. If omitted, it defaults to 1. A negative step reverses the string. 6. **Immutability:** Operations like `s[0] = 'a'` will result in a `TypeError`.

Worked Example:

Extracting Substrings using Slicing Given the string `text = "PythonProgramming"`.

1. Extract the first 6 characters.
2. Extract "Programming".
3. Reverse the entire string.
4. Extract every alternate character from the string.

Solution:

1. **First 6 characters:** Using slicing `text[0:6]`, we get characters from index 0 up to 5.

   `Result = "Python"`

2. **Extract "Programming":** The word starts at index 6 and goes to the end. Using `text[6:]`.

   `Result = "Programming"`

3. **Reverse the string:** Omitting start and stop, and using a step of -1, i.e., `text[::-1]`.

   `Result = "gnimmargorPnohtyP"`

4. **Alternate characters:** Using a step of 2, i.e., `text[::2]`.

   `Result = "PtoPormig"`

5.2 Strings Methods

Python provides a rich set of built-in methods for string manipulation. Since strings are immutable, these methods always return a new string rather than modifying the original one.

Methodology:

Common String Methods 1. **Case Conversion:**

- `s.upper()`: Converts all lowercase characters to uppercase.
- `s.lower()`: Converts all uppercase characters to lowercase.
- `s.title()`: Capitalizes the first letter of every word.

2. **Searching and Replacing:**

- `s.find(sub)`: Returns the lowest index where substring `sub` is found. Returns -1 if not found.
- `s.count(sub)`: Returns the number of non-overlapping occurrences of `sub`.
- `s.replace(old, new)`: Returns a copy with all occurrences of substring `old` replaced by `new`.

3. **Splitting and Joining:**

- `s.split(sep)`: Returns a list of the words in the string, using `sep` as the delimiter.
- `sep.join(iterable)`: Joins elements of an iterable (like a list) into a single string using `sep` as a separator.

4. **Stripping Whitespace:**

- `s.strip()`: Removes leading and trailing whitespace.

Worked Example:

Formatting and Processing Text Data Consider the unformatted log entry: `log = "  error:  file not found  "`. Perform the following sequence of operations to clean and reformat the log:

1. Remove the leading and trailing spaces.
2. Replace "error" with "WARNING".
3. Convert the entire string to uppercase.
4. Split the string into a list of words.

Solution:

1. **Strip spaces:** `step1 = log.strip()`
   `step1` becomes `"error:  file not found"`
2. **Replace text:** `step2 = step1.replace("error", "WARNING")`
   `step2` becomes `"WARNING: file not found"`
3. **Uppercase:** `step3 = step2.upper()`
   `step3` becomes `"WARNING: FILE NOT FOUND"`
4. **Split into words:** `step4 = step3.split(" ")`
   `step4` becomes `["WARNING:", "FILE", "NOT", "FOUND"]`

5.3 Introduction to List

A list in Python is an ordered, mutable, and heterogeneous collection of elements. Lists are defined by enclosing elements in square brackets `[]`, separated by commas.

Methodology:

List Creation and Access 1. **Creation:** Lists can be created empty `[]` or with elements `[1, 2, "three"]`. They can contain elements of different data types, including other lists. 2. **Accessing Elements:** Just like strings, lists support 0-based indexing and negative indexing. 3. **Mutability:** Elements can be modified directly using their index, e.g., `my_list[0] = 10`. 4. **Slicing:** Sub-lists can be extracted using `list[start:stop:step]`. Slicing a list returns a new list. 5. **Length:** The built-in `len(list)` function returns the number of elements in the list.

Worked Example:

Creating and Modifying Lists Given a list `scores = [85, 90, 78, 92, 88]`.

1. Update the third element to 80.
2. Extract the top 3 scores assuming they are the last three elements after a hypothetical sort (extract elements from index 2 to end).
3. Change the first two scores to 86 and 91 using slice assignment.

Solution:

1. **Update the third element:** The third element is at index 2. `scores[2] = 80`
   The list is now `[85, 90, 80, 92, 88]`.
2. **Extract elements:** Use slicing from index 2 to the end. `top_scores = scores[2:]`
   `top_scores` is `[80, 92, 88]`.
3. **Slice assignment:** Assign an iterable to a slice. `scores[0:2] = [86, 91]`
   The original list becomes `[86, 91, 80, 92, 88]`.

5.4 List Methods

Python lists possess a variety of built-in methods that allow for dynamic resizing and manipulation. These methods mostly modify the list in-place and return `None`.

Methodology:

Operations and Methods on Lists 1. **Adding Elements:**

- `lst.append(x)`: Adds a single item `x` to the end of the list.
- `lst.insert(i, x)`: Inserts an item `x` at a given index `i`.
- `lst.extend(iterable)`: Appends all items from an iterable to the list.

2. **Removing Elements:**

- `lst.remove(x)`: Removes the first item whose value is `x`. Raises a `ValueError` if not found.
- `lst.pop([i])`: Removes and returns the item at the given index `i`. If no index is specified, it removes and returns the last item.
- `lst.clear()`: Removes all items from the list.

3. **Ordering Elements:**

- `lst.sort(key=None, reverse=False)`: Sorts the items of the list in place.
- `lst.reverse()`: Reverses the elements of the list in place.

4. **Information:**
   - `lst.index(x)`: Returns the index of the first item whose value is `x`.
   - `lst.count(x)`: Returns the number of times `x` appears in the list.

Worked Example:

Managing a Collection of Data using List Methods Start with an empty list `tasks = []` and perform the following sequence:

1. Append "Read Chapter 1".
2. Append "Write Code".
3. Insert "Setup Environment" at the beginning of the list.
4. Extend the list with `["Test Code", "Deploy"]`.
5. Remove "Write Code" from the list.
6. Pop the last element.

Solution:

1. `tasks.append("Read Chapter 1")` → `["Read Chapter 1"]`
2. `tasks.append("Write Code")` → `["Read Chapter 1", "Write Code"]`
3. `tasks.insert(0, "Setup Environment")` → `["Setup Environment", "Read Chapter 1", "Write Code"]`
4. `tasks.extend(["Test Code", "Deploy"])` → `["Setup Environment", "Read Chapter 1", "Write Code", "Test Code", "Deploy"]`
5. `tasks.remove("Write Code")` → `["Setup Environment", "Read Chapter 1", "Test Code", "Deploy"]`
6. `tasks.pop()` returns `"Deploy"`. The list becomes → `["Setup Environment", "Read Chapter 1", "Test Code"]`

5.5 List as Arguments

When a list is passed as an argument to a function, the function receives a reference to the original list object. This means lists are mutable, and any in-place modifications made to the list inside the function will affect the original list passed by the caller.

Methodology:

Passing Lists to Functions 1. **Pass by Object Reference:** Variables holding lists are references to the list object in memory. Passing a list to a function passes this reference. 2. **In-Place Modification:** Using methods like `append()`, `remove()`, or modifying elements via indexing (e.g., `lst[0] = x`) inside the function alters the original list. 3. **Reassignment:** If the list parameter is assigned a completely new list inside the function (e.g., `lst = [1, 2]`), this breaks the link to the original list, and the caller's list remains unchanged. 4. **Preventing Modification:** If you want to protect the original list from changes,

pass a copy of the list using slicing (e.g., `my_function(my_list[:])`) or the `list()` constructor.

Worked Example:

Writing Functions that Modify Lists Write a function `square_elements(lst)` that takes a list of numbers and squares each element in place. Also, show how to call this function and observe the original list.
Solution:

```
def square_elements(lst):
    # Iterate through the list using indices to modify elements in place
    for i in range(len(lst)):
        lst[i] = lst[i] ** 2

# Caller code
numbers = [2, 3, 4, 5]
print("Before:", numbers)

# Pass the list to the function
square_elements(numbers)

print("After:", numbers)
```

Execution Trace:

1. Initially, `numbers` references the list `[2, 3, 4, 5]`.
2. `square_elements(numbers)` is called, and the parameter `lst` references the exact same list object.
3. The `for` loop accesses each index:
   - `i=0`: `lst[0] = 2 ** 2 = 4`. List is now `[4, 3, 4, 5]`.
   - `i=1`: `lst[1] = 3 ** 2 = 9`. List is now `[4, 9, 4, 5]`.
   - `i=2`: `lst[2] = 4 ** 2 = 16`. List is now `[4, 9, 16, 5]`.
   - `i=3`: `lst[3] = 5 ** 2 = 25`. List is now `[4, 9, 16, 25]`.
4. Because `lst` and `numbers` point to the same memory location, printing `numbers` after the function call outputs `[4, 9, 16, 25]`.

Appendix: Key Formulae

This appendix provides a quick reference to the key formulae and mathematical relationships discussed in this book.

Financial Mathematics

Simple Interest

$$SI = \frac{P \times R \times T}{100}$$

Where:

- SI is the Simple Interest.
- P is the Principal amount.
- R is the Rate of interest per annum.
- T is the Time period.

Combinatorics

Factorial

$$N! = 1 \times 2 \times \cdots \times N$$

Where:

- $N!$ is the factorial of a positive integer N.

Number Theory

Even Number Condition

$$N \pmod 2 = 0$$

Where:

- N is an integer. If the remainder when N is divided by 2 is 0, then N is even.

Geometry

Area of a Circle

$$A = \pi R^2$$

Where:

- A is the Area of the circle.
- R is the Radius of the circle.
- $\pi \approx 3.14159$