Python (4311601) - 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.

Contents

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.

CHAPTER 1

Problem Solving using Flowchart and Algorithm

1.1 Introduction to Problem Solving

Methodology:

Steps for Problem Solving To effectively solve computational problems using a computer system, use the following structured steps:

1. **Analyze the Problem:** Understand the given inputs, the expected outputs, and the constraints.
2. **Develop an Algorithm:** Write a step-by-step logical sequence of instructions to solve the problem.
3. **Design a Flowchart:** Create a visual representation of the algorithm using standard symbols.
4. **Write Pseudocode:** Translate the logic into a human-readable generic code structure.
5. **Implement the Code:** Write the actual program in a programming language like Python.
6. **Test and Debug:** Run the program with test cases and fix any logical or syntax errors.

Worked Example:

Applying Problem Solving Steps Apply the problem-solving steps to calculate the simple interest for a given principal, rate of interest, and time.

Step 1: Analyze the Problem

- Input: Principal (P), Rate of Interest (R), and Time in years (T).
- Output: Simple Interest (SI).
- Formula: $SI = (P \times R \times T)/100$.

Step 2: Develop an Algorithm

1. Read values for P, R, and T.
2. Calculate $SI = (P \times R \times T)/100$.
3. Display SI.

Step 3: Write Pseudocode

```
START
    READ P, R, T
    COMPUTE SI = (P * R * T) / 100
    PRINT "Simple Interest is", SI
END
```

1.2 Algorithms and Flowcharts

Methodology:

Algorithm Characteristics and Importance An **Algorithm** is a finite sequence of unambiguous instructions designed to perform a specific task or solve a problem.
Characteristics of an Algorithm:

- **Input:** It must accept zero or more well-defined inputs.
- **Output:** It must produce at least one well-defined output.
- **Definiteness:** Every step must be unambiguous, clear, and precisely defined.
- **Finiteness:** It must eventually terminate after a finite number of steps.
- **Effectiveness:** Each step must be basic and feasible enough to be executed manually or computationally.

Importance of Flowchart and Algorithm: They help in standardizing the logic, dividing complex problems into smaller, manageable steps, and reducing errors before actual coding begins. They act as a logical blueprint that can be translated into any programming language.

Worked Example:

Algorithm for Swapping Two Numbers Write an algorithm to swap the values of two variables A and B without using a third variable.
Solution:

1. Start.
2. Read values for A and B.
3. Compute $A = A + B$.
4. Compute $B = A - B$.
5. Compute $A = A - B$.
6. Print the swapped values of A and B.
7. Stop.

1.3 Symbolic Representation of a Flowchart

Methodology:

Flowchart Symbols and Flow of Control A flowchart visually represents the flow of control in an algorithm.
Standard Symbols:

Symbol Name	Shape	Purpose
Terminal	Oval / Pill	Indicates the Start or Stop of the flowchart.
Input / Output	Parallelogram	Represents taking input data or displaying output results.
Process	Rectangle	Used for mathematical calculations or data assignments.
Decision	Diamond	Used for condition checking or branching (Yes/No).
Flow Lines	Arrows	Shows the direction of the flow of control.
Connector	Circle	Connects different parts of a flowchart across pages.

Flow of Control: The sequence in which instructions are executed. The primary control structures include:

- **Sequential:** Step-by-step, linear execution of instructions.
- **Selection (Branching):** Execution path diverges based on a boolean condition (e.g., if-else logic).
- **Iteration (Looping):** Repeated execution of a specific block of instructions until a condition is met.

Limitations of Flowchart: They are difficult and time-consuming to draw for large, complex programs. Modifying a flowchart often requires redrawing it completely, and there is a lack of strict universal formatting for highly complex branching logic.

Worked Example:

Flowchart Logic for Checking Leap Year Trace the logical flowchart steps to determine if a given year is a leap year.
Solution (Step-by-Step Flowchart Translation):

1. **Start** (Oval shape).
2. **Input** Year Y (Parallelogram).
3. **Decision** (Diamond): Is $Y \pmod 4 == 0$?
   - **If No:** Flow points to **Output** "Not a Leap Year" (Parallelogram) → Go to Stop.
   - **If Yes:** Flow points to a nested **Decision** (Diamond): Is $Y \pmod{100} == 0$?
     - **If No:** Flow points to **Output** "Leap Year" (Parallelogram) → Go to Stop.
     - **If Yes:** Flow points to a nested **Decision** (Diamond): Is $Y \pmod{400} == 0$?
       - **If Yes:** Flow points to **Output** "Leap Year" (Parallelogram).
       - **If No:** Flow points to **Output** "Not a Leap Year" (Parallelogram).
4. **Stop** (Oval shape).

1.4 Problem Solving using Pseudocode

Methodology:

Constructing Pseudocode Pseudocode bridges the gap between algorithms and actual programming languages by using structural conventions of programming without adhering to strict syntax rules.
Common Syntax Conventions:

- `READ`, `INPUT`, or `GET` to gather data.
- `PRINT`, `WRITE`, or `DISPLAY` to show output.
- `COMPUTE`, `CALCULATE`, or `SET` to evaluate expressions or assign values.
- `IF condition THEN ... ELSE ... ENDIF` for conditional branching.
- `WHILE condition DO ... ENDWHILE` for loop control.
- `FOR loopVar = start TO end DO ... ENDFOR` for counter-controlled loops.

Worked Example:

Pseudocode for Fibonacci Series Write pseudocode to print the first N terms of the Fibonacci series. Each term is the sum of the two preceding ones, starting from 0 and 1.
Solution:

```
START
```

```
    PRINT "Enter number of terms:"
    READ N

    SET T1 = 0
    SET T2 = 1

    PRINT T1
    PRINT T2

    SET Counter = 3

    WHILE Counter <= N DO
        COMPUTE NextTerm = T1 + T2
        PRINT NextTerm
        SET T1 = T2
        SET T2 = NextTerm
        COMPUTE Counter = Counter + 1
    ENDWHILE
END
```

Worked Example:

Pseudocode for Finding Maximum in a List Write pseudocode to find the maximum number from a list of N given numbers.

Solution:

```
START
    PRINT "Enter total numbers:"
    READ N
    PRINT "Enter the first number:"
    READ MaxNumber

    SET Counter = 2

    WHILE Counter <= N DO
        PRINT "Enter next number:"
        READ CurrentNumber

        IF CurrentNumber > MaxNumber THEN
            SET MaxNumber = CurrentNumber
        ENDIF

        COMPUTE Counter = Counter + 1
    ENDWHILE

    PRINT "The maximum number is:", MaxNumber
END
```

CHAPTER 2

Python Introduction

2.1 Introduction to Python Features and Applications

Methodology:

Understanding Python and its Characteristics Python is a high-level interpreted programming language known for its simplicity and readability. Key characteristics include:

1. **Interpreted**: Code is executed line by line making debugging easier.
2. **Dynamically Typed**: Variable types are determined at runtime.
3. **Extensive Libraries**: Standard libraries for web development data analysis and more.
4. **Object-Oriented and Procedural**: Supports multiple programming paradigms.

Major applications involve:

1. Web Development (using frameworks like Django or Flask)
2. Data Science and Machine Learning (using Pandas or Scikit-learn)
3. Automation and Scripting
4. Software Development and Testing

2.2 Python Installation

Methodology:

Steps for Python Installation To execute Python programs the Python interpreter must be installed on your operating system:

1. Download the installer for your specific operating system from the official Python website (`python.org`).
2. Run the executable installer.
3. **Crucial Step**: Check the box that says "Add Python to PATH" before clicking Install Now. This ensures Python can be run from the command line.
4. Verify the installation by opening a command prompt or terminal and typing `python -version`.

2.3 Basic Structure Keywords Identifiers and Variables

Methodology:

Python Program Structure and Naming Rules A basic Python program consists of statements executed sequentially.

1. **Comments**: Lines starting with `#` are ignored by the interpreter and used for documentation.
2. **Keywords**: Reserved words (e.g. `if for while def class`) that cannot be used as variable names.
3. **Identifiers**: Names given to variables functions or classes.
   - Must start with a letter (a-z A-Z) or an underscore (_).
   - Can contain letters digits (0-9) and underscores.
   - Case-sensitive (`Var` and `var` are treated differently).
4. **Variables**: Created the moment a value is assigned using the `=` operator. No explicit type declaration is needed.

Worked Example:

Validating Python Identifiers Problem: Determine whether the following identifiers are valid or invalid in Python. Give reasons. 1) `my_var` 2) `2ndValue` 3) `total-sum` 4) `global` 5) `_private`
Solution:

1. `my_var`: **Valid**. Contains only letters and an underscore.
2. `2ndValue`: **Invalid**. Cannot start with a digit.
3. `total-sum`: **Invalid**. Hyphens (`-`) are not allowed; only underscores are permitted.
4. `global`: **Invalid**. `global` is a reserved keyword in Python.
5. `_private`: **Valid**. Identifiers can start with an underscore.

2.4 Data Types and Operators

Methodology:

Python Data Types and Operators Python supports various built-in data types and operators to manipulate data.

1. **Numeric Types**: `int` (integers) `float` (decimals) `complex` (real + imaginary).
2. **Sequence Types**: `str` (strings) `list` (ordered mutable) `tuple` (ordered immutable).
3. **Mapping Type**: `dict` (key-value pairs).
4. **Boolean**: `bool` (`True` or `False`).

Key Operators:

- **Arithmetic**: `+` (add) `-` (subtract) `*` (multiply) `/` (float division) `//` (floor division) `%` (modulus) `**` (exponentiation).
- **Relational**: `== != > < >= <=`.
- **Logical**: `and or not`.

Worked Example:

Evaluating Arithmetic and Relational Expressions Problem: Given `a = 15` and `b = 4`. Evaluate the following Python expressions: 1) `a / b` 2) `a // b` 3) `a % b` 4) `a ** 2` 5) `(a > 10) and (b != 4)`
Solution:

1. `a / b` = `15 / 4` = `3.75` (Standard float division yields a decimal)
2. `a // b` = `15 // 4` = `3` (Floor division truncates the decimal part)
3. `a % b` = `15 % 4` = `3` (Modulus gives the remainder of $15 \div 4$)
4. `a ** 2` = 15^2 = `225` (Exponentiation)
5. `(a > 10) and (b != 4)` = `(15 > 10) and (4 != 4)` = `True and False` = `False`

2.5 Type Conversion

Methodology:

Implicit and Explicit Type Conversion Type conversion (or typecasting) is the process of converting data of one type to another.

1. **Implicit Type Conversion**: Python automatically converts a smaller data type to a larger data type to prevent data loss (e.g. converting an integer to a float during addition).
2. **Explicit Type Conversion**: The user forces the conversion using built-in functions such as:
   - `int(x)`: Converts `x` to an integer.
   - `float(x)`: Converts `x` to a floating-point number.
   - `str(x)`: Converts `x` to a string representation.

Worked Example:

Demonstrating Type Conversion Problem: Given `num_int = 10` `num_str = "25"` and `num_float = 5.5`. 1) Perform addition of `num_int` and `num_float` and identify the type of the result. 2) Add `num_int` and `num_str` to produce an integer result.
Solution:
Step 1: Implicit Conversion

```
result1 = num_int + num_float
# result1 = 10 + 5.5 = 15.5
```

Python automatically converts the integer `10` to a float `10.0` before addition. The type of `result1` is `float`.
Step 2: Explicit Conversion Directly adding an `int` and a `str` will cause a `TypeError`. We must explicitly convert `num_str` to an integer first.

```
result2 = num_int + int(num_str)
# result2 = 10 + int("25")
# result2 = 10 + 25 = 35
```

The type of `result2` is `int`.

CHAPTER 3
Flow of Control

3.1 Introduction to Flow of Control

In Python programming, flow of control refers to the order in which individual statements, instructions, or function calls are executed or evaluated. By default, statements are executed sequentially from top to bottom. However, control flow statements such as selection (decision-making) and repetition (loops) allow us to alter this sequential execution to perform complex computations and solve problems efficiently.

3.2 Selection

3.2.1 If statement

The `if` statement is the simplest form of selection control flow. It evaluates a condition and executes a block of code only if the condition is true.

Methodology:

If Statement Execution **Algorithm:**

1. Formulate a boolean condition using comparison or logical operators.
2. Evaluate the condition.
3. If the condition evaluates to `True`, execute the indented statements immediately following the `if` clause.
4. If the condition evaluates to `False`, skip the indented statements and proceed to the next unindented line of code.

Syntax:

```
if condition:
    # statement(s) to execute if condition is True
```

Worked Example:

Checking Even Numbers **Problem Statement:** Write a Python snippet to determine if a given integer variable n is an even number. If it is, print a confirmation message.
Step-by-step Solution:

1. **Initialize the variable:** Let $n = 14$.
2. **Formulate the condition:** A number is even if the remainder when divided by 2 is 0. The condition is `n % 2 == 0`.
3. **Apply the if statement:**

```
n = 14
```

```
if n % 2 == 0:
    print("The number is even.")
```

4. **Execution trace:** Since $14 \pmod 2 = 0$, the condition evaluates to `True`, and the program prints `"The number is even."`.

3.2.2 Elif statement

The `elif` (else if) statement allows checking multiple conditions sequentially. It is used in conjunction with `if` and an optional `else` statement.

Methodology:

Elif Statement Logic **Algorithm:**

1. Evaluate the initial `if` condition. If `True`, execute its block and exit the entire selection structure.
2. If `False`, evaluate the next `elif` condition.
3. If an `elif` condition is `True`, execute its block and exit the structure.
4. Repeat step 2 and 3 for all `elif` conditions.
5. If no conditions are `True`, execute the `else` block (if provided).

Worked Example:

Grading System **Problem Statement:** Calculate the grade of a student based on their marks M. The grading criteria are: $M \geq 80$ (Distinction), $60 \leq M < 80$ (First Class), $40 \leq M < 60$ (Pass), $M < 40$ (Fail).
Step-by-step Solution:

1. **Initialize marks:** Let $M = 65$.
2. **Construct selection sequence:** Order conditions from highest to lowest criteria.

```
marks = 65
if marks >= 80:
    grade = "Distinction"
elif marks >= 60:
    grade = "First Class"
elif marks >= 40:
    grade = "Pass"
else:
    grade = "Fail"
print(grade)
```

3. **Execution trace:**
   - `marks >= 80` is `False`.
   - `marks >= 60` is `True`. The `grade` is set to "First Class".
   - The rest of the structure is skipped.

3.2.3 Nested if statement

A nested `if` statement is an `if` statement placed inside another `if` statement. It is used when a condition needs to be checked only if a previous condition is true.

Methodology:

Nested If Evaluation **Algorithm:**

1. Evaluate the outer `if` condition.
2. If the outer condition is `False`, skip the entire nested structure.
3. If the outer condition is `True`, enter its block and evaluate the inner `if` condition.
4. Execute the inner block only if both the outer and inner conditions evaluate to `True`.

Worked Example:

Leap Year Validation **Problem Statement:** Determine if a given year is a leap year. A year is a leap year if it is divisible by 4. However, if it is a century year (divisible by 100), it must also be divisible by 400 to be a leap year.
Step-by-step Solution:

1. **Initialize year:** Let $Y = 2024$.
2. **Implement nested logic:**

```
year = 2024
is_leap = False

if year % 4 == 0:
    if year % 100 == 0:
        if year % 400 == 0:
            is_leap = True
        else:
            is_leap = False
    else:
        is_leap = True
else:
    is_leap = False

print("Leap Year:" , is_leap)
```

3. **Execution trace:**
   - $2024 \pmod 4 = 0$ is `True`. Enter outer block.
   - $2024 \pmod{100} = 0$ is `False`. Execute the `else` block of the first nested `if`.
   - `is_leap` is set to `True`.

3.3 Repetition

3.3.1 For loop

The `for` loop is used to iterate over a sequence (such as a list, tuple, string, or range) and execute a block of code for each item in the sequence.

Methodology:

For Loop Traversal **Algorithm:**

1. Define an iterable sequence (e.g. using the `range()` function).

2. Assign the first element of the sequence to the iteration variable.
3. Execute the block of code within the loop.
4. Reassign the iteration variable to the next element in the sequence.
5. Repeat steps 3 and 4 until the sequence is exhausted.

Worked Example:

Multiplication Table **Problem Statement:** Generate the multiplication table for a given integer N from 1 to 5 using a `for` loop.
Step-by-step Solution:

1. **Initialize variable:** Let $N = 7$.
2. **Define loop sequence:** Use `range(1, 6)` to generate numbers 1 through 5.
3. **Implement the loop:**

```
N = 7
for i in range(1, 6):
    result = N * i
    print(N, "x", i, "=", result)
```

4. **Execution trace:**
   - Iteration 1: $i = 1$, result$=7 \times 1 = 7$.
   - Iteration 2: $i = 2$, result$=7 \times 2 = 14$.
   - The loop continues up to $i = 5$.

3.3.2 While loop

The `while` loop repeatedly executes a target statement or block of statements as long as a given boolean condition evaluates to `True`.

Methodology:

While Loop Execution **Algorithm:**

1. Initialize loop control variables before the loop begins.
2. Evaluate the loop condition.
3. If the condition is `False`, bypass the loop block and continue program execution.
4. If the condition is `True`, execute the loop body.
5. Update the loop control variables within the body to prevent an infinite loop.
6. Jump back to step 2.

Worked Example:

Reversing an Integer **Problem Statement:** Given a positive integer N, find its reverse using a `while` loop.
Step-by-step Solution:

1. **Initialize variables:** Let $N = 145$. Set `reversed_num = 0`.

2. **Define loop condition:** The loop should run as long as $N > 0$.
3. **Implement logic:** Extract the last digit, append it, and reduce N.

```
n = 145
reversed_num = 0

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

4. **Execution trace:**
   - Step 1: `digit` $= 5$, `reversed_num` $= 5$, $n = 14$.
   - Step 2: `digit` $= 4$, `reversed_num` $= 54$, $n = 1$.
   - Step 3: `digit` $= 1$, `reversed_num` $= 541$, $n = 0$.
   - Loop terminates since $n > 0$ is `False`.

3.3.3 Nested loop

A nested loop is a loop inside the body of another loop. The inner loop executes completely for each iteration of the outer loop.

Methodology:

Nested Loop Execution **Algorithm:**

1. Initialize outer loop.
2. For the current iteration of the outer loop, initialize the inner loop.
3. Execute the inner loop until its termination condition is met.
4. Complete the remainder of the outer loop body, update its variables, and advance to its next iteration.
5. Repeat until the outer loop termination condition is met.

Worked Example:

Right Angled Triangle Pattern **Problem Statement:** Print a right-angled triangle pattern of stars with a height of $R = 4$ rows.
Step-by-step Solution:

1. **Outer loop logic:** Controls the number of rows. Iterate i from 1 to R.
2. **Inner loop logic:** Controls the columns per row. Iterate j from 1 to i.
3. **Implementation:**

```
R = 4
for i in range(1, R + 1):
    # inner loop
    for j in range(1, i + 1):
        print("*", end=" ")
    # move to next line after each row
    print()
```

4. **Execution trace:**
    - $i = 1$: inner loop prints one * and ends.
    - $i = 2$: inner loop prints two * symbols.
    - $i = 3$: inner loop prints three * symbols.
    - $i = 4$: inner loop prints four * symbols.

3.4 Break and Continue Statements

3.4.1 Break statement

The `break` statement alters the normal flow of a loop by immediately terminating it entirely. Execution resumes at the first statement following the loop structure.

Methodology:

Break Statement Control Flow **Algorithm:**

1. Start normal loop execution.
2. Evaluate a conditional check for the `break` trigger.
3. If the condition is `True`, immediately stop the current iteration.
4. Exit the loop structure entirely and do not execute any further iterations.

Worked Example:

Finding the First Divisor **Problem Statement:** Find the first number in the range 20 to 50 that is perfectly divisible by both 3 and 7.
Step-by-step Solution:

1. **Setup sequence:** Iterate over numbers from 20 to 50 using a `for` loop.
2. **Check conditions:** Within the loop, check if the current number modulo 3 is 0 **and** modulo 7 is 0.
3. **Apply break:** Once a match is found, print the number and use `break` to avoid checking the rest.

```
for num in range(20, 51):
    if num % 3 == 0 and num % 7 == 0:
        print("First match:", num)
        break
```

4. **Execution trace:** The loop iterates until `num = 21`. The condition evaluates to `True`. It prints "First match: 21", executes `break`, and the loop is permanently terminated.

3.4.2 Continue statement

The `continue` statement forces the loop to skip the remainder of its current iteration and immediately re-evaluates the loop condition (or grabs the next item in a `for` loop) to proceed with the next iteration.

Methodology:

Continue Statement Control Flow **Algorithm:**

1. Start a loop iteration.

2. Check a conditional statement for the `continue` trigger.
3. If the condition is `True`, skip all remaining statements in the current iteration block.
4. Jump immediately to the loop update expression or the next sequence item and begin the next iteration.

Worked Example:

Sum of Positive Numbers **Problem Statement:** Given a list of integers `[12, -4, 5, -9, 8]`, compute the sum of strictly positive numbers by skipping negative values using a `continue` statement.
Step-by-step Solution:

1. **Initialize variables:** `total = 0`.
2. **Iterate through list:** Use a `for` loop.
3. **Apply continue:** If a number is less than 0, execute `continue`.

```
numbers = [12, -4, 5, -9, 8]
total = 0

for num in numbers:
    if num < 0:
        continue
    total = total + num

print("Sum of positive numbers:", total)
```

4. **Execution trace:**
   - `num = 12`: added to `total` (12).
   - `num = -4`: `num < 0` is `True`, executes `continue`, skips addition.
   - `num = 5`: added to `total` (17).
   - `num = -9`: executes `continue`.
   - `num = 8`: added to `total` (25).

CHAPTER 4
Functions

4.1 Introduction to Functions

4.1.1 User Defined Functions

Methodology:

Defining and Calling User Defined Functions A function is a reusable block of code that performs a specific computational task.

1. **Define the function** using the `def` keyword followed by the function name and parentheses `()`.
2. **Specify parameters** inside the parentheses if the function requires input values to perform calculations.
3. **Write the function body** with proper indentation. This block contains the core computational logic.
4. **Return a value** using the `return` statement to pass the computed result back to the caller.
5. **Call the function** by using its name followed by arguments in parentheses.

Syntax:

```
def function_name(parameters):
    # Computational logic
    return result
```

Worked Example:

Calculate Simple Interest using a Function Write a Python function to compute simple interest given a principal amount, rate of interest, and time period.
Step 1: Define the function and parameters We need three parameters to represent the variables in the simple interest formula.

```
def calculate_simple_interest(p, r, t):
```

Step 2: Implement the formula inside the function The mathematical formula for simple interest is $\frac{p \times r \times t}{100}$. We calculate this and return the result.

```
    si = (p * r * t) / 100
    return si
```

Step 3: Call the function and print the result Assign values to variables and pass them as arguments to the function.

```
principal = 1000
rate = 5
```

```
time = 2
interest = calculate_simple_interest(principal, rate, time)
print("Simple Interest is:", interest)
```

Output:

```
Simple Interest is: 100.0
```

4.1.2 Arguments and Parameters

Methodology:

Types of Function Arguments Python supports various computational ways to pass arguments to a function:

1. **Positional Arguments:** Passed in strict positional order. The number of arguments passed must exactly match the number of parameters.
2. **Keyword Arguments:** Passed as `parameter_name=value`. The order of arguments does not matter because they are explicitly linked to parameter names.
3. **Default Arguments:** Parameters defined with a default value, e.g., `def func(a, b=10)`. The default value is used if no argument is provided during the function call.
4. **Variable-Length Arguments:** Used when the exact number of inputs is unknown.
   - `*args`: Gathers a variable number of positional arguments into a tuple.
   - `**kwargs`: Gathers a variable number of keyword arguments into a dictionary.

Worked Example:

Using Different Argument Types Create a function that calculates the total cost of an item with varying argument types for the item price, tax rate, and discount amount.
Step 1: Define the function with default arguments Set default values for `tax_rate` and `discount`.

```
def calculate_total_cost(price, tax_rate=0.05, discount=0):
    total = price + (price * tax_rate) - discount
    return total
```

Step 2: Use Positional Arguments Provide only the required argument `price`. Default values are automatically applied for `tax_rate` and `discount`.

```
cost1 = calculate_total_cost(100)
print(cost1)  # Output: 105.0
```

Step 3: Use Keyword Arguments Specify parameter names explicitly to pass arguments out of order or bypass earlier default arguments.

```
cost2 = calculate_total_cost(discount=10, price=200)
print(cost2)  # Output: 200.0 (calculation: 200 + 10 - 10)
```

4.2 Scope of a Variable

Methodology:

Managing Local and Global Variable Scope The scope of a variable defines the region of the code where the variable is accessible.

1. **Local Variable:** A variable declared inside a function block. It exists only in local memory and is destroyed once the function completes execution.
2. **Global Variable:** A variable declared outside all function blocks. It is accessible globally throughout the entire script.
3. **Modifying Global Variables:** If a function needs to modify a global variable, the variable must be explicitly declared within the function using the `global` keyword before any assignment occurs.
4. **Name Resolution (LEGB Rule):** Python sequentially resolves variable names in the following order: Local (L), Enclosing (E), Global (G), Built-in (B).

Worked Example:

Tracking Counter with Global Variables Write a program using a function to increment a globally scoped counter variable that tracks the number of times an operation is performed.
Step 1: Define a global variable

```
operation_counter = 0
```

Step 2: Create a function to modify the global variable Attempting to modify `operation_counter` directly via `+=` without the `global` keyword will cause a local unbound error. The `global` keyword binds the local identifier to the global memory space.

```
def perform_operation():
    global operation_counter
    operation_counter += 1
    print("Operation performed. Count is now:", operation_counter)
```

Step 3: Call the function and observe changes to the global state

```
print("Initial count:", operation_counter)
perform_operation()
perform_operation()
print("Final count:", operation_counter)
```

Output:

```
Initial count: 0
Operation performed. Count is now: 1
Operation performed. Count is now: 2
Final count: 2
```

4.3 Python Standard Library

4.3.1 Built-in Functions

Methodology:

Input Output and Mathematical Built-in Functions Python provides foundational built-in functions for routine operations without requiring explicit module imports.

1. **Input and Output Functions:**
   - `input(prompt)`: Reads a line from standard input, converts it to a string, and returns it.
   - `print(*objects)`: Evaluates objects, converts them to strings, and prints them to standard output.
2. **Mathematical Functions:**
   - `abs(x)`: Returns the absolute (positive magnitude) value of a number.
   - `divmod(a, b)`: Performs floor division and modulo simultaneously. Returns a tuple `(a // b, a % b)`.
   - `max(*args)`: Evaluates an iterable or multiple arguments and returns the largest computational value.
   - `min(*args)`: Evaluates and returns the smallest computational value.
   - `pow(base, exp)`: Performs exponentiation returning *base* raised to the power of *exp* ($base^{exp}$).
   - `sum(iterable)`: Computes the numeric sum of items in an iterable object from left to right.

Worked Example:

Applying Built-in Mathematical Functions Solve a data processing problem where an array of numerical scores needs aggregation, absolute value transformation, and modular arithmetic calculations.
Step 1: Initialize the dataset Assume we have the following list of raw data:

```
data = [85, -92, 78, -65, 100]
```

Step 2: Find the Maximum Minimum and Sum Utilize built-in aggregation functions.

```
highest_val = max(data)
lowest_val = min(data)
total_sum = sum(data)
```

Step 3: Convert to absolute values Transform all negative magnitudes to positive magnitudes.

```
abs_data = [abs(val) for val in data]
```

Step 4: Apply divmod and pow Determine how many chunks of 30 fit into the maximum absolute value and extract the remainder. Then square the remainder.

```
chunks, remainder = divmod(max(abs_data), 30)
squared_remainder = pow(remainder, 2)
```

Step 5: Output the computational results

```
print("Max:", highest_val, "Min:", lowest_val, "Sum:", total_sum)
print("Absolute data:", abs_data)
print("Chunks of 30:", chunks, "Remainder:", remainder)
print("Squared remainder:", squared_remainder)
```

Output:

```
Max: 100 Min: -92 Sum: 106
Absolute data: [85, 92, 78, 65, 100]
Chunks of 30: 3 Remainder: 10
Squared remainder: 100
```

4.3.2 Modules

Methodology:

Importing and Using math random and statistics Modules A module is a Python file containing predefined computational functions, classes, and variables.

1. **Importing:** Load a module into the workspace using `import module_name` or import specific functions using `from module_name import function_name`.
2. **`math` Module:** Grants access to C-standard mathematical functions.
   - `math.sqrt(x)`: Computes the square root of x.
   - `math.factorial(x)`: Computes $x!$ (factorial).
   - `math.pi`: Returns the mathematical constant π.
3. **`random` Module:** Implements pseudo-random number generators for simulations.
   - `random.randint(a, b)`: Returns a random integer N such that $a \leq N \leq b$.
   - `random.choice(seq)`: Selects a random element from an iterable sequence.
   - `random.random()`: Generates a random floating-point number in the range [0.0, 1.0).
4. **`statistics` Module:** Provides algorithms for mathematical statistics.
   - `statistics.mean(data)`: Calculates the arithmetic mean (average).
   - `statistics.median(data)`: Calculates the median (middle value).
   - `statistics.mode(data)`: Calculates the mode (most common value).

Worked Example:

Statistical Analysis of Random Data using Modules Write a script to generate a synthetic dataset of random test scores, perform mathematical transformations, and extract statistical metrics.

Step 1: Import required library modules

```
import random
import math
import statistics
```

Step 2: Generate random data array Use a list comprehension to generate 5 random integer scores scaled between 50 and 100.

```
random.seed(42) # Seed for reproducible output
scores = [random.randint(50, 100) for _ in range(5)]
print("Generated scores:", scores)
```

Step 3: Perform advanced math operations Calculate the square root of each generated score utilizing the `math` module.

```
sqrt_scores = [math.sqrt(s) for s in scores]
```

Step 4: Calculate statistical parameters Find the statistical mean and median using the `statistics` module.

```
mean_score = statistics.mean(scores)
median_score = statistics.median(scores)

print("Mean of scores:", mean_score)
print("Median of scores:", median_score)
```

Output:

```
Generated scores: [90, 64, 53, 94, 63]
Mean of scores: 72.8
Median of scores: 64
```

CHAPTER 5
Strings and Lists

5.1 Introduction to Strings and Operations

A string in Python is an immutable sequence of characters.

Methodology:

Basic String Operations

1. **Concatenation**: Use the `+` operator to join two strings.
2. **Repetition**: Use the `*` operator to repeat a string multiple times.
3. **Indexing**: Access individual characters using zero-based indices (e.g., `s[0]`). Negative indices access from the end (e.g., `s[-1]`).
4. **Slicing**: Extract substrings using the syntax `s[start:stop:step]`.
5. **Traversing**: Iterate through characters using a `for` or `while` loop.

Worked Example:

Traversing and Reversing a String **Problem:** Write a Python program to traverse the string "PYTHON" and print each character along with its index. Then print the string in reverse.
Step-by-step Solution:

1. **Initialize the string**: Let `text = "PYTHON"`.
2. **Traverse using an index loop**:

```
for i in range(len(text)):
    print("Index", i, "->", text[i])
```

Output:

```
Index 0 -> P
Index 1 -> Y
Index 2 -> T
Index 3 -> H
Index 4 -> O
Index 5 -> N
```

3. **Reverse the string using slicing**: The syntax `text[::-1]` creates a reversed copy.

```
reversed_text = text[::-1]
print("Reversed:", reversed_text)
```

Output: `Reversed:  NOHTYP`

Worked Example:

Extracting Substrings using Slicing **Problem:** Given the string `S = "DiplomaEngineering"` extract the following: 1. The first 7 characters. 2. The word "Engineering". 3. Every second character of the string.
Step-by-step Solution:

1. **First 7 characters**: Slice from index 0 to 7 (exclusive).

```
part1 = S[0:7]
# part1 is "Diploma"
```

2. **Word "Engineering"**: Slice from index 7 to the end.

```
part2 = S[7:]
# part2 is "Engineering"
```

3. **Every second character**: Use step size 2.

```
part3 = S[::2]
# part3 is "DpoaEgnrin"
```

5.2 String Methods and Built-in Functions

Python provides built-in functions and string methods to process text data efficiently.

Methodology:

Common String Methods

1. `len(s)`: Returns the total number of characters in the string.
2. `s.lower()` / `s.upper()`: Converts string to lowercase or uppercase.
3. `s.find(sub)`: Returns the lowest index where substring `sub` is found. Returns -1 if not found.
4. `s.replace(old new)`: Replaces occurrences of substring `old` with `new`.
5. `s.split(sep)`: Splits the string into a list of words using `sep` as the delimiter.
6. `sep.join(list)`: Joins a list of strings into a single string separated by `sep`.

Worked Example:

Counting Vowels in a String **Problem:** Write a program to count the number of vowels in the string "Education".
Step-by-step Solution:

1. **Initialize string and counter**: `s = "Education"` and `count = 0`.
2. **Convert string to lowercase** to make matching easier:

```
s_lower = s.lower()  # "education"
```

3. **Iterate and check**: Loop through each character and check if it is in "aeiou".

```
for char in s_lower:
    if char in "aeiou":
        count += 1
```

4. **Final result**: The variable `count` holds the value 5.

Worked Example:

String Replacement and Splitting **Problem:** Given the string `data = "Python is hard"` replace the word "hard" with "easy" and split the sentence into a list of words.
Step-by-step Solution:

1. **Replace the word**:

```
new_data = data.replace("hard", "easy")
# new_data is "Python is easy"
```

2. **Split into words**:

```
word_list = new_data.split(" ")
# word_list is ['Python', 'is', 'easy']
```

5.3 Introduction to List and its Operations

A list is a mutable ordered collection of items enclosed in square brackets `[]`. Elements can be of different data types.

Methodology:

Basic List Operations

1. **Access and Modification**: Elements are accessed via index `L[i]`. Since lists are mutable you can update elements using `L[i] = new_value`.
2. **Concatenation**: Use the `+` operator to join two lists.
3. **Repetition**: Use the `*` operator to repeat list elements multiple times.
4. **Membership**: Use the `in` operator to check if an item exists in the list.
5. **Slicing**: Extract sublists using the syntax `L[start:stop:step]`.

Worked Example:

Updating and Slicing a List **Problem:** Given the list `L = [10, 20, 30, 40, 50]` replace the value 30 with 35 and extract the last three elements.
Step-by-step Solution:

1. **Update element**: The value 30 is located at index 2.

```
L[2] = 35
# L is now [10, 20, 35, 40, 50]
```

2. **Extract last three elements**: Slice from index -3 to the end.

```
sub_L = L[-3:]
# sub_L is [35, 40, 50]
```

5.4 List Methods and Built-in Functions

Lists come with numerous methods to dynamically add remove and arrange items.

Methodology:

Common List Methods

1. `L.append(x)`: Adds item `x` to the end of the list.
2. `L.insert(i x)`: Inserts item `x` at index `i`.
3. `L.remove(x)`: Removes the first occurrence of item `x`.
4. `L.pop(i)`: Removes and returns the element at index `i` (or the last item if `i` is omitted).
5. `L.sort()`: Sorts the list in ascending order in place.
6. `L.reverse()`: Reverses the list elements in place.
7. **Built-in functions**: `len(L)` `max(L)` `min(L)` `sum(L)`.

Worked Example:

Dynamic List Manipulation **Problem:** Create an empty list add the numbers 5 2 and 8 sort them in ascending order and remove the smallest number.
Step-by-step Solution:

1. **Create list and add items**:

```
nums = []
nums.append(5)
nums.append(2)
nums.append(8)
# nums is [5, 2, 8]
```

2. **Sort the list**:

```
nums.sort()
# nums is [2, 5, 8]
```

3. **Remove the smallest number**: Since the list is sorted the smallest number is at index 0.

```
nums.pop(0)
# nums is [5, 8]
```

5.5 Nested and Copying Lists

Lists can contain other lists as elements forming nested lists. This is particularly useful for representing matrices. When duplicating lists copying behaves differently depending on whether it is a shallow copy or deep copy.

Methodology:

List Copying and Nested Lists

1. **Aliasing**: Assignment `L2 = L1` creates a reference. Changing `L2` affects `L1`.
2. **Shallow Copying**: Use `L2 = L1[:]` or `L2 = L1.copy()` to create an independent copy of a flat list.
3. **Deep Copying**: For nested lists a shallow copy only copies outer references. Use `copy.deepcopy(L)` from the `copy` module to completely duplicate nested lists.
4. **Nested Access**: Access inner elements using chained brackets for example `matrix[row][col]`.

Worked Example:

Matrix Addition using Nested Lists **Problem:** Add two 2×2 matrices represented by nested lists: `A = [[1, 2], [3, 4]]` and `B = [[5, 6], [7, 8]]`.
Step-by-step Solution:

1. **Initialize result matrix**: Create an empty 2×2 list `C = [[0, 0], [0, 0]]`.
2. **Iterate through rows and columns**:

```
for i in range(2):          # Iterate rows
    for j in range(2):      # Iterate columns
        C[i][j] = A[i][j] + B[i][j]
```

3. **Calculate step-by-step**:
   - `C[0][0] = A[0][0] + B[0][0] = 1 + 5 = 6`
   - `C[0][1] = A[0][1] + B[0][1] = 2 + 6 = 8`
   - `C[1][0] = A[1][0] + B[1][0] = 3 + 7 = 10`
   - `C[1][1] = A[1][1] + B[1][1] = 4 + 8 = 12`
4. **Final result**: `C` is `[[6, 8], [10, 12]]`.

Worked Example:

Aliasing vs Slicing **Problem:** Demonstrate the effect of aliasing vs copying using the slicing operator.
Step-by-step Solution:

1. **Create list and alias**:

```
L1 = [1, 2, 3]
L2 = L1         # Aliasing (Reference assignment)
```

2. **Modify alias**:

```
L2[0] = 99
# Because of aliasing both L1 and L2 become [99, 2, 3]
```

3. **Create independent copy**:

```
L3 = L1[:]      # Copying using slicing
L3[1] = 88
# L1 remains [99, 2, 3] while L3 becomes [99, 88, 3]
```

5.6 Lists as Arguments to Functions

When a list is passed as an argument to a function the function receives a reference to the original list. Any modifications made to the list's elements inside the function will directly alter the original list.

Methodology:

Passing Lists to Functions

1. Define a function to accept a parameter representing the list.
2. Apply operations like `append()` `remove()` or direct index assignment (e.g. `L[i] = new_val`) to modify the list in place.

3. To prevent modifying the original list inadvertently pass a shallow copy of the list (e.g. `func(my_list[:])`).

Worked Example:

Modifying List Elements in Place **Problem:** Write a function `square_elements(lst)` that takes a list of numbers and squares each element in place.
Step-by-step Solution:

1. **Define the function**: Iterate through the list using indices to overwrite original elements.

```
def square_elements(lst):
    for i in range(len(lst)):
        lst[i] = lst[i] ** 2
```

2. **Call the function**: Create a list and pass it.

```
numbers = [2, 3, 4]
square_elements(numbers)
```

3. **Trace execution**:
   - `i = 0`: `lst[0] = 2 ** 2 = 4`
   - `i = 1`: `lst[1] = 3 ** 2 = 9`
   - `i = 2`: `lst[2] = 4 ** 2 = 16`
4. **Check original list**: Because lists are passed by reference the variable `numbers` is now `[4, 9, 16]`.

Formulae

Appendix: Key Formulae

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

Unit 1: Problem Solving using Flowchart and Algorithm

Summation The total sum of two variables, commonly used in basic algorithms:

$$\text{Sum} = A + B$$

where A and B represent the numeric inputs.