Python (DI02032011) - Problem Solver

Antigravity AI

June 4, 2026

Contents

CHAPTER 1

Problem Solving using Flowchart and Algorithm

This chapter focuses on the fundamental techniques of problem-solving in computer science: algorithms and flowcharts. We will explore how to formulate step-by-step solutions to computational problems and represent them visually.

1.1 Developing Algorithms

An algorithm is a finite sequence of precise, unambiguous steps to solve a specific problem.

Methodology:

Algorithm Construction Steps Follow these steps to construct an effective algorithm:

1. **Identify the Input:** Determine the data required from the user or system to solve the problem.
2. **Identify the Processing Variables:** Define the variables needed to store intermediate results or final outputs.
3. **Determine the Logic:** Formulate the sequence of mathematical operations or logical conditions required to transform the input into the desired output.
4. **Define the Output:** Specify the final result that the algorithm must produce.
5. **Write Step-by-Step Instructions:** Express the logic sequentially using simple English or pseudocode, starting with a "Start" step and ending with a "Stop" step.

Worked Example:

Algorithm to Calculate Simple Interest **Problem Statement:** Write an algorithm to compute the simple interest for a given principal amount, rate of interest and time period.
Solution:

1. **Step 1:** Start
2. **Step 2:** Read the values of Principal (P), Rate of Interest (R) and Time (T).
3. **Step 3:** Calculate the simple interest using the formula:
   $SI = (P \times R \times T)/100$
4. **Step 4:** Print the calculated simple interest (SI).
5. **Step 5:** Stop

Worked Example:

Algorithm to Check Even or Odd Number **Problem Statement:** Write an algorithm to check whether a given integer is even or odd.
Solution:

1. **Step 1:** Start
2. **Step 2:** Read an integer number as N.
3. **Step 3:** Check if the remainder of N divided by 2 is 0 ($N \pmod 2 == 0$).
4. **Step 4:** If the condition in Step 3 is true, then print "Number is Even".
5. **Step 5:** If the condition in Step 3 is false, then print "Number is Odd".
6. **Step 6:** Stop

1.2 Designing Flowcharts

A flowchart is a graphical representation of an algorithm using standardized geometric shapes connected by arrows to show the flow of control.

Methodology:

Standard Flowchart Symbols When drawing a flowchart, use the following standard geometric shapes:

Purpose	Shape Name	Usage in Flowchart
Terminal	Oval / Ellipse	Marks the Start or Stop point of the program.
Input or Output	Parallelogram	Represents reading input data or printing output results.
Process	Rectangle	Represents mathematical calculations or data assignments.
Decision	Diamond	Evaluates a condition and branches the flow (Yes or No).
Flowlines	Arrows	Indicates the direction of logic flow between symbols.

Worked Example:

Flowchart Logic for Finding Largest of Three Numbers **Problem Statement:** Formulate the step-by-step logic and describe the flowchart components required to find the largest among three distinct numbers A, B and C.
Solution:

1. **Terminal:** Use an oval shape with the text "Start".
2. **Input:** Use a parallelogram to "Read A, B, C".
3. **Decision 1:** Use a diamond shape with the condition "Is $A > B$?".
   - **If Yes:** Flow goes to Decision 2.
   - **If No:** Flow goes to Decision 3.
4. **Decision 2 (From Yes branch of Decision 1):** Use a diamond shape with the condition "Is $A > C$?".
   - **If Yes:** Use a parallelogram (Output) to "Print A is largest".
   - **If No:** Use a parallelogram (Output) to "Print C is largest".
5. **Decision 3 (From No branch of Decision 1):** Use a diamond shape with the condition "Is $B > C$?".
   - **If Yes:** Use a parallelogram (Output) to "Print B is largest".

- **If No:** Use a parallelogram (Output) to "Print C is largest".

6. **Terminal:** All output paths converge to an oval shape with the text "Stop".

1.3 Iterative Problem Solving (Loops)

Iteration involves executing a set of statements repeatedly until a specific condition is met.

Methodology:

Algorithm for Loop Execution To construct an algorithm involving loops:

1. **Initialization:** Set an initial value for a loop counter variable.
2. **Condition Evaluation:** Define the stopping condition for the loop.
3. **Loop Body:** Write the steps that need to be repeated.
4. **Update Counter:** Increment or decrement the loop counter.
5. **Branching:** Direct the flow back to the condition evaluation until the condition fails.

Worked Example:

Algorithm to Calculate Factorial of a Number **Problem Statement:** Write an algorithm to find the factorial of a positive integer N (where $N! = 1 \times 2 \times \cdots \times N$).
Solution:

1. **Step 1:** Start
2. **Step 2:** Read the number N.
3. **Step 3:** Initialize two variables: $Fact = 1$ and $i = 1$.
4. **Step 4:** Check condition: If $i > N$, go to Step 8.
5. **Step 5:** Calculate $Fact = Fact \times i$.
6. **Step 6:** Increment i by 1 ($i = i + 1$).
7. **Step 7:** Go back to Step 4.
8. **Step 8:** Print the factorial value $Fact$.
9. **Step 9:** Stop

Worked Example:

Algorithm to Generate Fibonacci Series **Problem Statement:** Write an algorithm to generate the Fibonacci series up to N terms. The series starts with 0 and 1, and each subsequent term is the sum of the previous two (0, 1, 1, 2, 3, 5...).
Solution:

1. **Step 1:** Start
2. **Step 2:** Read the number of terms N.
3. **Step 3:** Initialize variables: $A = 0$, $B = 1$, and $Count = 2$.
4. **Step 4:** Print A and B.

5. **Step 5:** Check condition: If $Count \geq N$, go to Step 10.
6. **Step 6:** Calculate the next term: $NextTerm = A + B$.
7. **Step 7:** Print $NextTerm$.
8. **Step 8:** Update variables for the next iteration: $A = B$ and $B = NextTerm$.
9. **Step 9:** Increment counter: $Count = Count + 1$ and go to Step 5.
10. **Step 10:** Stop

CHAPTER 2

Introduction to Python

2.1 Python Programming Basics

Python is a high-level, interpreted programming language known for its simplicity and readability. It is dynamically typed and supports multiple programming paradigms.

Methodology:

Python Installation and Setup 1. Visit the official Python website at `python.org`. 2. Download the installer matching your operating system (Windows, macOS, or Linux). 3. Run the downloaded installer. 4. **Crucial Step:** Ensure you check the box that says "Add Python to PATH" before proceeding. 5. Click "Install Now" and follow the wizard. 6. Verify the installation by opening a command prompt or terminal and executing `python -version`.

Methodology:

Writing a Basic Python Program 1. Use comments starting with `#` to document the code. 2. Define variables and assign them values using the `=` operator. There is no need to declare types explicitly. 3. Use appropriate operators (arithmetic, relational, etc.) to perform computations. 4. Output the final results to the console using the `print()` function.

Worked Example:

Simple Arithmetic Computation Write a Python program to calculate the area of a rectangle given length $L = 8$ and width $W = 12$.
Solution:

1. Assign the given values to variables `L` and `W`.
2. Compute the area using the formula Area $= L \times W$.
3. Display the computed area.

```
# Calculate Area of a Rectangle
L = 8
W = 12
Area = L * W
print("The area of the rectangle is:", Area)
```

Output:

```
The area of the rectangle is: 96
```

2.2 Identifiers, Variables and Data Types

Methodology:

Rules for Identifiers and Determining Data Types 1. **Identifiers** must begin with a letter (a-z, A-Z) or an underscore (_), followed by letters, digits, or underscores. 2. Identifiers are case-sensitive (`Var` and `var` are different). 3. Reserved keywords (e.g., `if`, `True`, `def`) cannot be used as identifiers. 4. **Variables** are created the moment a value is assigned to them. 5. **Data Types** (such as `int`, `float`, `str`, `bool`) are inferred automatically. 6. Use the built-in `type()` function to check a variable's data type.

Worked Example:

Declaring Variables and Determining Types Assign the values `-45`, `9.81`, `False`, and `"Hello World"` to valid variables and output their data types.
Solution:

1. Create valid identifiers for each value.
2. Assign the values.
3. Use `type()` inside `print()` to display the data type.

```
# Variable assignments
temperature_val = -45
gravity = 9.81
is_active = False
greeting_message = "Hello World"

# Output types
print(type(temperature_val))
print(type(gravity))
print(type(is_active))
print(type(greeting_message))
```

Output:

```
<class 'int'>
<class 'float'>
<class 'bool'>
<class 'str'>
```

2.3 Operators in Python

Python provides a rich set of operators to manipulate variables and values computationally.

2.3.1 Arithmetic Operators

Methodology:

Evaluating Arithmetic Expressions 1. **Addition (+)**: Adds operands. 2. **Subtraction (-)**: Subtracts the right operand from the left. 3. **Multiplication (*)**: Multiplies operands. 4. **Division (/)**: Divides left by right (always results in a float). 5. **Floor Division (//)**: Divides and truncates the fractional part to return an integer. 6. **Modulus (%)**: Returns the remainder of the division. 7. **Exponentiation (**)**: Raises the left operand to the power of the right operand. 8. Expressions are evaluated following operator precedence (Parentheses, Exponentiation, Multiplication/Division/Floor/Mod, Addition/Subtraction).

Worked Example:

Evaluating a Mathematical Expression Given $x = 14$ and $y = 4$, evaluate the expression $z = (x + y) \times (x \,//\, y) - (x \,\%\, y)^3$.

Solution:

1. Substitute the values: $z = (14 + 4) \times (14 \,//\, 4) - (14 \,\%\, 4)^3$
2. Evaluate parentheses: $14 + 4 = 18$
3. Evaluate floor division: $14 \,//\, 4 = 3$
4. Evaluate modulus: $14 \,\%\, 4 = 2$ (since $14 = 4 \times 3 + 2$)
5. Substitute back: $z = 18 \times 3 - 2^3$
6. Evaluate exponentiation: $2^3 = 8$
7. Evaluate multiplication: $18 \times 3 = 54$
8. Evaluate subtraction: $54 - 8 = 46$

```
x = 14
y = 4
z = (x + y) * (x // y) - (x % y)**3
print("The result is:", z)
```

Output:

```
The result is: 46
```

2.3.2 Relational and Logical Operators

Methodology:

Evaluating Relational and Logical Expressions 1. **Relational Operators** (`==`, `!=`, `>`, `<`, `>=`, `<=`) compare values and evaluate to a boolean (`True` or `False`). 2. **Logical AND (`and`)**: Returns `True` only if both operands are `True`. 3. **Logical OR (`or`)**: Returns `True` if at least one operand is `True`. 4. **Logical NOT (`not`)**: Reverses the boolean state of its operand.

Worked Example:

Checking Multiple Conditions Given $a = 25$, $b = 30$, and $c = 25$, mathematically evaluate the following Python expressions: 1. `(a == c) and (b > a)` 2. `(a != b) or (c > b)` 3. `not (a <= c)`

Solution:

1. `(25 == 25) and (30 > 25)` → `True and True` → `True`.
2. `(25 != 30) or (25 > 30)` → `True or False` → `True`.
3. `not (25 <= 25)` → `not (True)` → `False`.

```
a = 25
b = 30
c = 25

print((a == c) and (b > a))
print((a != b) or (c > b))
print(not (a <= c))
```

Output:

```
True
True
False
```

2.3.3 Bitwise Operators

Methodology:

Applying Bitwise Operations 1. Convert the integer operands into binary representation. 2. **Bitwise AND (`&`)**: Sets each bit to 1 if both corresponding bits are 1. 3. **Bitwise OR (`|`)**: Sets each bit to 1 if at least one corresponding bit is 1. 4. **Bitwise XOR (`^`)**: Sets each bit to 1 if exactly one corresponding bit is 1. 5. **Left Shift (`« n`)**: Shifts bits to the left by n positions, filling with zeros on the right (equivalent to multiplying by 2^n). 6. **Right Shift (`» n`)**: Shifts bits to the right by n positions (equivalent to integer division by 2^n). 7. Convert the resulting binary sequence back to an integer.

Worked Example:

Computing Bitwise Operations Let $A = 12$ and $B = 7$. Compute $A \,\&\, B$, $A \mid B$, and $A \ll 2$.
Solution:

1. Convert to binary: $A = 12_{10} = 1100_2$, and $B = 7_{10} = 0111_2$.
2. **Bitwise AND ($A \,\&\, B$):**

$$\begin{array}{cccc} & 1 & 1 & 0 & 0 \\ \& & 0 & 1 & 1 & 1 \\ \hline & 0 & 1 & 0 & 0 \end{array}$$

   $0100_2 = 4_{10}$.
3. **Bitwise OR ($A \mid B$):**

$$\begin{array}{cccc} & 1 & 1 & 0 & 0 \\ \mid & 0 & 1 & 1 & 1 \\ \hline & 1 & 1 & 1 & 1 \end{array}$$

   $1111_2 = 15_{10}$.
4. **Left Shift ($A \ll 2$):** 1100_2 shifted left by 2 becomes 110000_2.
   $110000_2 = 32 + 16 = 48_{10}$. (Also $12 \times 2^2 = 48$).

```
A = 12
B = 7
print("A & B =", A & B)
print("A | B =", A | B)
print("A << 2 =", A << 2)
```

Output:

```
A & B = 4
A | B = 15
A << 2 = 48
```

2.3.4 Identity and Membership Operators

Methodology:

Using Identity and Membership Operators 1. **Identity Operators** (`is`, `is not`): Used to determine if two variables reference the exact same object in memory, not just if their values are equal. 2. **Membership Operators** (`in`, `not in`): Used to test whether a value or variable is found within a sequence (such as a string, list, or tuple).

Worked Example:

Identity and Membership Checks Test whether the substring `"Gram"` is present in the string `"Programming"`. Then check if two distinct lists with the same values are identical objects.
Solution:

1. Use the `in` operator on the string.
2. Create two lists with the same elements and use the `is` operator to check identity, and `==` to check value equality.

```
text = "Programming"
# Membership check
print("Gram" in text)
print("gram" in text)

list1 = [10, 20, 30]
list2 = [10, 20, 30]
# Identity check vs Value equality check
print(list1 is list2)
print(list1 == list2)
```

Output:

```
False
True
False
True
```

Note: "Gram" is not in "Programming" because string matching is case-sensitive.

2.4 Type Conversion

Type conversion (or type casting) is the process of converting data of one type into another.

Methodology:

Performing Type Conversion 1. **Implicit Type Conversion**: The Python interpreter automatically promotes a lower data type to a higher data type to prevent data loss (e.g., `int` + `float` → `float`). 2. **Explicit Type Conversion**: The programmer intentionally forces the type conversion using built-in functions:

- `int(val)`: Converts `val` to an integer (truncates decimals).
- `float(val)`: Converts `val` to a floating-point number.
- `str(val)`: Converts `val` to a string representation.

Worked Example:

Implicit and Explicit Type Conversion Demonstrate implicit conversion by adding an integer to a float. Then, demonstrate explicit conversion by concatenating an integer to a string, and parsing a numeric string to perform addition.

Solution:

1. For implicit conversion, directly add an `int` and a `float`.
2. For explicit conversion to string, use `str()` on the integer before applying the `+` operator for concatenation.
3. For explicit conversion to integer, use `int()` on the numeric string before applying the `+` operator for addition.

```
# Implicit Conversion
num_int = 15
num_float = 3.5
result = num_int + num_float
print("Implicit Result:", result)
print("Type of Result:", type(result))

# Explicit Conversion - Integer to String
rank = 1
message = "Your rank is " + str(rank)
print("String Concatenation:", message)

# Explicit Conversion - String to Integer
str_val = "150"
total = int(str_val) + 50
print("Integer Addition:", total)
```

Output:

```
Implicit Result: 18.5
Type of Result: <class 'float'>
String Concatenation: Your rank is 1
Integer Addition: 200
```

CHAPTER 3
Flow of Control

Flow of control refers to the order in which individual statements and instructions are executed or evaluated in a program. Python provides several control structures to direct the flow of execution, such as conditional statements and loops.

3.1 Conditional Statements

Conditional statements allow a program to make decisions based on certain conditions.

Methodology:

if Statement The `if` statement executes a block of code only if a specified condition is true.

1. Evaluate the condition expression.
2. If the condition evaluates to `True`, execute the indented block of code under the `if` statement.
3. If the condition evaluates to `False`, skip the block and proceed with the rest of the program.

Syntax:

```
if condition:
    # block of code
```

Worked Example:

Checking for Positive Number Write a program to check if a given number is positive.
Solution:

```
num = 10
if num > 0:
    print("The number is positive.")
```

Output:

```
The number is positive.
```

Methodology:

if-else Statement The `if-else` statement provides an alternative block of code to execute if the condition is false.

1. Evaluate the condition expression.
2. If `True`, execute the `if` block.
3. If `False`, execute the `else` block.

Syntax:

```
if condition:
    # block executed if true
else:
    # block executed if false
```

Worked Example:

Even or Odd Number Write a program to determine if a number is even or odd.
Solution:

```
num = 15
if num % 2 == 0:
    print(f"{num} is even.")
else:
    print(f"{num} is odd.")
```

Output:

```
15 is odd.
```

Methodology:

if-elif-else Statement The `if-elif-else` chain is used when there are multiple conditions to be checked sequentially.

1. Evaluate the first `if` condition. If `True`, execute its block and exit the chain.
2. If `False`, evaluate the next `elif` condition.
3. Repeat step 2 for all `elif` blocks.
4. If all conditions are `False`, execute the `else` block.

Syntax:

```
if condition1:
    # block 1
elif condition2:
    # block 2
else:
    # default block
```

Worked Example:

Largest of Three Numbers Write a program to find the largest of three given numbers.
Solution:

```
a = 12
b = 25
c = 9

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

```
print(f"The largest number is {largest}.")
```

Output:

```
The largest number is 25.
```

3.2 Looping Statements

Looping statements are used to execute a block of code repeatedly as long as a condition is met or for a specific sequence of items.

Methodology:

while Loop The `while` loop repeatedly executes a block of statements as long as a given condition is true.

1. Initialize loop control variables before the loop.
2. Evaluate the test condition.
3. If the condition is `True`, execute the loop body.
4. Update the loop control variables inside the loop.
5. Repeat steps 2-4 until the condition becomes `False`.

Syntax:

```
while condition:
    # loop body
```

Worked Example:

Sum of First N Natural Numbers Write a program to calculate the sum of the first N natural numbers using a while loop.
Solution:

```
n = 5
sum_n = 0
i = 1

while i <= n:
    sum_n += i
    i += 1

print(f"Sum of first {n} numbers is {sum_n}.")
```

Output:

```
Sum of first 5 numbers is 15.
```

Methodology:

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

1. Define the sequence or range to iterate over.
2. The loop variable takes the value of the first item in the sequence.

3. Execute the loop body.
4. The loop variable takes the value of the next item.
5. Repeat steps 3-4 until the sequence is exhausted.

Syntax:

```
for item in sequence:
    # loop body
```

Worked Example:

Factorial of a Number Write a program to compute the factorial of a given number using a for loop.
Solution:

```
num = 6
factorial = 1

for i in range(1, num + 1):
    factorial *= i

print(f"The factorial of {num} is {factorial}.")
```

Output:

```
The factorial of 6 is 720.
```

Worked Example:

Multiplication Table Write a program to print the multiplication table of a given number using a for loop.
Solution:

```
num = 4

for i in range(1, 11):
    print(f"{num} x {i} = {num * i}")
```

Output:

```
4 x 1 = 4
4 x 2 = 8
4 x 3 = 12
4 x 4 = 16
4 x 5 = 20
4 x 6 = 24
4 x 7 = 28
4 x 8 = 32
4 x 9 = 36
4 x 10 = 40
```

3.3 Loop Control Statements

Loop control statements change execution from its normal sequence. They allow you to prematurely terminate a loop or skip an iteration.

Methodology:

break Statement The `break` statement is used to terminate the loop entirely.

1. When a `break` statement is encountered inside a loop, the loop stops immediately.
2. The program control resumes at the next statement immediately following the loop.
3. If the loop is nested, `break` only terminates the innermost loop.

Worked Example:

Finding a Specific Number Iterate through numbers 1 to 10 and stop the loop when the number 7 is found.
Solution:

```
for i in range(1, 11):
    if i == 7:
        print("Number 7 found! Exiting loop.")
        break
    print(i)
```

Output:

```
1
2
3
4
5
6
Number 7 found! Exiting loop.
```

Methodology:

continue Statement The `continue` statement skips the rest of the code inside the loop for the current iteration only.

1. When a `continue` statement is encountered, the remaining statements in the current iteration are skipped.
2. The loop proceeds directly to the next iteration.

Worked Example:

Printing Only Odd Numbers Write a program to print only odd numbers from 1 to 10 using a loop and the continue statement.
Solution:

```
for i in range(1, 11):
    if i % 2 == 0:
        continue
    print(i)
```

Output:

```
1
3
5
7
9
```

Methodology:

pass Statement The `pass` statement is a null operation. It is used as a placeholder when a statement is required syntactically but no code needs to be executed.

1. Encountering `pass` results in no operation.
2. It is commonly used in empty functions loops or conditional blocks that are yet to be implemented.

Worked Example:

Using pass as a Placeholder Write a loop that iterates over a string and does nothing when it encounters a vowel, but prints consonants.

Solution:

```
word = "python"
vowels = "aeiou"

for char in word:
    if char in vowels:
        pass  # Placeholder for future code
    else:
        print(char)
```

Output:

```
p
y
t
h
n
```

CHAPTER 4
Functions

Functions are reusable blocks of code designed to perform a specific task. They help in breaking down complex computational problems into manageable and modular components.

4.1 Defining and Calling Functions

Methodology:

Defining a Custom Function 1. Use the `def` keyword followed by the function name and parentheses. 2. Specify parameters inside the parentheses if the function requires input variables. 3. End the function header line with a colon (`:`). 4. Write the function body with an indentation (usually 4 spaces). 5. Use the `return` statement at the end to output a value back to the caller. If omitted, the function returns `None`.

Worked Example:

Simple Interest Calculator Function **Problem:** Write a Python function to compute the simple interest given principal, rate, and time. Call the function for a principal of 5000, rate of 4.5%, and time of 3 years.
Step 1: Define the function and parameters.

```
def calculate_simple_interest(principal, rate, time):
```

Step 2: Write the computation logic inside the function body.

```
    interest = (principal * rate * time) / 100
    return interest
```

Step 3: Call the function with the given arguments.

```
result = calculate_simple_interest(5000, 4.5, 3)
print("Simple Interest:", result)
```

Output:

```
Simple Interest: 675.0
```

4.2 Function Arguments

Python supports various ways to pass arguments to a function, allowing flexibility in how functions are invoked.

Methodology:

Handling Different Types of Arguments 1. **Positional Arguments:** Pass values in the exact order as the parameters are defined. 2. **Keyword Arguments:** Pass values by specifying the parameter name explicitly (e.g. `rate=4.5`). Order does not matter. 3. **Default Arguments:** Assign a default value to a parameter in the function definition (e.g. `def func(a, b=10):`). This value is used if the caller skips providing it. 4.

Variable-Length Arguments: - Use `*args` to accept an arbitrary number of positional arguments as a tuple. - Use `**kwargs` to accept an arbitrary number of keyword arguments as a dictionary.

Worked Example:

Calculating Sum with Variable Arguments **Problem:** Write a function that accepts an arbitrary number of numeric arguments and returns their product.

Step 1: Define the function using *args.

```
def multiply_numbers(*args):
```

Step 2: Initialize a result variable and loop through args.

```
    result = 1
    for num in args:
        result *= num
    return result
```

Step 3: Call the function with multiple values.

```
product1 = multiply_numbers(2, 3, 4)
product2 = multiply_numbers(5, 10)

print(product1)  # Output: 24
print(product2)  # Output: 50
```

4.3 Variable Scope

The scope of a variable determines the portion of the program where you can access a particular identifier.

Methodology:

Variable Scope Rules 1. **Local Scope:** Variables created inside a function belong to the local scope of that function and can only be used inside it. 2. **Global Scope:** Variables created in the main body of the Python code are global and can be accessed anywhere. 3. **Global Keyword:** To modify a global variable inside a function, declare it within the function using the `global` keyword.

Worked Example:

Updating a Global Counter **Problem:** Create a global variable `counter` initialized to 0. Write a function `increment_counter` that increases its value by 1 each time it is called.

Step 1: Define the global variable.

```
counter = 0
```

Step 2: Define the function and use the global keyword.

```
def increment_counter():
    global counter
    counter += 1
```

Step 3: Call the function and print the counter.

```
increment_counter()
increment_counter()
print("Counter value:", counter)
```

Output:

```
Counter value: 2
```

4.4 Lambda Functions

Lambda functions are small anonymous functions defined using the `lambda` keyword instead of `def`. They are frequently used as arguments to higher-order functions like `map()`, `filter()`, and `sorted()`.

Methodology:

Creating Lambda Functions 1. Start with the `lambda` keyword. 2. Specify a space-separated list of arguments. 3. Write a colon (`:`). 4. Write a single expression. The result of this expression is automatically returned. Syntax: `lambda arguments: expression`

Worked Example:

Sorting a List of Tuples using Lambda **Problem:** Given a list of tuples representing students and their scores, sort the list in descending order based on the scores using a lambda function.
Step 1: Define the list of tuples.

```
students = [("Alice", 88), ("Bob", 95), ("Charlie", 82)]
```

Step 2: Use the built-in sorted() function with a lambda key. The lambda function takes one tuple `x` and returns the second element `x[1]` (the score).

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

Step 3: Print the result.

```
print(sorted_students)
```

Output:

```
[('Bob', 95), ('Alice', 88), ('Charlie', 82)]
```

4.5 Recursion

Recursion occurs when a function calls itself to solve a smaller instance of the same problem.

Methodology:

Designing a Recursive Function 1. **Identify the Base Case:** Determine the simplest condition where the function can return a result without making a recursive call. This prevents infinite recursion. 2. **Formulate the Recursive Step:** Define how the function calls itself with modified arguments, moving progressively closer to the base case. 3. Combine the base case and recursive step using `if-else` statements.

Worked Example:

Computing the Factorial of a Number **Problem:** Write a recursive function to compute the factorial of a non-negative integer n (denoted as $n!$).
Step 1: Identify the Base Case. The factorial of 0 or 1 is 1.

```
if n == 0 or n == 1:
    return 1
```

Step 2: Formulate the Recursive Step. For $n > 1$, $n! = n \times (n-1)!$.

```
else:
    return n * factorial(n - 1)
```

Step 3: Write the complete function and call it.

```
def factorial(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n - 1)

print("Factorial of 5 is:", factorial(5))
```

Output:

```
Factorial of 5 is: 120
```

Worked Example:

Generating Fibonacci Sequence recursively **Problem:** Write a recursive function to find the n-th term of the Fibonacci sequence, where $F_0 = 0$, $F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$.
Step 1: Define the base cases.

```
if n == 0:
    return 0
elif n == 1:
    return 1
```

Step 2: Define the recursive step.

```
else:
    return fibonacci(n - 1) + fibonacci(n - 2)
```

Step 3: Assemble the function.

```
def fibonacci(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)

print("The 6th Fibonacci number is:", fibonacci(6))
```

Output:

```
The 6th Fibonacci number is: 8
```

4.6 Built-in Functions

Python provides several built-in functions that accept other functions as arguments. These are particularly powerful when combined with lambda functions for data processing.

Methodology:

Using Map and Filter 1. **Map Method (`map(function, iterable)`):** Applies the given function to every item of the iterable (like a list) and returns a map object (an iterator) containing the results. 2. **Filter Method (`filter(function, iterable)`):** Constructs an iterator from elements of the iterable for which the function returns true.

Worked Example:

Squaring Numbers using Map **Problem:** Given a list of integers, create a new list containing the squares of each integer using the `map()` function.

Step 1: Define the list and the lambda function.

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

Step 2: Apply the map function.

```
squared_numbers = map(lambda x: x**2, numbers)
```

Step 3: Convert the map object to a list and print.

```
print(list(squared_numbers))
```

Output:

```
[1, 4, 9, 16, 25]
```

Worked Example:

Extracting Even Numbers using Filter **Problem:** Given a list of integers, extract only the even numbers using the `filter()` function.

Step 1: Define the list.

```
numbers = [10, 15, 20, 25, 30]
```

Step 2: Apply the filter function with a condition.

```
even_numbers = filter(lambda x: x % 2 == 0, numbers)
```

Step 3: Convert the filter object to a list and print.

```
print(list(even_numbers))
```

Output:

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

CHAPTER 5

Dictionary List Set String and Tuple

5.1 Strings

Methodology:

String Slicing and Basic Operations Strings in Python are ordered sequences of characters.

1. **Indexing**: Access individual characters using `s[i]`. Indexing starts at 0 for positive indices and -1 for negative indices (from the end).
2. **Slicing**: Extract substrings using `s[start:stop:step]`.
   - `start`: Starting index (inclusive).
   - `stop`: Ending index (exclusive).
   - `step`: Step size (default is 1).
3. **Concatenation and Repetition**: Use `+` to join strings and `*` to repeat a string.

Worked Example:

Extracting Substrings and Reversing a String Given the string `text = "PythonProgramming"`, perform the following operations:

1. Extract the word "Python".
2. Extract the word "Programming".
3. Reverse the entire string.
4. Extract every second character from the entire string.

Solution:

1. **Extract "Python"**: The characters are from index 0 to 5.

   `result = text[0:6]`
   Output: `"Python"`

2. **Extract "Programming"**: The characters start from index 6 to the end.

   `result = text[6:]`
   Output: `"Programming"`

3. **Reverse the string**: Use a step of -1.

   `result = text[::-1]`
   Output: `"gnimmargorPnohtyP"`

4. **Extract every second character**: Use a step of 2.

`result = text[::2]`
Output: `"PtoPormig"`

Methodology:

Common String Methods Strings are immutable but offer numerous built-in methods that return new strings:

1. **Case Modification**: `lower()`, `upper()`, `title()`, `capitalize()`.
2. **Search and Replace**:
   - `find(sub)`: Returns the lowest index of the substring or -1 if not found.
   - `replace(old new)`: Replaces all occurrences of `old` with `new`.
3. **Splitting and Joining**:
   - `split(sep)`: Splits the string into a list based on the separator.
   - `sep.join(iterable)`: Joins elements of an iterable into a single string using `sep`.
4. **Stripping**: `strip()`, `lstrip()`, `rstrip()` to remove leading and trailing whitespace or characters.

Worked Example:

String Manipulation and Formatting Write code to convert the sentence `sentence = " data science with python "` into title case, remove leading and trailing whitespace, and replace spaces with hyphens. Finally split the result into a list of words.
Solution:

1. **Strip Whitespace**: Remove extra spaces at the ends.

`s1 = sentence.strip()` → `"data science with python"`

2. **Convert to Title Case**: Capitalize the first letter of each word.

`s2 = s1.title()` → `"Data Science With Python"`

3. **Replace Spaces with Hyphens**: Replace `" "` with `"-"`.

`s3 = s2.replace(" ", "-")` → `"Data-Science-With-Python"`

4. **Split into List**: Split based on the hyphen separator.

`result = s3.split("-")` → `["Data", "Science", "With", "Python"]`

5.2 Lists

Methodology:

List Operations and Comprehension Lists are mutable ordered sequences.

1. **Addition**:
   - `append(elem)`: Adds an element to the end.

- `insert(index, elem)`: Inserts an element at a specific position.
- `extend(iterable)`: Appends all elements from an iterable.

2. **Removal**:
   - `remove(elem)`: Removes the first occurrence of the element.
   - `pop(index)`: Removes and returns the element at the index (default is last element).
   - `clear()`: Removes all elements.
3. **List Comprehension**: A concise way to create lists.

```
[expression for item in iterable if condition]
```

Worked Example:

Applying List Methods and Comprehension Given an initial list `numbers = [10, 20, 30, 40, 50]`:

1. Add 60 to the end of the list.
2. Insert 15 at index 1.
3. Remove the element 30.
4. Use list comprehension to create a new list containing the squares of all remaining numbers that are greater than 20.

Solution:

1. **Append 60**:

```
numbers.append(60)  →  [10, 20, 30, 40, 50, 60]
```

2. **Insert 15 at index 1**:

```
numbers.insert(1, 15)  →  [10, 15, 20, 30, 40, 50, 60]
```

3. **Remove 30**:

```
numbers.remove(30)  →  [10, 15, 20, 40, 50, 60]
```

4. **List Comprehension**: Iterate through the current `numbers`, check if greater than 20, and square them.

```
squares = [x**2 for x in numbers if x > 20]
```

Elements > 20 are 40, 50, and 60.

```
squares  →  [1600, 2500, 3600]
```

5.3 Tuples

Methodology:

Tuple Creation and Operations Tuples are immutable ordered sequences, meaning their elements cannot be changed after creation.

1. **Creation**: Defined using parentheses `()` or simply comma-separated values. A single-element tuple must include a trailing comma (e.g. `(5,)`).

2. **Tuple Packing and Unpacking**:
    - *Packing*: Assigning multiple values to a single tuple variable.
    - *Unpacking*: Assigning tuple elements to multiple distinct variables.
3. **Built-in Methods**: Due to immutability only two methods are available:
    - `count(value)`: Returns the number of times a value appears.
    - `index(value)`: Returns the first index of a value.

Worked Example:

Tuple Unpacking and Value Swapping Given a tuple containing a student's data: `student = ("Alice", 21, "Computer Engineering")`.

1. Unpack the tuple into three separate variables.
2. Use tuple unpacking to swap two variables `a = 10` and `b = 20` without using a temporary variable.

Solution:

1. **Unpack the Tuple**:

```
name, age, branch = student
```

   This assigns `"Alice"` to `name`, `21` to `age`, and `"Computer Engineering"` to `branch`.

2. **Swap Variables**: Python allows assigning multiple values simultaneously through tuple packing and unpacking.

```
a, b = b, a
```

   The right side `b, a` creates a tuple `(20, 10)`. Then unpacking assigns 20 to `a` and 10 to `b`.

5.4 Sets

Methodology:

Set Mathematical Operations Sets are mutable unordered collections of unique elements. They are optimized for checking membership and performing mathematical set operations.

1. **Union (`|` or `union()`)**: Elements in either set.
2. **Intersection (`&` or `intersection()`)**: Elements common to both sets.
3. **Difference (`-` or `difference()`)**: Elements in the first set but not in the second.
4. **Symmetric Difference (`^` or `symmetric_difference()`)**: Elements in either set, but not in both.
5. **Modification**:
    - `add(elem)`: Adds an element.
    - `remove(elem)`: Removes an element (raises KeyError if not found).
    - `discard(elem)`: Removes an element safely (no error if not found).

Worked Example:

Applying Set Operations Let A be the set of students who play Cricket `{"Alice", "Bob", "Charlie", "David"}` and B be the set of students who play Football `{"Charlie", "David", "Eve", "Frank"}`. Find:

1. Students who play both sports.
2. Students who play only Cricket.
3. Students who play exactly one of the two sports.
4. The total distinct list of students playing either sport.

Solution: Let $A = \{\text{Alice}, \text{Bob}, \text{Charlie}, \text{David}\}$ and $B = \{\text{Charlie}, \text{David}, \text{Eve}, \text{Frank}\}$.

1. **Both sports (Intersection)**:

```
A & B = {"Charlie", "David"}
```

2. **Only Cricket (Difference)**:

```
A - B = {"Alice", "Bob"}
```

3. **Exactly one sport (Symmetric Difference)**:

```
A ^ B = {"Alice", "Bob", "Eve", "Frank"}
```

4. **Either sport (Union)**:

```
A | B = {"Alice", "Bob", "Charlie", "David", "Eve", "Frank"}
```

5.5 Dictionaries

Methodology:

Dictionary Operations and Methods Dictionaries are mutable, ordered collections of key-value pairs. Keys must be immutable and unique.

1. **Accessing Values**: Use `dict[key]` (raises error if not found) or `dict.get(key, default)` (returns `default` safely).
2. **Updating**: Assign directly `dict[key] = value` or use `dict.update(other_dict)` to merge.
3. **Iteration Methods**:
   - `keys()`: Returns a view of all keys.
   - `values()`: Returns a view of all values.
   - `items()`: Returns a view of all `(key, value)` tuple pairs.
4. **Removal**:
   - `pop(key)`: Removes the key and returns its value.
   - `popitem()`: Removes and returns the last inserted key-value pair.
5. **Dictionary Comprehension**: `{key_exp: value_exp for item in iterable if condition}`.

Worked Example:

Managing Dictionary Data You are given a dictionary containing employee salaries: `salary_dict = {"John": 50000, "Emma": 60000, "Harry": 45000}`.

1. Retrieve Emma's salary using a safe method that returns 0 if the employee is not found.
2. Add a new employee "Sophia" with a salary of 55000.
3. Increase Harry's salary by 10%.
4. Create a new dictionary containing only the employees earning 55000 or more using dictionary comprehension.

Solution:

1. **Safe Access**: Use the `get()` method.

   `salary_dict.get("Emma", 0)` → `60000`

2. **Add New Element**:

   `salary_dict["Sophia"] = 55000`

   The dictionary becomes `{"John": 50000, "Emma": 60000, "Harry": 45000, "Sophia": 55000}`.

3. **Update Existing Element**:

   `salary_dict["Harry"] = salary_dict["Harry"] * 1.10`

   Harry's new salary becomes $45000 \times 1.10 = 49500$.

4. **Dictionary Comprehension**: Filter pairs where the value ≥ 55000.

   `high_earners = {k: v for k, v in salary_dict.items() if v >= 55000}`

   Evaluating values: John(50000), Emma(60000), Harry(49500), Sophia(55000).
   `high_earners` → `{"Emma": 60000, "Sophia": 55000}`

Appendix: Key Formulae

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

Financial Mathematics

Simple Interest

The simple interest (SI) accrued on a principal amount (P) at an annual interest rate (R) over a time period (T) is calculated as:

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

Combinatorics and Sequences

Factorial

The factorial of a non-negative integer n, denoted by $n!$, is the product of all positive integers less than or equal to n:

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

It can also be defined recursively as:

$$n! = n \times (n-1)! \quad \text{for } n > 0, \text{ with } 0! = 1$$

Fibonacci Sequence

The Fibonacci sequence is a series of numbers where each number is the sum of the two preceding ones. It is defined by the recurrence relation:

$$F_n = F_{n-1} + F_{n-2}$$

with initial conditions:

$$F_0 = 0, \quad F_1 = 1$$

Programming Logic and Arithmetic

Even Number Verification

An integer N is even if and only if it is completely divisible by 2, meaning the remainder is zero:

$$N \pmod 2 == 0$$

Integer Division and Modulo

In integer arithmetic, dividing a dividend a by a divisor n yields a quotient q and a remainder r:

$$a = n \times q + r$$

Where q is the integer quotient and r is the remainder. In Python, these are obtained using the `//` and `%` operators respectively:

$$q = a \mathbin{//} n$$
$$r = a \mathbin{\%} n$$

Bitwise Left Shift

A bitwise left shift of an integer x by n bits is equivalent to multiplying x by 2^n:

$$x \ll n = x \times 2^n$$

Number Systems

Binary to Decimal Conversion

To convert a binary number to decimal, each bit is multiplied by the corresponding power of 2. For example, a 4-bit binary number $b_3 b_2 b_1 b_0$ evaluates to:

$$\text{Decimal} = (b_3 \times 2^3) + (b_2 \times 2^2) + (b_1 \times 2^1) + (b_0 \times 2^0)$$