Python (1323203) - Problem Solver

Antigravity AI

June 4, 2026

Contents

CHAPTER 1

Introduction to Python and Basic Operations

This unit covers foundational computational techniques in Python. The focus is strictly on executing input and output operations, evaluating mathematical formulas, type casting, and string manipulations.

1.1 Basic Input and Output

Methodology:

Input and Output Operations To read data and display results in Python, follow these steps:

1. Use the `input(prompt)` function to read text from the user. The `prompt` is an optional string displayed before input.
2. Store the returned value in a variable. Note that `input()` always reads data as a string type.
3. Use the `print(value)` function to output values to the screen.
4. To embed variables inside output strings, use an f-string by placing an `f` before the quotation marks and enclosing variables in curly braces `{}`.

Worked Example:

Display User Information **Problem:** Write a Python sequence to read a user's name and age, and then print a greeting formatted exactly as "Hello [Name]! You are [Age] years old."
Solution:

1. Read the user's name from standard input:

   ```
   name = input("Enter your name: ")
   ```

2. Read the user's age:

   ```
   age = input("Enter your age: ")
   ```

3. Format and print the greeting using an f-string:

   ```
   print(f"Hello {name}! You are {age} years old.")
   ```

1.2 Evaluating Arithmetic Expressions

Methodology:

Arithmetic Operations To compute numerical expressions in Python:

1. Map the mathematical formula to Python operators: Addition (`+`), Subtraction (`-`), Multiplication (`*`), Division (`/`), Floor Division (`//`), Modulo (`%`) and Exponentiation (`**`).
2. Ensure standard order of operations (PEMDAS). Use parentheses `()` to enforce a specific calculation order.
3. Assign the evaluated arithmetic expression to a new variable.

Worked Example:

Calculate Simple Interest **Problem:** Calculate the simple interest for a principal amount of 10000 at a rate of 5.5 percent for 3 years. Use the formula $SI = \frac{P \times R \times T}{100}$.
Solution:

1. Define variables for Principal (P), Rate (R), and Time (T):

```
P = 10000
R = 5.5
T = 3
```

2. Translate the mathematical formula into a Python expression:

```
SI = (P * R * T) / 100
```

3. Display the computed simple interest:

```
print(f"The Simple Interest is: {SI}")
```

Worked Example:

Calculate Area of a Circle **Problem:** Find the area of a circle with a radius of 7 units. The formula is $A = \pi \times r^2$. Use $\pi = 3.14159$.
Solution:

1. Assign values to the radius and Pi:

```
pi = 3.14159
radius = 7
```

2. Compute the area using the exponentiation operator (`**`) for r^2:

```
area = pi * (radius ** 2)
```

3. Output the resulting area:

```
print(f"The Area is: {area}")
```

1.3 Type Casting

Methodology:

Type Conversion When user input is read as a string but numerical calculation is required, type conversion (casting) must be applied:

1. Determine the target mathematical type: `int` for whole numbers, `float` for decimals.
2. Wrap the variable or value in the respective type conversion function: `int(value)` or `float(value)`.
3. Store the casted value in a variable before attempting arithmetic operations.

Worked Example:

Add Two Numbers from String Input **Problem:** Accept two numerical inputs from the user, convert them to integers, and compute their sum.
Solution:

1. Prompt for and capture the inputs (which default to strings):

```
num1_str = input("Enter first number: ")
num2_str = input("Enter second number: ")
```

2. Cast both string variables to integers:

```
num1 = int(num1_str)
num2 = int(num2_str)
```

3. Perform addition on the integer variables and print:

```
total = num1 + num2
print(f"The total sum is: {total}")
```

1.4 Basic String Manipulation

Methodology:

String Concatenation and Repetition Python permits algorithmic manipulation of string data:

1. **Concatenation:** Use the `+` operator to append one string to the end of another. Ensure both variables are strings.
2. **Repetition:** Use the `*` operator to duplicate a string multiple times. This operator strictly requires one string operand and one integer operand.

Worked Example:

Generate Formatted String Output **Problem:** Formulate a single string that consists of the word "Code" repeated 4 times, immediately followed by the phrase " and Deploy".
Solution:

1. Use the repetition operator to multiply the base string:

```
repeated_str = "Code" * 4
```

2. Use the concatenation operator to join the second string:

```
final_str = repeated_str + " and Deploy"
```

3. Print the concatenated result:

```
print(final_str)
```

CHAPTER 2

Introduction to Python

2.1 Basic Input and Output

Methodology:

Basic Output and Type Identification In Python programming we use specific functions to display data and check data types.

1. Use the `print()` function to output text or variable values to the screen. Multiple items can be passed separated by spaces.
2. Use the `type()` function to determine the underlying data type of a variable or value. Common data types include `int` (integer) `float` (decimal number) and `str` (string).

Worked Example:

Print student details and identify data types of variables **Solution Steps:**

1. Define variables with the required information representing different data types.
2. Print the information using the `print()` function.
3. Use the `type()` function inside a `print()` statement to show the data types.

Python Code:

```
# Define variables
name = "John Doe"
mobile = 9876543210
height = 5.9

# Print details
print("Name:", name)
print("Mobile Number:", mobile)
print("Height:", height)

# Identify and print data types
print("Type of name:", type(name))
print("Type of mobile:", type(mobile))
print("Type of height:", type(height))
```

Output:

```
Name: John Doe
Mobile Number: 9876543210
Height: 5.9
Type of name: <class 'str'>
Type of mobile: <class 'int'>
Type of height: <class 'float'>
```

2.2 Arithmetic Operations

Methodology:

Performing Arithmetic Computations Python uses operators to perform mathematical calculations on variables and values.

1. Assign values to variables using the assignment operator `=`.
2. Apply basic arithmetic operators: Addition (`+`) Subtraction (`-`) Multiplication (`*`) Division (`/`) Floor Division (`//`) Modulo (`%`) and Exponentiation (`**`).
3. Formulate the mathematical expression using correct precedence.
4. Evaluate the expression and store or print the final result.

Worked Example:

Calculate simple interest and compound interest **Problem Statement:** Calculate Simple Interest (SI) and Compound Interest (CI) for Principal (P) = 10000 Rate (R) = 5% and Time (T) = 3 years.
Formulas:

- Simple Interest = $\frac{P \times R \times T}{100}$
- Amount = $P \times \left(1 + \frac{R}{100}\right)^T$
- Compound Interest = Amount − P

Solution Steps:

1. Initialize variables `P` `R` and `T` with given values.
2. Calculate Simple Interest using the formula `(P * R * T) / 100`.
3. Calculate Compound Interest using the exponentiation operator `**`.
4. Print both calculated interests.

Python Code:

```
# Given data
P = 10000  # Principal amount
R = 5.0    # Rate of interest
T = 3      # Time in years

# Calculate Simple Interest
SI = (P * R * T) / 100

# Calculate Compound Interest
amount = P * ( (1 + R / 100) ** T )
CI = amount - P

# Print results
print("Simple Interest:", SI)
print("Compound Interest:", CI)
```

Output:

```
Simple Interest: 1500.0
Compound Interest: 1576.25
```

2.3 User Input and Type Conversion

Methodology:

Handling Input and Type Casting User input in Python is read as a string by default. Arithmetic computations require converting this input to appropriate numerical types.

1. Use `input("Prompt message")` to accept input from the user.
2. Apply explicit type conversion using functions like `int()` to convert strings to integers or `float()` to convert strings to floating-point numbers.
3. Perform the required mathematical operations on the numeric variables.
4. Use `print()` to display the output.

Worked Example:

Read three numbers from the user and find their average **Solution Steps:**

1. Read three inputs from the user using the `input()` function.
2. Convert the input strings to floating-point numbers using `float()`.
3. Calculate the sum of the three numbers.
4. Divide the sum by 3 to find the average.
5. Print the calculated average.

Python Code:

```
# Read input and convert to float
num1 = float(input("Enter first number: "))
num2 = float(input("Enter second number: "))
num3 = float(input("Enter third number: "))

# Calculate average
total_sum = num1 + num2 + num3
average = total_sum / 3

# Print the result
print("The average is:", average)
```

Worked Example:

Convert temperature from Fahrenheit to Celsius **Problem Statement:** Convert temperature from Fahrenheit to Celsius using the equation: $C = (F - 32)/1.8$.
Solution Steps:

1. Read the temperature in Fahrenheit from the user.
2. Convert the user input to a `float`.
3. Apply the formula `C = (F - 32) / 1.8`.
4. Print the temperature in Celsius.

Python Code:

```
# Read Fahrenheit temperature from user
F = float(input("Enter temperature in Fahrenheit: "))

# Convert to Celsius
C = (F - 32) / 1.8

# Print result
print("Temperature in Celsius:", C)
```

CHAPTER 3
Flow of Control

Control structures determine the order in which statements are executed in a Python program. This chapter focuses on selection and repetition statements to solve computational problems.

3.1 Introduction to Flow of Control

In a standard Python program, execution flows sequentially line by line. However, complex computational problems require altering this flow based on specific conditions or executing blocks of code multiple times. This is achieved using **Selection Structures** (Decision Making) and **Repetition Structures** (Looping).

3.2 Selection Structures

Methodology:

If and If-Else Statements The `if` statement executes a block of code only if a specified condition evaluates to `True`. The `if-else` statement provides an alternative block of code when the condition is `False`.
Pseudocode:

```
IF condition THEN
    Execute Statement(s)
ELSE
    Execute Alternative Statement(s)
ENDIF
```

Algorithm Steps:

1. Formulate a boolean condition that evaluates to `True` or `False`.
2. Write the `if` keyword followed by the condition and a colon (`:`).
3. Indent the block of code to be executed when the condition is true.
4. (Optional) Write the `else:` keyword followed by an indented block of code to execute when the condition is false.

Worked Example:

Check for Even or Odd Number Write a Python program to check whether a given number is even or odd.
Step 1: Read the input from the user.

```
number = int(input("Enter a number: "))
```

Step 2: Apply the if-else logic using the modulo operator. A number is even if it leaves a remainder of 0 when divided by 2.

```
if number % 2 == 0:
```

```
    print(f"{number} is an Even number.")
else:
    print(f"{number} is an Odd number.")
```

Output for Input 15:

```
Enter a number: 15
15 is an Odd number.
```

Methodology:

Elif and Nested If Statements The `elif` (else if) statement allows checking multiple conditions sequentially. A nested `if` statement is an `if` statement placed inside another `if` or `else` block for multi-level decision making.

Pseudocode for Multiple Selections:

```
IF condition1 THEN
    Execute Block 1
ELSE IF condition2 THEN
    Execute Block 2
ELSE
    Execute Default Block
ENDIF
```

Algorithm Steps:

1. Start with an `if` statement for the primary condition.
2. Use one or more `elif` statements for subsequent conditions.
3. End with an optional `else` block to handle all remaining cases.
4. To nest conditions check a secondary condition inside an already indented block.

Worked Example:

Determine Grade Based on Marks Write a Python program to determine the grade of a student based on marks.

Step 1: Read the marks.

```
marks = float(input("Enter marks: "))
```

Step 2: Apply elif statements to evaluate the grade sequentially.

```
if marks >= 90:
    grade = 'A'
elif marks >= 80:
    grade = 'B'
elif marks >= 70:
    grade = 'C'
else:
    grade = 'Fail'

print(f"Your grade is: {grade}")
```

Worked Example:

Find Largest of Three Numbers Using Nested If Write a program to find the largest among three distinct numbers using nested `if` statements.

Step 1: Initialize three numbers.

```
a = 12
b = 25
c = 19
```

Step 2: Apply nested logic to isolate the maximum value.

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

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

3.3 Repetition Structures

Methodology:

For Loop Iteration The `for` loop is used to iterate over a sequence (such as a string, list or range) and execute a block of code for each element.

Pseudocode:

```
FOR EACH element IN sequence
    Execute Statement(s)
ENDFOR
```

Algorithm Steps:

1. Use the `for` keyword followed by an iterator variable.
2. Use the `in` keyword followed by an iterable sequence or the `range()` function.
3. The `range(start, stop, step)` function generates numbers from `start` up to (but not including) `stop` incrementing by `step`.
4. Write a colon and indent the block of code to repeatedly execute.

Worked Example:

Generate a Multiplication Table Write a program to generate the multiplication table for a given number up to 10.

Step 1: Define the target number.

```
n = 5
```

Step 2: Iterate over the range 1 to 10. The loop needs to run from 1 to 10 so we use `range(1, 11)`.

```
for i in range(1, 11):
```

```
    result = n * i
    print(f"{n} x {i} = {result}")
```

Worked Example:

Calculate Factorial of a Number Write a program to calculate the factorial of a positive integer `N` ($N! = N \times (N-1) \times \cdots \times 1$).

Step 1: Initialize the accumulator variable.

```
N = 5
factorial = 1
```

Step 2: Multiply sequentially using a for loop.

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

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

Methodology:

While Loop Iteration The `while` loop repeatedly executes a block of code as long as a given boolean condition remains `True`.

Pseudocode:

```
WHILE condition IS TRUE DO
    Execute Statement(s)
    Update loop control variable
ENDWHILE
```

Algorithm Steps:

1. Initialize a loop control variable before starting the loop.
2. Write the `while` keyword followed by a condition involving the control variable.
3. Indent the block of code to execute.
4. Critically ensure you update the control variable inside the loop to avoid infinite loops.

Worked Example:

Reverse a Number Write a program to reverse the digits of an integer using a while loop.

Step 1: Initialize the number and the output variable.

```
num = 1234
reverse_num = 0
```

Step 2: Extract digits iteratively.

```
while num > 0:
    remainder = num % 10
    reverse_num = (reverse_num * 10) + remainder
    num = num // 10
```

Step 3: Output the reversed number.

```
print(f"Reversed Number: {reverse_num}")
```

Methodology:

Nested Loops A nested loop is a loop placed inside the body of another loop. The inner loop executes all of its iterations completely for every single iteration of the outer loop.
Pseudocode:

```
FOR i FROM 1 TO N DO
    FOR j FROM 1 TO M DO
        Execute Inner Statement(s)
    ENDFOR
    Execute Outer Statement(s)
ENDFOR
```

Algorithm Steps:

1. Define the outer loop to handle rows or major passes.
2. Inside the outer loop define an inner loop to handle columns or minor operations.
3. Place the core logic inside the innermost level of indentation.

Worked Example:

Print a Pattern of Stars Write a program to print a right-angled triangle pattern of stars with 5 rows.
Step 1: Set up the outer loop for rows.

```
rows = 5
for i in range(1, rows + 1):
```

Step 2: Set up the inner loop for columns. The number of columns in each row corresponds exactly to the row number `i`.

```
    for j in range(1, i + 1):
        print("*", end=" ")
    print("") # Move to the next line after completing a row
```

Output:

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

3.4 Break and Continue Statements

Methodology:

Break and Continue Logic Control flow can be further manipulated using jump statements.

- **Break:** Terminates the current loop completely and passes control to the next statement outside the loop.
- **Continue:** Skips the remaining code inside the loop for the current iteration only and immediately moves to evaluate the loop condition for the next iteration.

Algorithm Steps:

1. Place an `if` condition inside the loop block to monitor for a specific trigger state.

2. Execute `break` to exit the loop entirely.
3. Execute `continue` to skip processing and proceed to the next iteration.

Worked Example:

Exit Loop Prematurely Using Break Write a program to find the first number divisible by both 4 and 7 within the range 1 to 50.

Step 1: Iterate through the range.

```
for i in range(1, 51):
```

Step 2: Apply condition and trigger break.

```
    if i % 4 == 0 and i % 7 == 0:
        print(f"Found the first number: {i}")
        break
```

Output:

```
Found the first number: 28
```

Worked Example:

Skip Specific Iterations Using Continue Write a program to print all numbers from 1 to 10 excluding the multiples of 3.

Step 1: Iterate through the range.

```
for i in range(1, 11):
```

Step 2: Apply condition and trigger continue.

```
    if i % 3 == 0:
        continue
    print(i, end=" ")
```

Output:

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

CHAPTER 4
Functions

Functions are reusable blocks of code that perform a specific task. They help in breaking down complex problems into smaller, manageable sub-problems, improving code organization and reusability.

4.1 User-Defined Functions

User-defined functions are created by the programmer to perform specific operations that are not available in built-in libraries.

Methodology:

Creating and Calling a Function To define and execute a user-defined function in Python, follow these steps:

1. **Define the Function:** Use the `def` keyword followed by the function name and parentheses `()`.
2. **Specify Parameters (Optional):** Inside the parentheses, list any parameters (variables) the function expects to receive.
3. **Write the Function Body:** Add a colon `:` and indent the code block that makes up the function's logic.
4. **Return a Value (Optional):** Use the `return` statement to send a result back to the caller. If omitted, the function returns `None`.
5. **Call the Function:** Execute the function by writing its name followed by arguments inside parentheses.

Syntax:

```
def function_name(parameter1, parameter2):
    # Function body
    return result

# Calling the function
output = function_name(arg1, arg2)
```

Worked Example:

Calculating Simple Interest **Problem Statement:** Write a Python function that takes principal, rate of interest, and time in years as arguments and returns the simple interest.
Solution:

1. Define a function named `calculate_simple_interest` with parameters `p`, `r`, and `t`.
2. Compute the interest using the formula $SI = \frac{p \times r \times t}{100}$.
3. Return the computed interest.
4. Call the function with sample values.

Python Code:

```
def calculate_simple_interest(p, r, t):
    si = (p * r * t) / 100
    return si

# Calling the function
principal = 10000
rate = 5
time = 3
interest = calculate_simple_interest(principal, rate, time)
print("Simple Interest is:", interest)
```

Output:

```
Simple Interest is: 1500.0
```

4.2 Arguments and Parameters

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

Methodology:

Handling Different Types of Arguments Functions in Python can accept arguments in several ways:

1. **Positional Arguments:** Passed in the exact order as defined in the function signature.
2. **Keyword Arguments:** Passed by explicitly specifying the parameter name (e.g., `param=value`), allowing arguments to be passed out of order.
3. **Default Arguments:** Parameters can have a default value assigned during definition using `=`. If no argument is passed during the call, the default value is used.
4. **Variable-Length Arguments:**
   - `*args`: Allows passing a variable number of positional arguments as a tuple.
   - `**kwargs`: Allows passing a variable number of keyword arguments as a dictionary.

Worked Example:

Calculating Net Price with Default and Keyword Arguments **Problem Statement:** Write a function to calculate the final price of an item after applying a discount and tax. The default discount should be 10% and default tax should be 5%. Demonstrate calling the function using positional and keyword arguments.
Solution:

1. Define a function `calculate_net_price` with parameters `price`, `discount=10`, and `tax=5`.
2. Compute the discounted price: $\text{discounted} = \text{price} - (\text{price} \times \frac{\text{discount}}{100})$.
3. Compute the final price: $\text{final} = \text{discounted} + (\text{discounted} \times \frac{\text{tax}}{100})$.
4. Return the final price.

Python Code:

```
def calculate_net_price(price, discount=10, tax=5):
    discounted_price = price - (price * (discount / 100))
    final_price = discounted_price + (discounted_price * (tax / 100))
    return final_price
```

```
# Call 1: Using default discount and tax
print(calculate_net_price(1000))

# Call 2: Providing a specific discount (positional argument)
print(calculate_net_price(1000, 20))

# Call 3: Providing specific tax using keyword argument
print(calculate_net_price(1000, tax=8))
```

Output:

```
945.0
840.0
972.0
```

4.3 Scope of a Variable

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

Methodology:

Managing Local and Global Scope

1. **Local Variable:** A variable defined inside a function. It can only be accessed within that specific function. Once the function executes, the variable is destroyed.
2. **Global Variable:** A variable defined outside any function. It can be accessed anywhere in the program, including inside functions.
3. **The `global` Keyword:** If you need to modify a global variable from inside a function, you must declare it using the `global` keyword before modifying it.

Worked Example:

Modifying Global Variables within Functions **Problem Statement:** Write a program that maintains a global counter variable. Create a function to increment this counter and another function to print its current value.
Solution:

1. Initialize a global variable `counter = 0`.
2. Define `increment_counter()` that uses the `global` keyword to modify the `counter`.
3. Define `display_counter()` to simply print the `counter` (no `global` keyword needed just to read).

Python Code:

```
counter = 0  # Global variable

def increment_counter():
    global counter
    counter += 1

def display_counter():
    print("Current counter value:", counter)

display_counter()
```

```
increment_counter()
increment_counter()
display_counter()
```

Output:

```
Current counter value: 0
Current counter value: 2
```

4.4 Python Standard Library Built-in Functions

Python provides several built-in functions for fundamental input, output, and mathematical operations.

Methodology:

Utilizing Common Built-in Functions

- `print(data)`: Outputs data to the console.
- `input(prompt)`: Reads a line of input from the user as a string.
- `abs(x)`: Returns the absolute (positive) value of a number x.
- `divmod(a, b)`: Returns a tuple containing the quotient and remainder of a divided by b, i.e., `(a // b, a % b)`.
- `max(iterable, *args)`: Returns the largest item in an iterable or among two or more arguments.
- `min(iterable, *args)`: Returns the smallest item.
- `pow(base, exp)`: Returns `base` raised to the power of `exp` ($base^{exp}$).
- `sum(iterable)`: Returns the sum of all items in an iterable (like a list or tuple).

Worked Example:

Applying Built-in Mathematical Functions **Problem Statement:** Given a list of numbers `[-15, 20, -5, 30, 10]`, write a script to find the maximum value, minimum value, absolute value of the first element, sum of all elements, and perform a `divmod` operation on the maximum and minimum values.
Solution: Python Code:

```
numbers = [-15, 20, -5, 30, 10]

# 1. Find max and min
maximum = max(numbers)
minimum = min(numbers)

# 2. Find absolute value
absolute_first = abs(numbers[0])

# 3. Sum of elements
total_sum = sum(numbers)

# 4. Quotient and remainder using divmod
# Dividing max by the absolute value of min
quotient, remainder = divmod(maximum, abs(minimum))

print("Maximum:", maximum)
```

```
print("Minimum:", minimum)
print("Absolute of first:", absolute_first)
print("Sum:", total_sum)
print(f"divmod({maximum}, {abs(minimum)}): Quotient={quotient}, Remainder={remainder}")
```

Output:

```
Maximum: 30
Minimum: -15
Absolute of first: 15
Sum: 40
divmod(30, 15): Quotient=2, Remainder=0
```

4.5 Modules math random and statistics

Modules in Python are files containing Python definitions and statements. They allow you to logically organize your Python code.

4.5.1 The math Module

The `math` module provides access to mathematical functions defined by the C standard.

Methodology:

Using the math Module First, import the module using `import math`. Common functions and constants include:

- `math.pi`: Mathematical constant π (3.14159...).
- `math.sqrt(x)`: Returns the square root of x.
- `math.factorial(x)`: Returns $x!$ (x factorial).
- `math.ceil(x)`: Returns the smallest integer greater than or equal to x.
- `math.floor(x)`: Returns the largest integer less than or equal to x.

Worked Example:

Calculating Circle Area and Triangle Hypotenuse **Problem Statement:** Write a script using the `math` module to compute the area of a circle with a radius of 7, and the hypotenuse of a right-angled triangle with base 3 and height 4.
Solution: Python Code:

```
import math

# Area of a circle = pi * r * r
radius = 7
area = math.pi * pow(radius, 2)
print("Area of circle:", area)

# Hypotenuse = sqrt(base^2 + height^2)
base = 3
height = 4
hypotenuse = math.sqrt(pow(base, 2) + pow(height, 2))
print("Hypotenuse of triangle:", hypotenuse)
```

Output:

```
Area of circle: 153.93804002589985
Hypotenuse of triangle: 5.0
```

4.5.2 The random Module

The `random` module implements pseudo-random number generators for various distributions.

Methodology:

Using the random Module Import using `import random`. Common functions include:

- `random.random()`: Returns a random floating-point number in the range $[0.0, 1.0)$.
- `random.randint(a, b)`: Returns a random integer N such that $a \leq N \leq b$.
- `random.choice(sequence)`: Returns a randomly selected element from a non-empty sequence (like a list or string).

Worked Example:

Simulating a Dice Roll and Coin Toss **Problem Statement:** Write a program to simulate rolling a standard 6-sided die and flipping a coin.
Solution: Python Code:

```
import random

# Simulating a 6-sided dice roll
dice_roll = random.randint(1, 6)
print("You rolled a:", dice_roll)

# Simulating a coin toss
coin = ["Heads", "Tails"]
toss_result = random.choice(coin)
print("Coin toss result:", toss_result)
```

Output:

```
You rolled a: 4
Coin toss result: Heads
```

4.5.3 The statistics Module

The `statistics` module provides functions to calculate mathematical statistics of numeric data.

Methodology:

Using the statistics Module Import using `import statistics`. Common functions include:

- `statistics.mean(data)`: Returns the arithmetic mean (average) of the data.
- `statistics.median(data)`: Returns the median (middle value) of the data.
- `statistics.mode(data)`: Returns the single most common data point from discrete or nominal data.

Worked Example:

Calculating Mean Median and Mode **Problem Statement:** Given a list of student marks `[75, 82, 75, 90, 85, 82, 75, 95]`, calculate the mean, median, and mode using the `statistics` module.

Solution: Python Code:

```
import statistics

marks = [75, 82, 75, 90, 85, 82, 75, 95]

mean_marks = statistics.mean(marks)
median_marks = statistics.median(marks)
mode_marks = statistics.mode(marks)

print("Mean:", mean_marks)
print("Median:", median_marks)
print("Mode:", mode_marks)
```

Output:

```
Mean: 82.375
Median: 82.0
Mode: 75
```

CHAPTER 5

Dictionary List Set String and Tuple

5.1 String Operations

Strings in Python are immutable sequences of characters. Problem-solving with strings often involves iterating through characters and applying fundamental operations.

Methodology:

Basic String Operations **1. Concatenation and Repetition:** Strings can be concatenated using the `+` operator and repeated using the `*` operator.

- `"a" + "b"` results in `"ab"`
- `"a" * 3` results in `"aaa"`

2. Indexing and Slicing: Access individual characters using `s[i]` and substrings using `s[start:end:step]`. Negative indices count from the end of the string.
3. Traversing a String: Use a `for` loop to iterate over characters sequentially.

```
for char in s:
    # process char
```

Worked Example:

Traverse and Count Character Types **Problem Statement:** Write a Python program to traverse a string and count the number of alphabets and digits.
Step 1: Initialize counters

```
def analyze_string(text):
    alpha_count = 0
    digit_count = 0
```

Step 2: Traverse the string

```
    for char in text:
        if char.isalpha():
            alpha_count += 1
        elif char.isdigit():
            digit_count += 1

    return alpha_count, digit_count
```

Step 3: Execute the function

```
result = analyze_string("Python 3.10")
print("Alphabets:", result[0])
print("Digits:", result[1])
```

5.2 String Manipulation

String manipulation relies heavily on Python's built-in string methods to format and transform text data efficiently.

Methodology:

Built-in String Functions and Methods **1. Case Manipulation:**

- `s.lower()` and `s.upper()`: Convert all characters to lower or upper case.
- `s.capitalize()` and `s.title()`: Capitalize the first letter or every word.

2. Searching and Replacing:

- `s.find(sub)`: Returns the lowest index of the substring or `-1` if not found.
- `s.replace(old, new)`: Replaces all occurrences of the old substring with the new one.

3. Formatting and Splitting:

- `s.strip()`: Removes leading and trailing whitespace.
- `s.split(delim)`: Splits the string into a list of strings based on the delimiter.
- `delim.join(iterable)`: Joins elements of an iterable into a single string.

Worked Example:

Format and Extract Information from a String **Problem Statement:** Given an unformatted string of comma-separated values with extra spaces, extract the words and join them with hyphens.

Step 1: Define the raw string

```
raw_data = "  apple , banana,  orange   , grape   "
```

Step 2: Split the string by commas

```
items = raw_data.split(',')
```

Step 3: Strip whitespace from each item

```
cleaned_items = []
for item in items:
    cleaned_items.append(item.strip())
```

Step 4: Join the cleaned items using a hyphen

```
result = "-".join(cleaned_items)
print(result)
```

Output:

```
apple-banana-orange-grape
```

5.3 List Operations

Lists are mutable sequences that can hold elements of any data type. Problem-solving with lists often involves sorting, filtering, and performing aggregate operations.

Methodology:

List Manipulation Methods **1. Adding Elements:**

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

2. Removing Elements:

- `lst.pop(i)`: Removes and returns the item at index `i`.
- `lst.remove(x)`: Removes the first occurrence of item `x`.

3. Ordering and Aggregate Built-in Functions:

- `lst.sort()`: Sorts the list in-place.
- `lst.reverse()`: Reverses the list in-place.
- `len(lst)` or `max(lst)` or `min(lst)` or `sum(lst)`.

Worked Example:

Find the Largest and Second Largest Element **Problem Statement:** Given a list of integers, find the largest and second largest elements.

Step 1: Initialize list and variables

```
numbers = [12, 45, 2, 41, 31, 10, 8, 45]
largest = float('-inf')
second_largest = float('-inf')
```

Step 2: Iterate through the list Update the largest and second largest values sequentially.

```
for num in numbers:
    if num > largest:
        second_largest = largest
        largest = num
    elif num > second_largest and num != largest:
        second_largest = num
```

Step 3: Print the results

```
print("Largest:", largest)
print("Second Largest:", second_largest)
```

Output:

```
Largest: 45
Second Largest: 41
```

5.4 Set Manipulation

Sets are unordered, mutable collections of **unique** elements. They are highly optimized for mathematical set operations.

Methodology:

Set Operations and Mathematical Methods **1. Creating and Updating Sets:** Use `{1, 2, 3}` or `set()`. Add items with `s.add(x)` or remove them with `s.discard(x)`.
2. Mathematical Set Operations:

- **Union ($A \cup B$):** `A | B` or `A.union(B)` (Elements in either A or B).
- **Intersection ($A \cap B$):** `A & B` or `A.intersection(B)` (Elements in both A and B).
- **Difference ($A - B$):** `A - B` or `A.difference(B)` (Elements in A but not in B).
- **Symmetric Difference ($A \Delta B$):** `A ^ B` or `A.symmetric_difference(B)` (Elements in exactly one of the sets).

Worked Example:

Identify Unique and Common Elements **Problem Statement:** Given two lists of student IDs enrolled in Math and Science, find students enrolled in both, and students enrolled in exactly one subject.
Step 1: Convert lists to sets

```
math = {101, 102, 103, 104, 105}
science = {103, 104, 106, 107}
```

Step 2: Find common students using Intersection

```
common = math.intersection(science)
```

Step 3: Find students in exactly one subject using Symmetric Difference

```
unique_to_one = math.symmetric_difference(science)
```

Step 4: Display results

```
print("Enrolled in both:", common)
print("Enrolled in exactly one:", unique_to_one)
```

5.5 Tuple Operations

Tuples are ordered collections similar to lists, but they are **immutable**. Once created, their contents cannot be changed.

Methodology:

Tuple Creation and Operations **1. Creating Tuples:** Use parentheses: `t = (1, 2, 3)`. A single-element tuple requires a trailing comma: `t = (1,)`.
2. Accessing Elements: Like lists and strings, use zero-based indexing and slicing: `t[0]`, `t[1:3]`.
3. Tuple Packing and Unpacking:

- **Packing:** `t = 1, 2, "hello"` (Parentheses are optional).
- **Unpacking:** `a, b, c = t` assigns tuple elements to variables directly.

4. Built-in Functions and Methods: `len(t)`, `max(t)`, `t.count(x)` and `t.index(x)`.

Worked Example:

Swap Variables and Unpack a Tuple **Problem Statement:** Demonstrate how to swap two variables using tuples and extract specific values from a tuple representing a student record.

Step 1: Variable swapping using tuple packing and unpacking

```
a = 10
b = 20
a, b = b, a  # Packs (b, a) and unpacks into a, b
print("a:", a, "b:", b)
```

Step 2: Unpacking a student record Assume a tuple formatted as `(ID, Name, Age, Grade)`.

```
student = (101, "Alice", 19, "A")
std_id, name, age, grade = student

print("ID:", std_id)
print("Name:", name)
```

5.6 Dictionary Manipulation

Dictionaries are mutable collections of **key-value pairs**. Keys must be immutable and unique, while values can be of any type.

Methodology:

Dictionary Operations and Methods **1. Creation and Access:** Create dictionaries using curly braces: `d = {"a": 1, "b": 2}`. Access values via keys: `val = d["a"]`. Use `d.get(key, default)` to avoid KeyError.
2. Adding and Updating Items: Assign a value to a key: `d["c"] = 3`. Use `d.update({"d": 4})` to merge another dictionary.
3. Removing Items:

- `del d[key]`: Removes the key-value pair.
- `d.pop(key)`: Removes the pair and returns the value.

4. Iterating over Dictionaries:

- `d.keys()`: Returns all keys.
- `d.values()`: Returns all values.
- `d.items()`: Returns all key-value tuples.

Worked Example:

Count Word Frequencies **Problem Statement:** Write a script to count the frequency of each word in a given text string using a dictionary.
Step 1: Clean and split the string

```
text = "apple banana apple orange banana apple"
words = text.split()
```

Step 2: Populate the frequency dictionary

```
word_count = {}
for word in words:
    if word in word_count:
        word_count[word] += 1
    else:
        word_count[word] = 1
```

Step 3: Display the frequency of each word

```
for key, value in word_count.items():
    print(key, ":", value)
```

Output:

```
apple : 3
banana : 2
orange : 1
```

Appendix: Key Formulae

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

Geometry

Area of a Circle
The area A of a circle is calculated using its radius r:

$$A = \pi \times r^2$$

Financial Mathematics

Simple Interest
Simple interest (SI) earned on a principal amount P at an annual interest rate R over a time period T (in years) is:

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

Compound Interest
The accumulated amount after compound interest is applied to a principal P at a rate R for time T is:

$$\text{Amount} = P \times \left(1 + \frac{R}{100}\right)^T$$

The compound interest itself is the difference between the accumulated amount and the original principal:

$$\text{Compound Interest} = \text{Amount} - P$$

Temperature Conversion

Fahrenheit to Celsius
To convert a temperature from degrees Fahrenheit (F) to degrees Celsius (C):

$$C = \frac{F - 32}{1.8}$$

Combinatorics

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

$$N! = N \times (N-1) \times \cdots \times 1$$

Retail Mathematics

Discount Price

The discounted price of an item, given its original price and a discount percentage, is calculated as:

$$\text{Discounted Price} = \text{Price} - \left(\text{Price} \times \frac{\text{Discount Percentage}}{100}\right)$$

Final Price with Tax

The final price after applying a tax percentage to the discounted price is calculated as:

$$\text{Final Price} = \text{Discounted Price} + \left(\text{Discounted Price} \times \frac{\text{Tax Percentage}}{100}\right)$$