Oops (4351108) - Problem Solver

Antigravity AI

June 4, 2026

Oops

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

Basics of Python

1.1 Variables and Data Types

In Python, variables are created dynamically when a value is assigned to them. Understanding how to store and manipulate different data types is the first step in solving computational problems.

Methodology:

Creating Variables and Assigning Values To create a variable and assign a value to it, follow these steps:

1. Choose a descriptive variable name consisting of letters, numbers, and underscores (e.g., `length`, `width`). The name cannot start with a number.
2. Use the assignment operator `=` to assign a value to the variable.
3. Python automatically infers the data type based on the assigned value. Common data types include:
   - **Integer:** Whole numbers (e.g., `10`, `-5`).
   - **Float:** Decimal numbers (e.g., `3.14`, `2.0`).
   - **String:** Text enclosed in quotes (e.g., `"Hello"`).
   - **Boolean:** Logical values (`True` or `False`).
4. Use the `type()` function to check the data type of a variable if needed.

Worked Example:

Computing Area of a Rectangle **Problem:** Write a Python script to define the length and width of a rectangle, compute its area, and display the result.
Solution:

1. **Step 1: Assign values to variables.**
   Let the length be 15.5 and the width be 10.

   ```
   length = 15.5
   width = 10
   ```

2. **Step 2: Formulate the equation.**
   The area of a rectangle is computed by multiplying its length by its width.

   ```
   area = length * width
   ```

3. **Step 3: Display the output.**
   Use the `print()` function to output the computed area.

```
print("The area of the rectangle is:", area)
```

4. **Final Output:**

```
The area of the rectangle is: 155.0
```

1.2 Operators and Expressions

Operators are special symbols that carry out arithmetic or logical computation. Expressions are combinations of variables, values, and operators that yield a result.

Methodology:

Evaluating Arithmetic Expressions To construct and evaluate mathematical expressions in Python:

1. Identify the values or variables involved in the calculation.
2. Apply the appropriate arithmetic operators:
   - `+` for Addition
   - `-` for Subtraction
   - `*` for Multiplication
   - `/` for Float Division (returns a decimal result)
   - `//` for Integer Division (returns the quotient without decimal)
   - `%` for Modulus (returns the remainder)
   - `**` for Exponentiation (power)
3. Python evaluates expressions based on the standard order of operations (PEMDAS or BODMAS): Parentheses, Exponentiation, Multiplication or Division, and Addition or Subtraction.

Worked Example:

Converting Temperature from Celsius to Fahrenheit **Problem:** Given a temperature of 28°C, convert it to Fahrenheit using Python.
Solution:

1. **Step 1: Define the Celsius variable.**

```
celsius = 28
```

2. **Step 2: Apply the conversion formula.**
   The formula to convert Celsius to Fahrenheit is: $F = (C \times \frac{9}{5}) + 32$.

```
fahrenheit = (celsius * 9 / 5) + 32
```

3. **Step 3: Execute the calculation.**
   First, $28 \times 9 = 252$.
   Next, $252/5 = 50.4$.
   Finally, $50.4 + 32 = 82.4$.
4. **Step 4: Output the result.**

```
print("Temperature in Fahrenheit:", fahrenheit)
```

Final Output:

```
Temperature in Fahrenheit: 82.4
```

1.3 Conditional Statements

Conditional statements allow the program to execute specific blocks of code depending on whether a logical condition evaluates to `True` or `False`.

Methodology:

Using If Elif Else for Decision Making To implement decision-making logic:

1. Start with the `if` keyword followed by a boolean condition and a colon `:`.
2. Indent the code block (usually 4 spaces) that should run if the condition is true.
3. Use `elif` (short for else if) to check subsequent conditions if the previous `if` condition was false.
4. Use `else` at the end to define a default code block that runs if all preceding conditions are false.

Worked Example:

Determining the Largest of Three Numbers **Problem:** Write a program to find the largest among three numbers: 45, 78, and 62.
Solution:

1. **Step 1: Define the three numbers.**

   ```
   num1 = 45
   num2 = 78
   num3 = 62
   ```

2. **Step 2: Write the conditional logic.**
   Compare `num1` with both `num2` and `num3`. If it is greater than both, it is the largest. Otherwise, check `num2`. If both fail, `num3` is the largest.

   ```
   if num1 >= num2 and num1 >= num3:
       largest = num1
   elif num2 >= num1 and num2 >= num3:
       largest = num2
   else:
       largest = num3
   ```

3. **Step 3: Trace the execution.**
   - Condition 1: `45 >= 78 and 45 >= 62` evaluates to `False`.
   - Condition 2: `78 >= 45 and 78 >= 62` evaluates to `True`. Thus, `largest` becomes 78.
4. **Step 4: Output the result.**

```
print("The largest number is:", largest)
```

Final Output:

```
The largest number is: 78
```

1.4 Looping Statements

Loops are used to execute a block of code repeatedly as long as a condition is met or for a specific number of iterations.

Methodology:

Iteration using For and While Loops Python provides two main types of loops:

1. **For Loop:** Used for iterating over a sequence (like a list string or range).
   - Use `for variable in sequence:` followed by an indented block.
   - The `range(start, stop, step)` function is often used to generate a sequence of numbers.
2. **While Loop:** Used to repeat a block of code as long as a boolean condition remains true.
   - Use `while condition:` followed by an indented block.
   - Ensure the loop variable is updated within the block to prevent an infinite loop.

Worked Example:

Finding the Sum of First N Natural Numbers **Problem:** Calculate the sum of the first 5 natural numbers (1 to 5) using a `for` loop.
Solution:

1. **Step 1: Initialize the sum variable.**

   ```
   total_sum = 0
   ```

2. **Step 2: Set up the loop.**
   We need numbers from 1 to 5. The `range(1, 6)` function generates values: 1, 2, 3, 4, 5.

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

3. **Step 3: Trace the iterations.**
   - Iteration 1: `i = 1`, `total_sum = 0 + 1 = 1`
   - Iteration 2: `i = 2`, `total_sum = 1 + 2 = 3`
   - Iteration 3: `i = 3`, `total_sum = 3 + 3 = 6`
   - Iteration 4: `i = 4`, `total_sum = 6 + 4 = 10`
   - Iteration 5: `i = 5`, `total_sum = 10 + 5 = 15`
4. **Step 4: Output the result.**

```
print("Sum of first 5 natural numbers:", total_sum)
```

Final Output:

```
Sum of first 5 natural numbers: 15
```

1.5 Functions

Functions are reusable blocks of code designed to perform a specific task. They help in breaking down complex problems into smaller modular pieces.

Methodology:

Defining and Calling Functions To define and use a custom function:

1. Start with the `def` keyword followed by the function name and parentheses `()`.
2. Include any parameters inside the parentheses. End the line with a colon `:`.
3. Write the indented function body containing the computational logic.
4. Use the `return` statement to pass the computed result back to the caller.
5. To execute the function, call its name and provide the required arguments inside parentheses.

Worked Example:

Calculating the Factorial of a Number **Problem:** Write a function to calculate the factorial of a given positive integer, and use it to find the factorial of 4.
Solution:

1. **Step 1: Define the function and parameter.**

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

2. **Step 2: Trace the function logic for n = 4.**
   The loop runs for `i` in `range(1, 5)`, which means `i` takes values 1, 2, 3, 4.
   - Initially, `result = 1`.
   - `i = 1`: `result = 1 * 1 = 1`
   - `i = 2`: `result = 1 * 2 = 2`
   - `i = 3`: `result = 2 * 3 = 6`
   - `i = 4`: `result = 6 * 4 = 24`
3. **Step 3: Call the function and print the result.**

```
ans = factorial(4)
print("Factorial of 4 is:", ans)
```

4. **Final Output:**

```
Factorial of 4 is: 24
```

CHAPTER 2

Lists Tuples Sets Dictionaries and Strings

2.1 Strings

Methodology:

String Manipulation and Methods Strings in Python are immutable sequences of characters.

1. **Indexing and Slicing**: Access individual characters using `s[i]` or a range using `s[start:stop:step]`.
2. **Concatenation and Repetition**: Use `+` to join strings and `*` to repeat them.
3. **Common Methods**:
   - `len(s)`: Returns the length of the string.
   - `s.lower()` / `s.upper()`: Converts to lowercase/uppercase.
   - `s.strip()`: Removes leading and trailing whitespaces.
   - `s.find(sub)`: Returns the lowest index of substring `sub` or `-1` if not found.
   - `s.replace(old new)`: Replaces occurrences of `old` with `new`.
   - `s.split(delim)`: Splits the string into a list based on `delim`.

Worked Example:

String Formatting and Substring Replacement Given a string `text = "Hello World! Welcome to Python Programming."` write Python code to: 1. Convert the entire string to uppercase. 2. Find the index of the word "Welcome". 3. Replace "Python" with "Java". 4. Split the modified string into a list of words.
Solution:

1. **Uppercase conversion**:

```
text_upper = text.upper()
# Output: "HELLO WORLD! WELCOME TO PYTHON PROGRAMMING."
```

2. **Finding substring index**:

```
index_welcome = text.find("Welcome")
# Output: 13
```

3. **Replacing substring**:

```
text_replaced = text.replace("Python", "Java")
# Output: "Hello World! Welcome to Java Programming."
```

4. **Splitting into list**:

```
words = text_replaced.split()
# Output: ['Hello', 'World!', 'Welcome', 'to', 'Java', 'Programming.']
```

2.2 Lists

Methodology:

List Operations and Comprehensions Lists are mutable ordered sequences.

1. **Adding Elements**:
   - `lst.append(x)`: Adds `x` to the end.
   - `lst.insert(i x)`: Inserts `x` at index `i`.
   - `lst.extend(iterable)`: Appends elements from an iterable.
2. **Removing Elements**:
   - `lst.remove(x)`: Removes the first occurrence of `x`.
   - `lst.pop(i)`: Removes and returns the element at index `i`.
3. **Sorting and Reversing**: `lst.sort()` sorts in place; `lst.reverse()` reverses in place.
4. **List Comprehensions**: A concise way to create lists: `[expr for item in iterable if condition]`.

Worked Example:

Filtering and Transforming a List Given a list of integers `nums = [12 5 21 8 15 30 7]` perform the following operations: 1. Append the number `42` to the list. 2. Remove the number `5` from the list. 3. Sort the list in descending order. 4. Use a list comprehension to create a new list containing only the even numbers from the modified list.

Solution:

1. **Appending an element**:

```
nums.append(42)
# nums becomes [12, 5, 21, 8, 15, 30, 7, 42]
```

2. **Removing an element**:

```
nums.remove(5)
# nums becomes [12, 21, 8, 15, 30, 7, 42]
```

3. **Sorting in descending order**:

```
nums.sort(reverse=True)
# nums becomes [42, 30, 21, 15, 12, 8, 7]
```

4. **List comprehension for even numbers**:

```
evens = [x for x in nums if x % 2 == 0]
# evens becomes [42, 30, 12, 8]
```

2.3 Tuples

Methodology:

Tuple Creation and Unpacking Tuples are immutable sequences often used for heterogeneous data.

1. **Creation**: Defined using parentheses `(a b c)` or simply comma-separated values `a b c`. A single-element tuple requires a trailing comma: `(a,)`.
2. **Access**: Indexed and sliced just like lists and strings.
3. **Unpacking**: Assigning tuple elements to multiple variables: `x y z = my_tuple`. Use `*var` to gather remaining items.
4. **Methods**: Since they are immutable tuples only support `count(x)` and `index(x)`.

Worked Example:

Tuple Manipulation and Data Extraction Given a tuple `student = (101 "Alice" 85.5 "Computer Science")`: 1. Extract the student ID and name using indexing. 2. Unpack the tuple into variables `roll_no name marks` and `branch`. 3. Create a new tuple by concatenating the `student` tuple with `("Passed",)`.
Solution:

1. **Indexing**:

```
student_id = student[0]      # 101
student_name = student[1]    # "Alice"
```

2. **Unpacking**:

```
roll_no, name, marks, branch = student
```

3. **Concatenation**:

```
updated_student = student + ("Passed",)
# Output: (101, 'Alice', 85.5, 'Computer Science', 'Passed')
```

2.4 Sets

Methodology:

Set Mathematical Operations Sets are unordered collections of unique elements.

1. **Creation**: Defined using curly braces `{1 2 3}` or the `set()` function. Note: `{}` creates an empty dictionary not a set.
2. **Adding/Removing**: Use `s.add(x)` `s.remove(x)` (raises error if not found) or `s.discard(x)` (no error if not found).
3. **Mathematical Operations**:
   - **Union** (`|`) or `s1.union(s2)`: All unique elements from both sets.
   - **Intersection** (`&`) or `s1.intersection(s2)`: Elements common to both sets.

- **Difference** (`-`) or `s1.difference(s2)`: Elements in `s1` but not in `s2`.
- **Symmetric Difference** (`^`) or `s1.symmetric_difference(s2)`: Elements in either set but not in both.

Worked Example:

Evaluating Set Operations Given two sets `A = {1 2 3 4 5}` and `B = {4 5 6 7 8}` compute: 1. The union of A and B. 2. The intersection of A and B. 3. The difference A - B. 4. The symmetric difference of A and B.

Solution:

1. **Union**:

```
res_union = A | B
# Output: {1, 2, 3, 4, 5, 6, 7, 8}
```

2. **Intersection**:

```
res_intersection = A & B
# Output: {4, 5}
```

3. **Difference**:

```
res_difference = A - B
# Output: {1, 2, 3}
```

4. **Symmetric Difference**:

```
res_sym_diff = A ^ B
# Output: {1, 2, 3, 6, 7, 8}
```

2.5 Dictionaries

Methodology:

Dictionary Management and Comprehensions Dictionaries are mutable unordered collections of key-value pairs. Keys must be immutable and unique.

1. **Accessing Values**: Use `d[key]` (raises error if key missing) or `d.get(key default)` (returns `default` if missing).
2. **Adding/Updating**: `d[key] = value` adds a new pair or updates an existing one. `d.update(other_dict)` merges another dictionary.
3. **Removing Elements**:
   - `del d[key]`: Deletes the key-value pair.
   - `d.pop(key)`: Removes the key and returns its value.
4. **Iteration Methods**:
   - `d.keys()`: Returns all keys.

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

5. **Dictionary Comprehensions**: `{k: v for k v in iterable if condition}`.

Worked Example:

Dictionary Data Aggregation Given a dictionary of employee salaries `salaries = {"Alice": 50000 "Bob": 45000 "Charlie": 60000}`: 1. Add a new employee "David" with a salary of 55000. 2. Update "Alice"'s salary to 52000. 3. Calculate the average salary of all employees. 4. Create a new dictionary containing only employees earning more than 50000 using dictionary comprehension.
Solution:

1. **Adding an entry**:

```
salaries["David"] = 55000
# salaries: {"Alice": 50000, "Bob": 45000, "Charlie": 60000, "David": 55000}
```

2. **Updating an entry**:

```
salaries["Alice"] = 52000
# salaries: {"Alice": 52000, "Bob": 45000, "Charlie": 60000, "David": 55000}
```

3. **Calculating average**:

```
total = sum(salaries.values())
count = len(salaries)
average = total / count
# total = 212000, count = 4, average = 53000.0
```

4. **Dictionary comprehension**:

```
high_earners = {k: v for k, v in salaries.items() if v > 50000}
# Output: {"Alice": 52000, "Charlie": 60000, "David": 55000}
```

CHAPTER 3
Functions and Modules

3.1 Defining and Calling Functions

Methodology:

Function Definition and Calling A function is a reusable block of code that performs a specific task.

1. **Definition:** Use the `def` keyword followed by the function name and parentheses `()`.
2. **Parameters:** Input variables inside the parentheses (optional).
3. **Body:** An indented block of code.
4. **Return Value:** Use the `return` statement to output a result (optional).
5. **Calling:** Use the function name followed by arguments in parentheses.

Syntax:

```
def function_name(parameters):
    # statements
    return result
```

Worked Example:

Calculate Factorial using a Function Write a Python function to compute the factorial of a given number and call it for $n = 5$.
Step 1: Define the function Use a loop to multiply numbers from 1 to n.

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

Step 2: Call the function and print the result

```
result = calculate_factorial(5)
print("Factorial of 5 is:", result)
```

Output:

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

3.2 Types of Function Arguments

Methodology:

Argument Types Python supports various types of arguments for function calls:

1. **Positional Arguments:** Arguments passed in the correct positional order.
2. **Keyword Arguments:** Arguments passed using the parameter name (`key=value`).
3. **Default Arguments:** Parameters that assume a default value if no argument is provided in the call.
4. **Variable Length Arguments:**
   - `*args`: Accepts multiple positional arguments as a tuple.
   - `**kwargs`: Accepts multiple keyword arguments as a dictionary.

Worked Example:

Using Different Argument Types Demonstrate the use of default arguments and variable length arguments.
Step 1: Function with a default argument

```
def greet(name, message="Welcome to Python"):
    print(f"Hello {name} {message}")

greet("Alice")
greet("Bob", "Good Morning!")
```

Output:

```
Hello Alice Welcome to Python
Hello Bob Good Morning!
```

Step 2: Function with variable length positional arguments

```
def sum_all(*numbers):
    total = 0
    for num in numbers:
        total += num
    return total

print("Sum:", sum_all(10, 20, 30, 40))
```

Output:

```
Sum: 100
```

3.3 Variable Scope

Methodology:

Local and Global Scope The scope of a variable determines where it can be accessed in the code.

1. **Local Scope:** Variables declared inside a function. They cannot be accessed outside the function.
2. **Global Scope:** Variables declared outside all functions. They can be accessed everywhere.
3. **Global Keyword:** Used inside a function to modify a global variable instead of creating a new local variable.

Worked Example:

Modifying a Global Variable Show how a local variable shadows a global variable and how to use the `global` keyword.

Step 1: Local vs Global Variables

```
count = 10  # Global variable

def show_scope():
    count = 5  # Local variable
    print("Inside function count:", count)

show_scope()
print("Outside function count:", count)
```

Output:

```
Inside function count: 5
Outside function count: 10
```

Step 2: Using the global keyword

```
total = 100

def modify_global():
    global total
    total += 50
    print("Modified total inside:", total)

modify_global()
print("Total outside:", total)
```

Output:

```
Modified total inside: 150
Total outside: 150
```

3.4 Lambda Functions

Methodology:

Anonymous Functions A lambda function is a small anonymous function defined without the `def` keyword.

1. **Syntax:** `lambda arguments:  expression`
2. Can take any number of arguments but has only one evaluated expression.
3. Often used with built-in functions like `map()` and `filter()`.

Worked Example:

Using Lambda with Filter and Map Use lambda functions to filter even numbers from a list and square them.

Step 1: Filter even numbers

```
numbers = [1, 2, 3, 4, 5, 6]
evens = list(filter(lambda x: x % 2 == 0, numbers))
print("Even numbers:", evens)
```

Step 2: Map to compute squares

```
squares = list(map(lambda x: x ** 2, evens))
print("Squares of evens:", squares)
```

Output:

```
Even numbers: [2, 4, 6]
Squares of evens: [4, 16, 36]
```

3.5 Built-in Functions and Modules

Methodology:

Common Built-in Functions and Modules Python provides functions and modules for common tasks:

1. **Data Structure Built-ins:** `len()` returns length. `max()` and `min()` return extremum values. `sum()` calculates totals.
2. **Math Module:** Contains functions like `math.sqrt()` for square root and constants like `math.pi`.
3. **Datetime Module:** Used for manipulating dates and times such as fetching the current timestamp via `datetime.datetime.now()`.

Worked Example:

Using Math and Datetime Modules Write a program to calculate the area of a circle and print the current date.

Step 1: Import modules and perform calculations

```
import math
import datetime

# Area of circle: pi * r^2
radius = 7
area = math.pi * math.pow(radius, 2)
print(f"Area of circle with radius {radius}: {area:.2f}")

# Get current date and time
current_time = datetime.datetime.now()
print("Current Date and Time:", current_time.strftime("%Y-%m-%d"))
```

Output:

```
Area of circle with radius 7: 153.94
Current Date and Time: 2026-06-04
```

3.6 Creating and Using Custom Modules

Methodology:

Module Creation and Import A module is simply a file containing Python statements and definitions.

1. **Create a Module:** Save code in a file with a `.py` extension (example `calc.py`).
2. **Import a Module:** Use the `import` statement to load it.

3. **Accessing Attributes:** Use dot notation (`module_name.function_name()`).
4. **Selective Import:** Use `from module_name import function_name` to import specific parts directly into the namespace.

Worked Example:

Creating and Importing a Module Create a simple calculator module and use its functions in another script.

Step 1: Create the module (`calc.py`)

```
# calc.py
def add(a, b):
    return a + b

def multiply(a, b):
    return a * b
```

Step 2: Import and use the module in main script

```
# main.py
import calc
from calc import multiply

sum_result = calc.add(10, 5)
prod_result = multiply(10, 5)

print("Addition Result:", sum_result)
print("Multiplication Result:", prod_result)
```

Output:

```
Addition Result: 15
Multiplication Result: 50
```

CHAPTER 4

Object Oriented Programming

4.1 OOP Concepts

Object-Oriented Programming (OOP) is a paradigm based on the concept of "objects" which contain data (attributes) and code (methods). The primary concepts of OOP include Classes, Objects, Encapsulation, Inheritance, Polymorphism, and Abstraction. In Python, everything is an object, and OOP helps to organize complex programs into manageable and reusable pieces.

4.2 Class and Objects

A class is a blueprint for creating objects. An object is a specific instance of a class.

Methodology:

Defining a Class and Instantiating Objects

1. Use the `class` keyword followed by the class name to define a class.
2. Define variables inside the class to act as attributes.
3. Define functions inside the class to act as methods. The first parameter of an instance method is always `self`, which refers to the instance calling the method.
4. Create an object by calling the class name followed by parentheses: `obj = ClassName()`.
5. Access attributes and methods using the dot operator: `obj.attribute` or `obj.method()`.

Worked Example:

Creating a Simple Class and Object Create a class `Rectangle` with attributes `length` and `width`, and a method to calculate the area. Instantiate an object and calculate the area.
Solution:

```
class Rectangle:
    # Attributes
    length = 5
    width = 3

    # Method
    def calculate_area(self):
        return self.length * self.width

# Creating an object
rect1 = Rectangle()

# Accessing method
```

```
area = rect1.calculate_area()
print(f"Area of rectangle: {area}")
```

Output:

```
Area of rectangle: 15
```

4.3 Constructors

A constructor is a special method used to initialize objects when they are created. In Python, the constructor method is named `__init__`.

Methodology:

Using the Constructor Initialization Method

1. Define the `__init__(self, arg1, arg2, ...)` method inside the class.
2. Inside `__init__`, assign the arguments to instance attributes using `self.attribute_name = arg`.
3. When instantiating the object pass the required arguments: `obj = ClassName(val1, val2)`.

Worked Example:

Constructor for Complex Numbers Addition Design a class `Complex` that uses a constructor to initialize the real and imaginary parts of a complex number and provides a method to add two complex numbers and display the result.

Solution:

```
class Complex:
    # Constructor
    def __init__(self, real, imag):
        self.real = real
        self.imag = imag

    # Method to add two complex numbers
    def add(self, other_complex):
        res_real = self.real + other_complex.real
        res_imag = self.imag + other_complex.imag
        return Complex(res_real, res_imag)

    def display(self):
        if self.imag >= 0:
            print(f"{self.real} + {self.imag}i")
        else:
            print(f"{self.real} - {abs(self.imag)}i")

# Creating objects
c1 = Complex(3, 4)
c2 = Complex(5, -2)

print("Complex Number 1:")
c1.display()

print("Complex Number 2:")
c2.display()
```

```
# Adding the two complex numbers
c3 = c1.add(c2)

print("Sum:")
c3.display()
```

Output:

```
Complex Number 1:
3 + 4i
Complex Number 2:
5 - 2i
Sum:
8 + 2i
```

4.4 Types of Methods

Python classes can contain three main types of methods: Instance Methods, Class Methods, and Static Methods.

Methodology:

Instance Methods Class Methods and Static Methods

1. **Instance Methods:** Operate on an instance of the class. They take `self` as the first parameter. Used for accessing or modifying object state.
2. **Class Methods:** Operate on the class itself rather than instances. They take `cls` as the first parameter and are decorated with `@classmethod`. Used for factory methods or modifying class state.
3. **Static Methods:** Do not operate on instances or classes directly. They take neither `self` nor `cls` as a first parameter and are decorated with `@staticmethod`. Used for utility functions related to the class.

Worked Example:

Demonstrating Different Method Types Create a class `Student` that demonstrates the use of an instance method, a class method, and a static method.
Solution:

```
class Student:
    # Class attribute
    school_name = "GTU Diploma College"

    def __init__(self, name, age):
        self.name = name  # Instance attribute
        self.age = age

    # Instance Method
    def show_info(self):
        print(f"Student Name: {self.name}, Age: {self.age}")

    # Class Method
    @classmethod
    def change_school(cls, new_school):
        cls.school_name = new_school
        print(f"School changed to {cls.school_name}")
```

```
    # Static Method
    @staticmethod
    def is_adult(age):
        return age >= 18

# 1. Using Instance Method
s1 = Student("Rahul", 17)
s1.show_info()

# 2. Using Class Method
Student.change_school("New GTU Institute")
print(f"s1's school: {s1.school_name}")

# 3. Using Static Method
print(f"Is 17 an adult? {Student.is_adult(17)}")
print(f"Is 21 an adult? {Student.is_adult(21)}")
```

Output:

```
Student Name: Rahul, Age: 17
School changed to New GTU Institute
s1's school: New GTU Institute
Is 17 an adult? False
Is 21 an adult? True
```

4.5 Data Encapsulation

Data Encapsulation is the mechanism of hiding data implementation by restricting access to public methods. In Python, encapsulation is achieved by prefixing attribute names with a double underscore (__), making them "private".

Methodology:

Implementing Data Encapsulation

1. Prefix the attribute name with double underscores inside the constructor (e.g., `self.__balance = 0`).
2. Create public **getter** methods to retrieve the value of the private attribute.
3. Create public **setter** methods to update the value of the private attribute safely.

Worked Example:

Encapsulation with a Bank Account Create a `BankAccount` class to demonstrate encapsulation. The account balance should be private, and access should only be allowed through `deposit` and `get_balance` methods.
Solution:

```
class BankAccount:
    def __init__(self, owner, balance):
        self.owner = owner
        self.__balance = balance  # Private attribute

    # Public getter method
    def get_balance(self):
        return self.__balance

    # Public setter method
```

```
    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
            print(f"Deposited {amount}. New balance is {self.__balance}")
        else:
            print("Invalid deposit amount")

acc = BankAccount("Milav", 1000)
print(f"Account Owner: {acc.owner}")
print(f"Initial Balance: {acc.get_balance()}")

acc.deposit(500)

# Trying to access the private variable directly will result in an AttributeError
# print(acc.__balance)  # This line would cause an error
```

Output:

```
Account Owner: Milav
Initial Balance: 1000
Deposited 500. New balance is 1500
```

4.6 Inheritance

Inheritance allows one class (child class) to derive properties and methods from another class (parent class). Python supports various types of inheritance.

Methodology:

Implementing Single and Multiple Inheritance

1. **Single Inheritance:** A child class inherits from a single parent class. Syntax: `class ChildClass(ParentClass):`
2. **Multiple Inheritance:** A child class inherits from multiple parent classes. Syntax: `class ChildClass(Parent1, Parent2):`
3. **Multi-level Inheritance:** A child class inherits from another child class. `A -> B -> C`.
4. **Hierarchical Inheritance:** Multiple child classes inherit from a single parent class. `A -> B, A -> C`.
5. **Hybrid Inheritance:** A combination of two or more types of inheritance.

Worked Example:

Single Level Inheritance Design a class for single level inheritance using a base class `Person` and derived class `Employee`.

Solution:

```
class Person:
    def __init__(self, name):
        self.name = name

    def display_person(self):
        print(f"Name: {self.name}")
```

```
class Employee(Person): # Single Inheritance
    def __init__(self, name, emp_id):
        super().__init__(name) # Call parent constructor
        self.emp_id = emp_id

    def display_employee(self):
        self.display_person()
        print(f"Employee ID: {self.emp_id}")

emp = Employee("Amit", "E101")
emp.display_employee()
```

Output:

```
Name: Amit
Employee ID: E101
```

Worked Example:

Multiple and Multi-level Inheritance Implement multiple and multi-level inheritance using appropriate classes.

Solution:

```
# --- Multi-level Inheritance ---
class Animal:
    def eat(self):
        print("Animal is eating...")

class Dog(Animal):
    def bark(self):
        print("Dog is barking...")

class Puppy(Dog): # Inherits from Dog, which inherits from Animal
    def weep(self):
        print("Puppy is weeping...")

print("--- Multi-level Inheritance ---")
p = Puppy()
p.eat()
p.bark()
p.weep()

# --- Multiple Inheritance ---
class Father:
    def father_property(self):
        print("Father's property")

class Mother:
    def mother_property(self):
        print("Mother's property")

class Child(Father, Mother): # Inherits from both Father and Mother
    def child_property(self):
        print("Child's own property")

print("\n--- Multiple Inheritance ---")
```

```
c = Child()
c.father_property()
c.mother_property()
c.child_property()
```

Output:

```
--- Multi-level Inheritance ---
Animal is eating...
Dog is barking...
Puppy is weeping...

--- Multiple Inheritance ---
Father's property
Mother's property
Child's own property
```

Worked Example:

Hierarchical Inheritance Implement hierarchical inheritance where a single base class `Shape` is inherited by multiple derived classes `Circle` and `Square`.

Solution:

```
# Base Class
class Shape:
    def __init__(self, color):
        self.color = color

    def display_color(self):
        print(f"The color of the shape is {self.color}")

# Derived Class 1
class Circle(Shape):
    def draw_circle(self):
        print("Drawing a Circle.")

# Derived Class 2
class Square(Shape):
    def draw_square(self):
        print("Drawing a Square.")

# Hierarchical Inheritance Usage
c = Circle("Red")
c.draw_circle()
c.display_color()

s = Square("Blue")
s.draw_square()
s.display_color()
```

Output:

```
Drawing a Circle.
The color of the shape is Red
Drawing a Square.
The color of the shape is Blue
```

4.7 Polymorphism through Inheritance

Polymorphism means "many forms". In the context of inheritance, polymorphism allows a child class to provide a specific implementation of a method that is already provided by its parent class. This is called **Method Overriding**.

Methodology:

Implementing Method Overriding

1. Define a method in the base class.
2. Define a method with the exact same name and parameters in the derived class.
3. When you call the method on an object of the derived class, the derived class's method will execute instead of the base class's method.

Worked Example:

Method Overriding for Polymorphism Write a Python program to demonstrate method overriding using inheritance with a `Shape` class.
Solution:

```
class Shape:
    def area(self):
        return "Area calculation not defined for base Shape class"

class Square(Shape):
    def __init__(self, side):
        self.side = side

    # Overriding the area method
    def area(self):
        return self.side * self.side

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    # Overriding the area method
    def area(self):
        return 3.14 * self.radius * self.radius

# Creating instances
shape_obj = Shape()
square_obj = Square(4)
circle_obj = Circle(5)

# Demonstrating polymorphism
print("Shape area:", shape_obj.area())
print("Square area:", square_obj.area())
print("Circle area:", circle_obj.area())
```

Output:

```
Shape area: Area calculation not defined for base Shape class
Square area: 16
Circle area: 78.5
```

4.8 Abstraction

Abstraction is the process of hiding the real implementation of an application from the user and emphasizing only on usage of it. In Python, abstraction is achieved using Abstract Base Classes (ABC). An abstract class cannot be instantiated and often contains abstract methods that must be implemented by its subclasses.

Methodology:

Abstract Classes and Methods

1. Import `ABC` and `abstractmethod` from the `abc` module.
2. Inherit your base class from `ABC`.
3. Decorate the methods that must be implemented by subclasses with `@abstractmethod`.
4. Create a derived class and provide the actual implementation for the abstract methods.

Worked Example:

Implementing Abstraction Create an abstract class `Vehicle` with an abstract method `start_engine`. Create a derived class `Car` that implements this method.

Solution:

```
from abc import ABC, abstractmethod

# Abstract base class
class Vehicle(ABC):
    @abstractmethod
    def start_engine(self):
        pass

    def show_vehicle_type(self):
        print("This is a generic vehicle.")

# Derived class
class Car(Vehicle):
    # Implementing the abstract method
    def start_engine(self):
        print("Car engine starting with a key...")

# You cannot instantiate an abstract class:
# v = Vehicle() # This would raise an Error

c = Car()
c.show_vehicle_type() # Inherited normal method
c.start_engine()      # Implemented abstract method
```

Output:

```
This is a generic vehicle.
Car engine starting with a key...
```

Formulae

Appendix: Key Formulae

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

Temperature Conversion

Celsius to Fahrenheit

This formula is used to convert a given temperature in degrees Celsius (C) into degrees Fahrenheit (F). The conversion factor involves multiplying the Celsius temperature by the ratio of the scales ($\frac{9}{5}$) and then shifting the zero point by adding 32.

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