App (4321602) - Problem Solver

Antigravity AI

June 4, 2026

App

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

Basic of Python Data Structure - Dictionary Tuple and Set

1.1 Introduction to Strings and Lists

Methodology:

Strings and Lists Fundamentals **Strings** are immutable sequences of characters enclosed in single or double quotes.

1. **Creation:** `s = 'hello'` or `s = "hello"`
2. **Access:** Indexing `s[0]` and slicing `s[1:4]`

Lists are mutable ordered sequences of items, enclosed in square brackets.

1. **Creation:** `lst = [1, 2, 'apple']`
2. **Access:** Indexing `lst[0]` and slicing `lst[1:3]`
3. **Modification:** `lst.append(item)`, `lst.remove(item)`, `lst[0] = new_val`

Worked Example:

String and List Operations Given a string `text = "Python"` and a list `nums = [10, 20, 30]`, extract the first 3 characters of the string, append 40 to the list, and print both.
Solution:

1. Extract the first 3 characters of the string: `sub_text = text[0:3]` $\Rightarrow$ `"Pyt"`
2. Append 40 to the list: `nums.append(40)` $\Rightarrow$ `[10, 20, 30, 40]`
3. Print the results: `sub_text` is `"Pyt"`, and `nums` is `[10, 20, 30, 40]`.

1.2 Sets in Python

Methodology:

Creating and Accessing Sets A **Set** is an unordered, unindexed, and mutable collection of unique elements. Sets are written with curly brackets.

1. **Create a Set:** Use curly braces `{}` or the `set()` function. Note: `{}` creates an empty dictionary. Use `set()` for an empty set.
2. **Accessing Sets:** Since sets are unordered, elements cannot be accessed by index. Use a `for` loop to iterate, or the `in` keyword to check membership.

Worked Example:

Creating and Accessing a Set Create a set containing the elements 1, 2, and 3. Check if 2 is in the set, and iterate over its elements.
Solution:

1. **Create:** `my_set = {1, 2, 3}`
2. **Check membership:** `2 in my_set` evaluates to `True`.
3. **Iterate:**

```
for val in my_set:
    print(val)
```

Output: `1`, `2`, `3` (order may vary since sets are unordered).

Methodology:

Updating and Deleting from Sets Sets can be modified using built-in methods.

1. **Update and Add Elements:**
   - `set.add(item)`: Adds a single item.
   - `set.update(iterable)`: Adds multiple items from another set or iterable.
2. **Delete from Sets:**
   - `set.remove(item)`: Removes the item. Raises `KeyError` if not found.
   - `set.discard(item)`: Removes the item. Does not raise an error if not found.
   - `set.pop()`: Removes and returns a random element.
   - `set.clear()`: Empties the set entirely.

Worked Example:

Modifying a Set Given an initial set `S = {10, 20, 30}`. Add `40`, update it with multiple items `{50, 60}`, and remove `20` using `discard()`.
Solution:

1. Add 40: `S.add(40)` $\Rightarrow$ `{10, 20, 30, 40}`
2. Update with `{50, 60}`: `S.update({50, 60})` $\Rightarrow$ `{10, 20, 30, 40, 50, 60}`
3. Remove 20: `S.discard(20)` $\Rightarrow$ `{10, 30, 40, 50, 60}`

Methodology:

Python Set Operations Mathematical set operations can be performed using methods or operators.

1. **Union (`|` or `union()`):** Combines elements from both sets, ignoring duplicates.
2. **Intersection (`&` or `intersection()`):** Returns common elements present in both sets.
3. **Difference (`-` or `difference()`):** Returns elements present in the first set but not in the second.
4. **Symmetric Difference (`^` or `symmetric_difference()`):** Returns elements present in either set but not in both.

Worked Example:

Evaluating Set Operations Given `A = {1, 2, 3, 4}` and `B = {3, 4, 5, 6}`. Compute Union, Intersection, Difference (`A - B`), and Symmetric Difference.
Solution:

1. **Union:** `A | B` = `{1, 2, 3, 4, 5, 6}`
2. **Intersection:** `A & B` = `{3, 4}`
3. **Difference (A - B):** `A - B` = `{1, 2}`
4. **Symmetric Difference:** `A ^ B` = `{1, 2, 5, 6}`

1.3 Tuples in Python

Methodology:

Tuples and Operations A **Tuple** is an ordered, immutable sequence of elements.

1. **Creating Tuples:** Enclosed in parentheses `()` or created without parentheses (packing). E.g., `t = (1, 2, 3)` or `t = 1, 2, 3`. A single element tuple requires a trailing comma: `t_single = (1,)`.
2. **Accessing Tuple:** Use zero-based indexing. E.g., `t[0]` accesses the first element. Negative indexing is supported: `t[-1]` is the last element.
3. **Slicing Tuples:** Extracting a sub-tuple using the syntax `t[start:end:step]`. E.g., `t[1:3]`.
4. **Tuples are Immutable:** Elements cannot be changed, added, or removed after creation. To modify, one must convert to a list, modify the list, and convert back to a tuple.

Worked Example:

Tuple Creation and Indexing Create a tuple of numbers from 1 to 5. Access the second element, the last element, and extract a slice from index 1 to 3. Attempt to modify the first element.
Solution:

1. **Create:** `T = (1, 2, 3, 4, 5)`
2. **Access second element:** `T[1]` = `2`
3. **Access last element:** `T[-1]` = `5`
4. **Slicing index 1 to 3:** `T[1:4]` = `(2, 3, 4)` (Note: the end index is exclusive).
5. **Modify element:** `T[0] = 10` will raise a `TypeError` because tuples are immutable.

Methodology:

Iteration and Built in Tuple Functions Tuples support various built-in features for easy traversal and calculations.

1. **Iterate over tuples:** Use a `for` loop. `for item in t: print(item)`.
2. **Python Tuple Operations:** Concatenation (`+`) joins tuples together. Repetition (`*`) repeats the elements of a tuple. Membership check (`in`).
3. **Built-In Tuple Functions and methods:**
   - `len(t)`: Returns the number of items.
   - `max(t)` and `min(t)`: Returns the maximum and minimum items.
   - `sum(t)`: Returns the sum of all numeric items.
   - `t.count(item)`: Returns the number of occurrences of an item.
   - `t.index(item)`: Returns the index of the first occurrence of an item.

Worked Example:

Tuple Functions and Methods Given `T1 = (10, 20, 30)` and `T2 = (20, 20, 40)`. Concatenate them, find the count of `20`, and calculate the total sum.
Solution:

1. **Concatenation:** `T = T1 + T2` $\Rightarrow$ `(10, 20, 30, 20, 20, 40)`
2. **Count of 20:** `T.count(20)` $\Rightarrow$ `3`
3. **Sum:** `sum(T)` $= 10 + 20 + 30 + 20 + 20 + 40 = 140$

1.4 Dictionaries in Python

Methodology:

Creating and Accessing Dictionaries A **Dictionary** is an ordered, mutable, and indexed collection of key-value pairs.

1. **Creating Dictionaries:** Use curly braces with key-value pairs separated by colons `{key: value}` or use the `dict()` constructor.
2. **Properties of Dictionary Keys:** Keys must be immutable types (e.g., strings, numbers, tuples) and unique. If a duplicate key is provided, the last value overwrites previous ones.
3. **Accessing Items in Python Dictionaries:**
   - By key: `dict[key]`. Raises a `KeyError` if the key is missing.
   - Using `get()`: `dict.get(key, default_value)`. Provides safe access by returning a default value if the key does not exist.

Worked Example:

Creating and Accessing a Dictionary Create a dictionary representing a student with keys "name" (value "Alice") and "age" (value 20). Access the "name" using brackets and "grade" using `get()` with a default of "A".
Solution:

1. **Create:** `student = {"name": "Alice", "age": 20}`

2. **Access "name":** `student["name"]` ⇒ `"Alice"`
3. **Access "grade":** `student.get("grade", "A")` ⇒ `"A"` (since the key "grade" does not exist in the dictionary, it returns the default).

Methodology:

Modifying Dictionaries Dictionaries are mutable, allowing additions, updates, and removals of key-value pairs.

1. **Add and Update:**
   - Assignment: `dict[key] = value`. If the key exists, it updates the value; if not, it adds a new pair.
   - `update()`: `dict.update({"key": value})`. Updates the dictionary with elements from another dictionary or iterable of pairs.
2. **Remove in Dictionaries:**
   - `pop(key)`: Removes the item with the specified key and returns its value.
   - `popitem()`: Removes the last inserted key-value pair.
   - `del dict[key]`: Removes the item with the specified key.
   - `clear()`: Empties the entire dictionary.

Worked Example:

Modifying a Dictionary Given `D = {"a": 1, "b": 2}`. Update the value of "a" to 10, add "c" with the value 3, and remove "b" using `pop()`.
Solution:

1. **Update "a":** `D["a"] = 10` ⇒ `{"a": 10, "b": 2}`
2. **Add "c":** `D["c"] = 3` ⇒ `{"a": 10, "b": 2, "c": 3}`
3. **Remove "b":** `D.pop("b")` returns 2. The dictionary becomes ⇒ `{"a": 10, "c": 3}`

Methodology:

Built In Dictionary Methods and Functions Python provides several built-in functions and methods to work efficiently with dictionaries.

1. **keys():** Returns a view object of all the keys in the dictionary.
2. **values():** Returns a view object of all the values in the dictionary.
3. **items():** Returns a view object containing tuples of `(key, value)` pairs.
4. **len(dict):** Returns the total number of key-value pairs in the dictionary.
5. **Iteration:** Using a `for` loop directly on the dictionary iterates over its keys. To iterate over both keys and values, use `for k, v in dict.items():`.

Worked Example:

Iterating and Using Dictionary Methods Given `grades = {"Math": 85, "Science": 90}`, extract all keys, values, and print each key-value pair using a loop.
Solution:

1. **Keys:** `grades.keys()` ⇒ `dict_keys(['Math', 'Science'])`
2. **Values:** `grades.values()` ⇒ `dict_values([85, 90])`
3. **Iteration over items:**

```
for subject, score in grades.items():
    print(subject, score)
```

Output:
```
Math 85
Science 90
```

CHAPTER 2

Modules and Packages

2.1 Python Modules

2.1.1 Introduction to module

A Python module is simply a file containing Python statements and definitions. Breaking a large program down into smaller and more manageable files makes the code easier to maintain, debug, and reuse.

2.1.2 Creating user defined module

Methodology:

Creating a User Defined Module

1. Open a text editor or IDE.
2. Define Python functions variables or classes.
3. Save the file with a `.py` extension. The filename (excluding the extension) becomes the module name.

Worked Example:

Develop a module for arithmetic operations **Problem:** Create a module named `calc.py` that contains functions for addition and subtraction.
Step 1: Write the Python code Create a new file and add the following function definitions:

```
# calc.py
def add(x, y):
    return x + y

def subtract(x, y):
    return x - y

pi_value = 3.14159
```

Step 2: Save the file Save this file as `calc.py` in your current working directory. It is now ready to be imported.

2.1.3 Importing a module in python

Methodology:

Methods of Importing a Module

1. **Normal import:** Use `import modulename`. You must prefix the module name when calling its functions (e.g. `modulename.function()`).
2. **From import:** Use `from modulename import specific_function`. This allows calling the function directly without the module prefix.
3. **From import with *:** Use `from modulename import *`. This imports all public objects from the module into the current namespace.

Worked Example:

Use the calc module using different import methods **Problem:** Write three separate scripts demonstrating the three import techniques to use the `add` function from `calc.py`.
Solution 1: Normal import

```
import calc
result = calc.add(10, 5)
print("Normal import result:", result)
```

Solution 2: From import

```
from calc import add
result = add(10, 5)
print("From import result:", result)
```

Solution 3: From import with *

```
from calc import *
result = add(10, 5)
print("Star import result:", result)
```

2.1.4 Module search path

Methodology:

Managing Module Search Path When a module is imported Python searches for it in directories listed in `sys.path`. To manipulate this:

1. Import the `sys` built-in module.
2. View the current search paths using `print(sys.path)`.
3. Append a new directory to the search path using `sys.path.append('directory_path')`.

Worked Example:

Add a custom directory to the Python search path **Problem:** Add the directory `/home/user/my_modules` to the Python search path so that modules inside it can be imported.
Solution:

```
import sys

# Print original path
```

```
print("Original Path:", sys.path)

# Append custom directory
sys.path.append('/home/user/my_modules')

# Verify the new path is added
print("Updated Path:", sys.path[-1])
```

2.2 Packages in Python

2.2.1 Introduction to Packages

A package is a hierarchical file directory structure that defines a single Python application environment consisting of modules and subpackages. Packages allow for a dotted module namespace to prevent naming collisions.

2.2.2 Creating user defined package

Methodology:

Structuring a Python Package

1. Create a main directory representing the package name.
2. Inside the directory create an `__init__.py` file (can be empty). This tells Python to treat the directory as a package.
3. Place module files (`.py`) or subdirectories (subpackages) inside this main directory.

Worked Example:

Create an e commerce package structure **Problem:** Create a package named `ecommerce` containing two modules: `sales.py` and `inventory.py`.
Solution: Organize your files in the operating system exactly like this:

```
ecommerce/
    __init__.py
    sales.py
    inventory.py
```

Content of `ecommerce/sales.py`:

```
def calculate_tax(amount):
    return amount * 0.18
```

2.2.3 Importing a package in python

Methodology:

Methods of Importing from a Package

1. **Normal import:** `import packagename.modulename`. Use full dotted path: `packagename.modulename.func()`.
2. **From import:** `from packagename import modulename`. Use module prefix: `modulename.func()`.
3. **From import with *:** `from packagename.modulename import *`. Call function directly: `func()`.

Worked Example:

Importing the ecommerce package **Problem:** Call the `calculate_tax` function from the `ecommerce` package.
Solution 1: Normal import

```
import ecommerce.sales
tax = ecommerce.sales.calculate_tax(100)
print(tax)
```

Solution 2: From import

```
from ecommerce import sales
tax = sales.calculate_tax(100)
print(tax)
```

2.2.4 Intra-package References

Methodology:

Using Relative Imports within Packages Modules inside the same package can import each other using relative imports:

1. Use a single dot (`.`) to refer to the current package.
2. Use two dots (`..`) to refer to the parent package.
3. Syntax: `from .module_name import function_name`.

Worked Example:

Implement a relative import between sibling modules **Problem:** Inside the `ecommerce` package make `inventory.py` use a function from `sales.py`.
Solution: In `ecommerce/inventory.py`:

```
# Using relative import to get calculate_tax from sales module
from .sales import calculate_tax

def add_item_with_tax(item_price):
    final_price = item_price + calculate_tax(item_price)
    print("Item added. Final price:", final_price)
```

2.3 Package Management using PIP

2.3.1 Installing PIP

Methodology:

Verifying and Installing PIP PIP is the standard package manager for Python.

1. Check if PIP is already installed by running `pip -version` or `python -m pip -version` in the terminal.
2. If not installed download `get-pip.py` using curl or wget.
3. Execute the downloaded script via terminal using `python get-pip.py`.

Worked Example:

Install pip via command line **Problem:** Write the terminal commands required to install pip on a system that does not have it.
Solution:

```
# Download the installation script
curl https://bootstrap.pypa.io/get-pip.py -o get-pip.py

# Run the script to install pip
python get-pip.py

# Verify installation
pip --version
```

2.3.2 Installing/uninstalling python packages

Methodology:

Managing Python Packages with PIP

1. **Install a package:** Open terminal and execute `pip install package_name`.
2. **Install specific version:** Execute `pip install package_name==version_number`.
3. **Uninstall a package:** Execute `pip uninstall package_name`.
4. **List installed packages:** Execute `pip list`.

Worked Example:

Install and uninstall the requests package **Problem:** Use pip to install the `requests` library verify it and then uninstall it.
Solution: Execute the following commands in the terminal:

```
# Step 1: Install the requests package
pip install requests

# Step 2: Verify the installation by listing packages
pip list

# Step 3: Uninstall the requests package (using -y to auto-confirm)
pip uninstall requests -y
```

CHAPTER 3

Exception Handling

3.1 Introduction to Exception

An exception is an error that occurs during the execution of a program which disrupts the normal flow of instructions. Exception handling provides a computational mechanism to catch and manage these errors systematically, preventing the program from crashing abruptly and allowing alternative logic to execute.

3.2 Types of Exceptions

Exceptions can be broadly categorized into built-in exceptions provided by the programming language and user-defined exceptions created by the programmer.

3.2.1 Built-in Exceptions

Built-in exceptions are predefined error types raised automatically by the Python interpreter when specific computational errors occur.

Methodology:

Handling Built in Exceptions When performing computational tasks, identify which built-in exception might be triggered:

1. **`ZeroDivisionError`:** Raised when the denominator in a division or modulo operation is zero.
2. **`IndexError`:** Raised when attempting to access a sequence (like a list) using an index that is out of bounds.
3. **`TypeError`:** Raised when an operation is applied to an object of an inappropriate type (e.g. adding a string to an integer).
4. **`ValueError`:** Raised when a built-in operation or function receives an argument that has the right type but an inappropriate value.

Worked Example:

Triggering Built in Exceptions Demonstrate operations that naturally raise `ZeroDivisionError` and `TypeError`.

Step 1: Triggering `ZeroDivisionError`

```
# Attempting to divide an integer by zero
result = 100 / 0
# Output: ZeroDivisionError: division by zero
```

Step 2: Triggering `TypeError`

```
# Attempting to add an integer to a string
```

```
sum_val = 50 + " apples"
# Output: TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

3.2.2 User Defined Exceptions

Methodology:

Creating User Defined Exceptions To enforce specific domain logic computationally, you can create custom exceptions:

1. Define a new class that inherits from the base `Exception` class.
2. Define the `__init__` method to accept custom error properties or messages.
3. Instantiate and raise this custom class whenever the domain constraint is violated.

Worked Example:

Custom Exception for Invalid Balance Create a user-defined exception `InsufficientFundsError` to handle overdrawn bank withdrawals computationally.
Step 1: Define the custom exception class.

```
class InsufficientFundsError(Exception):
    def __init__(self, deficit):
        self.deficit = deficit
        self.message = f"Withdrawal denied. Short by ${deficit}."
        super().__init__(self.message)
```

Step 2: Trigger the custom exception in a function.

```
def process_withdrawal(balance, amount):
    if amount > balance:
        raise InsufficientFundsError(amount - balance)
    return balance - amount
```

3.3 Raising Exceptions

Methodology:

Raising Exceptions Manually The `raise` statement allows the programmer to manually trigger an exception under specific computational conditions.

1. Evaluate the input or state using conditional statements (e.g. `if` condition).
2. If the condition represents an error state, use the `raise` keyword followed by the exception class.
3. Optionally, pass a descriptive error message to the exception class constructor.

Worked Example:

Raising a ValueError for Invalid Data Write a function to compute the square root of a number and manually raise an exception if the input is negative.
Step 1: Implement validation using `raise`.

```
import math

def compute_square_root(num):
```

```
    if num < 0:
        # Manually raise an exception for invalid input
        raise ValueError("Cannot compute square root of a negative number.")
    return math.sqrt(num)
```

Step 2: Test the function with negative data.

```
compute_square_root(-16)
# Output: ValueError: Cannot compute square root of a negative number.
```

3.4 Handling Exceptions

To prevent a program from crashing when an exception is raised, exceptions must be computationally handled using the `try`, `except`, and `finally` clauses.

3.4.1 Try clause

The `try` block is used to enclose the code that might potentially generate an exception. The runtime environment monitors this block for errors.

3.4.2 Except clause

The `except` block specifies the handler code that executes if a specific exception is raised in the `try` block.

3.4.3 Finally clause

The `finally` block contains cleanup code that must execute strictly regardless of whether an exception occurred or not.

Methodology:

Implementing Exception Handling Blocks To systematically handle errors:

1. Place the risky computational code inside the `try` block.
2. Immediately following the `try` block, write one or more `except` blocks targeting specific error types.
3. Inside the `except` block, write the fallback or logging logic.
4. Add a `finally` block at the end to release resources like closing files or connections.

Worked Example:

Handling Invalid Input and Division by Zero Write a script that reads an integer from the user and calculates its reciprocal while handling invalid type inputs and zero appropriately.

Step 1: Implement the blocks.

```
def calculate_reciprocal(user_input):
    try:
        # Risky operations
        number = int(user_input)
        reciprocal = 1.0 / number
        print(f"The reciprocal is {reciprocal}")

    except ValueError:
        # Handled if user_input cannot be converted to int
        print("Error: Input provided is not a valid integer.")
```

```
    except ZeroDivisionError:
        # Handled if number is 0
        print("Error: Cannot calculate the reciprocal of zero.")

    finally:
        # Always executes
        print("Execution of reciprocal calculation completed.")
```

Step 2: Execute with a valid integer.

```
calculate_reciprocal("4")
# Output:
# The reciprocal is 0.25
# Execution of reciprocal calculation completed.
```

Step 3: Execute with zero.

```
calculate_reciprocal("0")
# Output:
# Error: Cannot calculate the reciprocal of zero.
# Execution of reciprocal calculation completed.
```

Step 4: Execute with non-integer string.

```
calculate_reciprocal("text")
# Output:
# Error: Input provided is not a valid integer.
# Execution of reciprocal calculation completed.
```

CHAPTER 4

Files Handling

4.1 Introduction to Files and Types

Methodology:

File Operations Basics In Python, file operations typically follow a three-step sequence:

1. Open the file using the `open()` function, which returns a file object.
2. Perform read or write operations on the file object.
3. Close the file using the `close()` method to free system resources.

Files are primarily classified into two types:

1. **Text Files:** Store characters. Opened in modes like `'r'` for read, `'w'` for write, and `'a'` for append.
2. **Binary Files:** Store data in the exact same format as held in memory. Opened in modes with a `'b'` suffix, like `'rb'` or `'wb'`.

4.2 Opening and Closing Text File

Methodology:

Opening and Closing Files To safely open and close files:

1. **Standard Method:**
   - `f = open("filename.txt" mode="r")`
   - Operations (read/write)
   - `f.close()`
2. **Context Manager (Recommended):**
   - `with open("filename.txt" mode="r") as f:`
   - Operations (read/write) inside the block.
   - The file automatically closes when exiting the block.

Worked Example:

Opening and Closing a Text File **Problem:** Write a Python script to open a file named `data.txt` in write mode, check if it is closed, and then close it properly.
Solution:

1. Open the file:

```
file_obj = open("data.txt", "w")
```

2. Print the closed status:

```
print("Is file closed?", file_obj.closed)
```

3. Close the file and print status again:

```
file_obj.close()
print("Is file closed after close()?", file_obj.closed)
```

Output will indicate `False` and then `True`.

4.3 Reading and Writing Files

Methodology:

Methods for Reading and Writing **Writing Data:**

1. `f.write(string)`: Writes a single string to the file. Returns the number of characters written.
2. `f.writelines(list_of_strings)`: Writes a list of strings to the file. Does not append newlines automatically.

Reading Data:

1. `f.read(size)`: Reads `size` characters. If omitted or negative, reads the entire file.
2. `f.readline()`: Reads a single line from the file including the newline character.
3. `f.readlines()`: Reads all lines and returns them as a list of strings.

Worked Example:

Writing Multiple Lines to a Text File **Problem:** Create a text file `notes.txt` and write three lines of text using both `write()` and `writelines()`.
Solution:

1. Use context manager to open file in write mode (`'w'`):

```
with open("notes.txt", "w") as f:
    # Method 1: Using write()
    f.write("Line 1: Python is fun.\n")

    # Method 2: Using writelines()
    lines = ["Line 2: File handling is easy.\n", "Line 3: End of notes.\n"]
    f.writelines(lines)
```

Worked Example:

Reading from a Text File **Problem:** Read and display the contents of `notes.txt` line by line using a loop. Count the total number of characters.
Solution:

1. Initialize a character counter.
2. Open the file in read mode (`'r'`):

```
total_chars = 0
with open("notes.txt", "r") as f:
    for line in f:
        print(line, end="") # end="" avoids double newlines
        total_chars += len(line)

print(f"\nTotal characters: {total_chars}")
```

4.4 Setting Offsets in File

Methodology:

File Pointer and Offsets The position of the file pointer determines where the next read or write operation occurs.

1. **Find Current Position:** `f.tell()` returns the current integer position (in bytes) of the file pointer.
2. **Change Position:** `f.seek(offset whence)` moves the pointer.
   - `offset`: Number of bytes to move.
   - `whence`: Reference point for the move.
     - `0` (default): Absolute file positioning (beginning of file).
     - `1`: Relative to current position (only supported in binary mode or offset 0).
     - `2`: Relative to end of file (only supported in binary mode or offset 0).

Worked Example:

Using Seek and Tell **Problem:** Write "ABCDEFGHIJ" to a file. Read and print the first 3 characters. Then move the pointer back to the beginning and read the first 5 characters. Finally determine the pointer's position.
Solution:

1. Write data to file:

```
with open("letters.txt", "w") as f:
    f.write("ABCDEFGHIJ")
```

2. Open for reading and use seek/tell:

```
with open("letters.txt", "r") as f:
    part1 = f.read(3)
    print("First 3 characters:", part1)  # Prints: ABC

    # Move pointer back to start
    f.seek(0)

    part2 = f.read(5)
    print("Next 5 characters after seek:", part2)  # Prints: ABCDE

    # Get current position
    position = f.tell()
    print("Current pointer position:", position)  # Prints: 5
```

4.5 Object Serialization using Pickle Module

Methodology:

Object Serialization with Pickle Serialization is the process of converting a Python object (like a list or dictionary) into a byte stream to store it in a binary file. Deserialization is the reverse process.

1. **Import Module:** `import pickle`
2. **Serialize (Write):** Open file in `'wb'` (write binary) mode.
   - `pickle.dump(object file_pointer)`: Writes the object to the binary file.
3. **Deserialize (Read):** Open file in `'rb'` (read binary) mode.
   - `object = pickle.load(file_pointer)`: Reads and reconstructs the object from the binary file.

Worked Example:

Serializing and Deserializing a Dictionary **Problem:** Store a dictionary containing student details (name roll number marks) into a binary file named `student.dat` and then read the file to reconstruct and display the dictionary.
Solution:

1. Import pickle module and define the dictionary.

```
import pickle

student_data = {"Name": "Alice", "RollNo": 101, "Marks": 85.5}
```

2. Serialize the dictionary using `dump()`:

```
with open("student.dat", "wb") as f:
    pickle.dump(student_data, f)
print("Data serialized successfully.")
```

3. Deserialize the data using `load()`:

```
with open("student.dat", "rb") as f:
    loaded_data = pickle.load(f)

print("Deserialized Data:")
print("Name:", loaded_data["Name"])
print("Roll No:", loaded_data["RollNo"])
print("Marks:", loaded_data["Marks"])
```

CHAPTER 5
Graphics with Turtle

5.1 Introduction to Turtle Graphics

Methodology:

Initializing Turtle Graphics

1. **Import the module**: Start by importing the turtle module using `import turtle`.
2. **Create a screen**: Set up the drawing area using `screen = turtle.Screen()`.
3. **Create a turtle object**: Instantiate a turtle object using `t = turtle.Turtle()`.
4. **Keep window open**: End your program with `turtle.done()` to keep the graphic window open until manually closed.

Worked Example:

Draw a Simple Line **Problem:** Write a Python script to initialize a turtle and draw a straight line of 100 units.
Solution:

1. Import turtle module.
2. Create a screen and a turtle object.
3. Move the turtle forward by 100 units.
4. Keep the window open.

```
import turtle

# Setup screen and turtle
screen = turtle.Screen()
t = turtle.Turtle()

# Draw line
t.forward(100)

# Complete drawing
turtle.done()
```

5.2 Turtle and Screen Methods

5.2.1 Turtle Methods

Methodology:

Basic Movement and Drawing Methods

1. **forward(distance)** or **fd(distance)**: Move the turtle forward by the specified distance.
2. **backward(distance)** or **bk(distance)**: Move the turtle backward.
3. **right(angle)** or **rt(angle)**: Turn turtle right by the specified angle.
4. **left(angle)** or **lt(angle)**: Turn turtle left by the specified angle.
5. **goto(x y)** or **setpos(x y)**: Move turtle to an absolute position (x, y).
6. **penup()** or **up()**: Pull the pen up (no drawing when moving).
7. **pendown()** or **down()**: Pull the pen down (drawing when moving).
8. **circle(radius)**: Draw a circle with the given radius.
9. **color(colorstring)**: Change the pen color.
10. **pensize(width)**: Set the line thickness.

Worked Example:

Drawing a Square using Turtle Methods **Problem:** Use basic turtle methods to draw a square with a side length of 100 units.
Solution:

1. Move forward 100 units and turn right 90 degrees.
2. Repeat step 1 four times to complete the square.

```
import turtle
t = turtle.Turtle()

t.forward(100)
t.right(90)

t.forward(100)
t.right(90)

t.forward(100)
t.right(90)

t.forward(100)
t.right(90)

turtle.done()
```

5.2.2 TurtleScreen Methods

Methodology:

Configuring the Screen

1. **bgcolor(colorstring)**: Set the background color of the turtle window.
2. **title(titlestring)**: Set the title of the turtle graphics window.
3. **setup(width height)**: Set the size and position of the main window.
4. **clear()**: Delete the turtle's drawings from the screen.
5. **tracer(n delay)**: Turn turtle animation on/off and set delay for update drawings.

Worked Example:

Configuring Window and Background **Problem:** Create a turtle window titled "My Graphics" with a size of 600×400 and a light blue background. Draw a red circle of radius 50.
Solution:

1. Initialize screen object.
2. Apply screen methods for title setup and background color.
3. Create turtle object change color to red and draw a circle.

```
import turtle

screen = turtle.Screen()
screen.title("My Graphics")
screen.setup(width=600, height=400)
screen.bgcolor("lightblue")

t = turtle.Turtle()
t.color("red")
t.circle(50)

turtle.done()
```

5.3 Turtle Programming Loops and Conditional Statements

Methodology:

Using Loops for Repetitive Shapes

1. **For Loops**: Use `for` loops to execute turtle movements a fixed number of times. This is ideal for drawing regular polygons.
2. **Formula for Polygons**: The turning angle for a regular polygon with n sides is $360/n$.
3. **Nested Loops**: Use nested loops to draw repeating complex patterns like spirographs or grids.

Worked Example:

Drawing a Hexagon using a For Loop **Problem:** Draw a regular hexagon with side length 80 using a `for` loop.

Solution:

1. A hexagon has 6 sides. The turning angle is $360/6 = 60$ degrees.
2. Use a `for` loop iterating 6 times.
3. Inside the loop move forward by 80 units and turn left by 60 degrees.

```
import turtle
t = turtle.Turtle()

sides = 6
length = 80
angle = 360 / sides

for _ in range(sides):
    t.forward(length)
    t.left(angle)

turtle.done()
```

Methodology:

Using Conditionals to Control Graphics

1. **If Statements**: Use `if` statements inside loops to change colors shapes or movement direction based on conditions.
2. **Modulo Operator**: The modulo operator `%` is useful in loops to create alternating patterns based on the loop counter.

Worked Example:

Drawing an Alternating Color Star **Problem:** Draw a 5-pointed star where the line colors alternate between blue and red based on the stroke number.
Solution:

1. A 5-pointed star requires turning 144 degrees for each point.
2. Iterate 5 times.
3. Use an `if-else` condition with modulo arithmetic to check if the current iteration index is even or odd.
4. Assign the color "blue" for even and "red" for odd.

```
import turtle
t = turtle.Turtle()
t.pensize(3)

for i in range(5):
    if i % 2 == 0:
        t.color("blue")
    else:
        t.color("red")

    t.forward(150)
    t.right(144)
```

```
turtle.done()
```

Worked Example:

Drawing a Spirograph with Loops and Conditionals **Problem:** Draw a spirograph of 36 circles. If the circle index is divisible by 3 color it green otherwise color it purple.
Solution:

1. Loop 36 times (since 360/10 = 36 for a 10-degree offset each time).
2. Check `i % 3 == 0` to set the color to green or purple.
3. Draw a circle with radius 100.
4. Turn right by 10 degrees before drawing the next circle.

```
import turtle
screen = turtle.Screen()
screen.bgcolor("black")

t = turtle.Turtle()
t.speed(0) # Fastest drawing speed

for i in range(36):
    if i % 3 == 0:
        t.color("green")
    else:
        t.color("purple")

    t.circle(100)
    t.right(10)

turtle.done()
```

Formulae

Appendix: Key Reference Points

This section typically contains key formulas. However, as this course primarily focuses on theoretical, conceptual, or programming methodologies, mathematical formulae are not applicable.