App (DI02016011) - 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

Strings are ordered sequences of characters used to store and represent text. Lists are ordered, mutable collections of items that can hold elements of varying data types. Both are foundational data structures in Python that support indexing and slicing.

Methodology:

Working with Strings and Lists

1. **Creating**: Assign characters inside quotes for strings (e.g. `s = "Hello"`). Assign comma-separated items inside square brackets for lists (e.g. `L = [1, 2, 3]`).
2. **Accessing Items**: Use zero-based index `[i]` to access individual elements. Use negative indexing `[-1]` to access from the end.
3. **Slicing**: Extract a sub-sequence using `[start:stop:step]`.
4. **List Operations**: Append items using `append()`, insert using `insert()`, and remove using `remove()` or `pop()`.

Worked Example:

String and List Manipulation Problem **Problem:** Given a string `"GTU Diploma"` and a list `[10, 20, 30]`, extract the word "Diploma" from the string, add 40 to the end of the list, and reverse the list.
Solution Step-by-Step:

1. Define the string: `s = "GTU Diploma"`.
2. Slice the string from index 4 to the end to get "Diploma": `word = s[4:]`.
3. Define the list: `L = [10, 20, 30]`.
4. Append 40: `L.append(40)`. The list becomes `[10, 20, 30, 40]`.
5. Reverse the list using slicing: `L_reversed = L[::-1]`.
6. Final outputs: `word` is `"Diploma"` and `L_reversed` is `[40, 30, 20, 10]`.

1.2 Set

A set is an unordered, unindexed collection of unique elements. Sets are highly optimized for checking whether a specific element is contained within them and for mathematical set operations.

Methodology:

Set Creation and Modification Methods

1. **Create a Set**: Enclose elements in curly braces `{}` or use the `set()` constructor. Example: `S = {1, 2, 3}`. Note: An empty set must be created using `set()` and not `{}`.
2. **Accessing Sets**: Because sets are unordered, elements cannot be accessed by index. Instead, iterate using a `for` loop or check membership using the `in` keyword.
3. **Update Sets**: Add a single item using `add(item)`. Add multiple items using `update(iterable)`.
4. **Delete from Sets**: Use `remove(item)` (raises an error if item does not exist) or `discard(item)` (does not raise an error). `pop()` removes an arbitrary element. `clear()` empties the set.
5. **Python Set Operations**:
   - **Union** (`|`): Combines elements from both sets.
   - **Intersection** (`&`): Returns elements common to both sets.
   - **Difference** (`-`): Returns elements in the first set but not in the second.
   - **Symmetric Difference** (`^`): Returns elements present in either set but not both.

Worked Example:

Applying Set Operations **Problem:** Given two sets `A = {1, 2, 3, 4}` and `B = {3, 4, 5, 6}`, find their union, intersection, difference `A - B`, and symmetric difference. Also, add the element `7` to set A and remove `1`.

Solution Step-by-Step:

1. **Union**: Elements in either A or B. `A | B = {1, 2, 3, 4, 5, 6}`.
2. **Intersection**: Elements in both A and B. `A & B = {3, 4}`.
3. **Difference (A - B)**: Elements in A excluding those in B. `A - B = {1, 2}`.
4. **Symmetric Difference**: Elements in A or B but not both. `A ^ B = {1, 2, 5, 6}`.
5. **Update**: `A.add(7)` changes A to `{1, 2, 3, 4, 7}`.
6. **Delete**: `A.remove(1)` changes A to `{2, 3, 4, 7}`.

1.3 Tuple

A tuple is an ordered collection of elements similar to a list, but it is immutable. Once a tuple is created, its elements cannot be modified, added, or removed.

Methodology:

Tuple Creation and Operations

1. **Creating Tuples**: Enclose comma-separated values in parentheses `()` or use the `tuple()` constructor. A single-element tuple requires a trailing comma: `T = (5,)`.
2. **Accessing and Slicing**: Access elements using `[index]` and extract sub-tuples using `[start:stop:step]`.
3. **Iterate Over Tuples**: Use a `for` loop to iterate through each item: `for item in T:`.
4. **Tuples are Immutable**: You cannot assign a new value to a specific index. Trying to do so (e.g. `T[0] = 10`) will raise a `TypeError`.

5. **Python Tuple Operations**: Tuples support concatenation (`+`) and repetition (`*`).
6. **Built-In Tuple Functions and Methods**:
   - **Functions**: `len(T)`, `max(T)`, `min(T)`, `sum(T)`.
   - **Methods**: `T.count(x)` returns the number of occurrences of x. `T.index(x)` returns the first index of x.

Worked Example:

Processing Data Using Tuples **Problem:** Let `T = (10, 20, 30, 20, 40, 50)`. Find the length of the tuple, the index of the first occurrence of `20`, the total number of times `20` appears, and slice the tuple to reverse it.
Solution Step-by-Step:

1. **Length**: Using `len(T)` evaluates to `6`.
2. **Index**: Using `T.index(20)` scans the tuple and evaluates to `1` (since 20 first appears at index 1).
3. **Count**: Using `T.count(20)` counts all occurrences, evaluating to `2`.
4. **Slicing**: Using `T[::-1]` returns a new reversed tuple: `(50, 40, 20, 30, 20, 10)`.
5. **Immutability Check**: Attempting `T[0] = 99` will throw a `TypeError` because tuples cannot be modified.

1.4 Dictionary

A dictionary is an ordered (as of Python 3.7+), mutable collection of key-value pairs. Dictionaries are highly optimized for data retrieval based on keys.

Methodology:

Dictionary Management and Methods

1. **Creating Dictionaries**: Enclose key-value pairs in curly braces `{key: value}` or use the `dict()` constructor.
2. **Properties of Dictionary Keys**: Keys must be immutable types (e.g. strings, numbers, tuples) and must be unique. Values can be of any data type and can be duplicated.
3. **Accessing Items**: Use `D[key]` to get a value. To avoid `KeyError` if the key is missing, use the `D.get(key)` method.
4. **Add and Update**: Add a new key-value pair using `D[new_key] = value`. Update an existing pair similarly or use the `D.update({key: new_value})` method.
5. **Remove**:
   - `pop(key)`: Removes the specified key and returns its value.
   - `popitem()`: Removes and returns the last inserted key-value pair.
   - `del D[key]`: Deletes the key-value pair.
   - `clear()`: Empties the entire dictionary.
6. **Built-In Methods**: `keys()` returns a view of all keys, `values()` returns a view of all values, and `items()` returns a view of all key-value tuples.

Worked Example:

Managing Student Records using Dictionary **Problem:** Initialize a dictionary for a student with keys `"name": "Amit"` and `"marks": 85`. Add a new key `"grade": "A"`, update `"marks"` to 90, and safely remove the key `"grade"`. Finally, retrieve all keys and values.
Solution Step-by-Step:

1. **Create**: `student = {"name": "Amit", "marks": 85}`.
2. **Add Item**: `student["grade"] = "A"`. The dictionary is now `{"name": "Amit", "marks": 85, "grade": "A"}`.
3. **Update Item**: `student["marks"] = 90`. The value for "marks" is modified.
4. **Remove Item**: `grade_val = student.pop("grade")`. The dictionary is back to `{"name": "Amit", "marks": 90}`.
5. **Access Methods**:
   - `student.keys()` yields `dict_keys(['name', 'marks'])`.
   - `student.values()` yields `dict_values(['Amit', 90])`.
   - `student.items()` yields `dict_items([('name', 'Amit'), ('marks', 90)])`.

1.5 Comprehensions

Comprehensions provide a concise and readable way to create collections based on existing iterables. They combine loops and conditional logic into a single line of code.

Methodology:

Comprehension Syntax for Different Types

1. **List Comprehension**: Uses square brackets.
   `[expression for item in iterable if condition]`
2. **Dictionary Comprehension**: Uses curly braces with key-value pairs.
   `{key_expr: value_expr for item in iterable if condition}`
3. **Set Comprehension**: Uses curly braces with a single expression.
   `{expression for item in iterable if condition}`
4. **Generator Comprehension**: Uses parentheses. Unlike list comprehensions, it does not store all items in memory at once but generates them on the fly.
   `(expression for item in iterable if condition)`

Worked Example:

Generating Data using Comprehensions **Problem:** Given a list of numbers `nums = [1, 2, 3, 4, 5]`, write four comprehensions to: 1) Create a list of their squares, 2) Create a dictionary mapping the number to its cube, 3) Create a set of their remainders when divided by 2, and 4) Create a generator for their squares.
Solution Step-by-Step:

1. **List Comprehension**:
   `squares_list = [x**2 for x in nums]`
   Output: `[1, 4, 9, 16, 25]`.
2. **Dictionary Comprehension**:

`cubes_dict = {x: x**3 for x in nums}`
Output: `{1: 1, 2: 8, 3: 27, 4: 64, 5: 125}`.

3. **Set Comprehension**:
`rem_set = {x % 2 for x in nums}`
Evaluates `1%2=1`, `2%2=0`, `3%2=1`, ...
Output: `{0, 1}` (since sets only keep unique values).

4. **Generator Comprehension**:
`squares_gen = (x**2 for x in nums)`
Output: Returns a generator object. You would iterate over `squares_gen` (e.g. using a `for` loop or `list()`) to yield `1, 4, 9, 16, 25` one by one, effectively saving memory for large datasets.

CHAPTER 2

Modules and Packages

2.1 Introduction to Modules and Creating User Defined Modules

A module in Python is simply a file containing Python definitions, functions, and statements. Modules allow us to logically organize Python code, making it easier to understand, maintain, and reuse.

Methodology:

Creating a User Defined Module To create a user defined module:

1. Create a new Python file with a `.py` extension (e.g. `math_operations.py`).
2. Define variables, functions, and classes inside this file.
3. Save the file. The file name (without the `.py` extension) becomes the module name.

Worked Example:

Creating a Basic Arithmetic Module **Problem:** Create a module named `arithmetic.py` that contains functions to add and subtract two numbers.
Solution:

1. Create a file named `arithmetic.py`.
2. Add the following code to the file:

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

def subtract(a, b):
    return a - b
```

2.2 Importing a Module in Python

Python provides multiple ways to import a module and access its contents.

Methodology:

Module Import Methods

1. **Normal import**: Imports the entire module. You must prefix module items with the module name.

```
import module_name
module_name.function_name()
```

2. **From import**: Imports specific items from a module directly into the current namespace.

```
from module_name import function_name
function_name()
```

3. **From import with ***: Imports all public items from a module into the current namespace. (Not recommended for large modules to avoid namespace collisions).

```
from module_name import *
function_name()
```

Worked Example:

Using Different Import Methods **Problem:** Given the `arithmetic.py` module from the previous example, demonstrate the three ways of importing and using its functions.
Solution:
1. Normal import

```
import arithmetic

result1 = arithmetic.add(10, 5)
print("Normal Import Add:", result1)
```

2. From import

```
from arithmetic import subtract

result2 = subtract(10, 5)
print("From Import Subtract:", result2)
```

3. From import with *

```
from arithmetic import *

result3 = add(20, 10)
print("Star Import Add:", result3)
```

2.3 Module Search Path

When a module is imported, Python searches for the module file in a specific order of directories.

Methodology:

Module Search Order Python searches for modules in the following order:

1. **Current Working Directory**: The directory from which the script is executed.
2. **PYTHONPATH**: A list of directories defined in the `PYTHONPATH` environment variable.
3. **Standard Library Directories**: The default directories where Python is installed.

The list of directories searched is stored in `sys.path`.

Worked Example:

Viewing the Module Search Path **Problem:** Write a Python script to display the directories Python searches when importing a module.
Solution:

1. Import the built-in `sys` module.
2. Print the `sys.path` list.

```
import sys

print("Module Search Path:")
for path in sys.path:
    print(path)
```

2.4 OS Module Operations

The `os` module provides functions for interacting with the operating system, particularly for file and directory management.

Methodology:

Handling Directories using os Module

1. **Get Current Working Directory**: `os.getcwd()` returns the absolute path of the current directory.
2. **Change Directory**: `os.chdir(path)` changes the current working directory to the specified path.
3. **Create a Directory**: `os.mkdir(path)` creates a new single directory.
4. **Create Directories Recursively**: `os.makedirs(path)` creates a directory and any missing parent directories.
5. **List Files and Directories**: `os.listdir(path)` returns a list of all files and directories in the specified path.

Worked Example:

Managing Directories with os Module **Problem:** Write a script to print the current directory, create a new directory named `test_dir`, change into it, and list the contents of the parent directory.
Solution:

```
import os

# 1. Get current working directory
current_dir = os.getcwd()
print("Current Directory:", current_dir)

# 2. Create a new directory
new_dir_name = "test_dir"
if not os.path.exists(new_dir_name):
    os.mkdir(new_dir_name)
    print(f"Directory '{new_dir_name}' created.")

# 3. Change directory
os.chdir(new_dir_name)
print("Changed Directory To:", os.getcwd())
```

```
# 4. List files and directories of the parent directory
os.chdir('..') # Go back to parent
contents = os.listdir('.')
print("Contents of Parent Directory:", contents)
```

2.5 Introduction to Packages and Creating User Defined Packages

A package is a hierarchical file directory structure that defines a single Python application environment consisting of modules and sub-packages. It allows organizing multiple related modules.

Methodology:

Creating a User Defined Package To create a package:

1. Create a main directory representing the package name (e.g. `my_package`).
2. Inside the directory, create an `__init__.py` file. This file can be empty but it indicates to Python that this directory should be treated as a package.
3. Add one or more Python module files (`.py`) inside the directory.

Worked Example:

Structuring a Simple Package **Problem:** Create a package named `geometry` that contains two modules: `area.py` (for area calculations) and `volume.py` (for volume calculations).
Solution:

1. Create a folder named `geometry`.
2. Inside `geometry`, create an empty file named `__init__.py`.
3. Create `area.py` inside `geometry`:

```
# geometry/area.py
def circle_area(radius):
    return 3.14159 * radius * radius

def rectangle_area(length, width):
    return length * width
```

4. Create `volume.py` inside `geometry`:

```
# geometry/volume.py
def cube_volume(side):
    return side * side * side
```

2.6 Importing a Package and Intra-package References

Packages can be imported similarly to individual modules. You can use dot notation to access modules inside a package.

Methodology:

Importing from Packages

1. **Importing a specific module from a package**:

```
import package_name.module_name
package_name.module_name.function()
```

2. **Importing a function directly from a package module**:

```
from package_name.module_name import function_name
function_name()
```

Worked Example:

Importing Modules from the Geometry Package **Problem:** Write a script outside the `geometry` package to calculate the area of a circle and the volume of a cube using the package modules.
Solution:

```
# main.py (located in the same directory as the geometry folder)

# Method 1: Import module from package
import geometry.area
area_result = geometry.area.circle_area(5)
print("Area of circle:", area_result)

# Method 2: Import function directly
from geometry.volume import cube_volume
volume_result = cube_volume(3)
print("Volume of cube:", volume_result)
```

Methodology:

Intra package References When modules inside the same package need to interact with each other, you can use relative imports based on the current module's position.

1. **Absolute Import**: Specifies the full path from the project root. (e.g. `from my_package import moduleB`).
2. **Relative Import**: Uses dots to indicate the current or parent directory.
   - `from . import moduleB` (Current directory)
   - `from .. subpackage import moduleC` (Parent directory)

Worked Example:

Using Relative Imports **Problem:** Suppose we add a `utils.py` module inside the `geometry` package. How can `area.py` import a function from `utils.py` using a relative import?
Solution: Inside `geometry/utils.py`:

```
# geometry/utils.py
def print_result(value):
    print("Calculated result is:", value)
```

Inside `geometry/area.py`, use a single dot (`.`) for a relative import:

```
# geometry/area.py
from . import utils

def circle_area(radius):
```

```
    area = 3.14159 * radius * radius
    utils.print_result(area)
    return area
```

2.7 Installing PIP and Python Packages

`pip` (Pip Installs Packages) is the standard package manager for Python. It allows you to install and manage additional libraries and dependencies that are not distributed as part of the standard library.

Methodology:

Managing Packages with PIP

1. **Check PIP Installation**: Open the terminal/command prompt and verify pip is installed by checking its version.

   ```
   pip --version
   ```

2. **Install PIP** (If not installed): Download `get-pip.py` and run it using Python.

   ```
   python get-pip.py
   ```

3. **Install a Package**: Use the `install` command followed by the package name (e.g. `requests` or `numpy`).

   ```
   pip install package_name
   ```

4. **Install a Specific Version**:

   ```
   pip install package_name==1.0.4
   ```

5. **Uninstall a Package**: Use the `uninstall` command to remove a package.

   ```
   pip uninstall package_name
   ```

6. **List Installed Packages**: View all packages currently installed in the environment.

   ```
   pip list
   ```

Worked Example:

Installing and Uninstalling the Requests Package **Problem:** Demonstrate the command-line steps to install the `requests` library, verify its installation, and then uninstall it.
Solution:

1. Open your operating system's terminal or command prompt.
2. Execute the installation command:

```
pip install requests
```

3. Verify the package is installed by listing all packages:

```
pip list
```

4. Once done, uninstall the package. You will be prompted to confirm the removal (type `y` and press Enter):

```
pip uninstall requests
```

CHAPTER 3

Exception Handling

3.1 Introduction to Exception

In Python programming, errors are broadly classified into syntax errors and exceptions. Syntax errors occur during parsing, whereas exceptions occur during execution when a syntactically correct statement fails. Exception handling provides a computational mechanism to manage these runtime errors gracefully without crashing the program, allowing developers to detect, isolate, and resolve problematic states during execution.

Methodology:

Basic Exception Handling Strategy

1. Identify the specific code segments that involve external inputs, resources, or mathematical operations that may cause a runtime error.
2. Wrap this code inside a `try` block to monitor for exceptions.
3. Define one or more `except` blocks to catch and handle specific error types logically.
4. Utilize a `finally` block for deterministic cleanup actions (such as closing files or releasing network connections) that must execute regardless of whether an exception occurred.

3.2 Types of Exceptions

Python exceptions are categorized into predefined errors generated by the interpreter and custom errors defined by the programmer to enforce specific application logic.

3.2.1 Built-in Exceptions

Built-in exceptions are predefined by Python and automatically raised during specific computational error conditions.

Methodology:

Common Built-in Exceptions

1. **ZeroDivisionError**: Raised when division or modulo by zero takes place for any numeric type.
2. **ValueError**: Raised when an operation or function receives an argument that has the correct data type but an inappropriate value.
3. **TypeError**: Raised when an operation or function is applied to an object of an inappropriate or incompatible data type.
4. **IndexError**: Raised when trying to access a sequence subscript or index that is out of range.

3.2.2 User Defined Exceptions

Programmers can create custom exceptions by declaring a new class that derives from Python's base `Exception` class. This is useful for domain-specific constraints.

Methodology:

Creating User Defined Exceptions

1. Define a new custom class that explicitly inherits from the `Exception` base class.
2. Initialize the class using the `__init__` method to accept and format custom error messages or diagnostic values.
3. Call the parent constructor using `super().__init__()` to properly register the error message.

Worked Example:

Implementing a Custom Exception for Negative Values **Problem:** Write a program that accepts a positive integer from the user. If the user enters a negative integer, raise a custom exception `NegativeNumberError`.
Solution:

1. **Step 1: Define the custom exception class.**

```
class NegativeNumberError(Exception):
    def __init__(self, value):
        self.value = value
        self.message = f"Negative numbers are not allowed: {value}"
        super().__init__(self.message)
```

2. **Step 2: Write the logic to check the condition.**

```
def process_positive_number(num):
    if num < 0:
        raise NegativeNumberError(num)
    return num ** 2
```

3. **Step 3: Handle the user defined exception.**

```
try:
    result = process_positive_number(-5)
except NegativeNumberError as e:
    print(e)
```

Output: `Negative numbers are not allowed:  -5`

3.3 Raising Exceptions

The `raise` statement allows the programmer to forcefully trigger an exception when a specific state or invalid condition occurs in the program logic. This prevents invalid data from propagating further into computations.

Methodology:

Algorithm for Raising Exceptions

1. Evaluate a logical condition, data validation constraint, or mathematical domain restriction in the program.
2. If the constraint is violated, execute the `raise` statement followed by the exception class name.
3. Pass a descriptive error message string or specific values as arguments to the exception class to aid in debugging.
4. Allow the calling function to handle the raised exception using standard `try-except` structures.

Worked Example:

Raising a ValueError for Invalid Data **Problem:** Create a function that calculates the square root of a number. Forcefully raise a `ValueError` if the input number is negative, as real square roots of negative numbers are mathematically undefined in this context.
Solution:

1. **Step 1: Implement the function with a condition to raise the exception.**

```
import math

def calculate_sqrt(number):
    if number < 0:
        raise ValueError("Cannot compute real square root of a negative number")
    return math.sqrt(number)
```

2. **Step 2: Call the function inside a try-except block.**

```
try:
    val = calculate_sqrt(-16)
except ValueError as ve:
    print("Computation Failed:", ve)
```

Output: `Computation Failed:  Cannot compute real square root of a negative number`

3.4 Handling Exceptions

Exception handling relies on a robust combination of `try`, `except`, and `finally` clauses to maintain control flow.

Methodology:

Syntax and Flow of Handling Blocks

1. **Try clause**: Contains the primary logic block that executes normally but is actively monitored for runtime errors.
2. **Except clause**: Specifies the code that executes exclusively if a matching exception is raised in the `try` block. Multiple `except` blocks can exist sequentially to filter different error types.
3. **Finally clause**: Contains cleanup code that is guaranteed to run after the `try` and `except` blocks finish, regardless of whether an exception was thrown, caught, or left unhandled.

Worked Example:

Handling Multiple Exceptions with Cleanup **Problem:** Write a program to read two numbers as strings, convert them to integers, and perform division. Handle `ValueError` if conversion fails and `ZeroDivisionError` if the denominator is zero. Ensure a final completion message is printed using a finally clause.
Solution:

1. **Step 1: Define the try block with potentially failing operations.**

```
def safe_divide(str_a, str_b):
    try:
        a = int(str_a)
        b = int(str_b)
        result = a / b
        print(f"Result: {result}")
```

2. **Step 2: Define specific except blocks.**

```
    except ValueError:
        print("Error: Input strings must represent valid integers.")
    except ZeroDivisionError:
        print("Error: Division by zero is mathematically undefined.")
```

3. **Step 3: Define the finally block.**

```
    finally:
        print("Execution of safe_divide completed.")
```

4. **Step 4: Test the function with a zero denominator.**

```
safe_divide("10", "0")
```

Output:

```
Error: Division by zero is mathematically undefined.
Execution of safe_divide completed.
```

3.5 Python Assertions

An assertion is a computational sanity check that allows you to test the fundamental correctness of internal states. If the condition evaluates to True, the program execution continues seamlessly. If it evaluates to False, the interpreter raises an `AssertionError` exception.

Methodology:

Using the Assert Statement

1. Identify a logical invariant or state condition that **must** be universally true for the subsequent code to proceed securely.
2. Implement the syntax: `assert <condition>, <error_message>`

3. If `<condition>` evaluates to `False`, Python immediately halts execution and raises an `AssertionError` containing the provided `<error_message>`.

4. Utilize assertions strictly for internal debugging and invariant checking, rather than replacing standard `try-except` validation for expected user input errors.

Worked Example:

Applying Assertions for Formula Validation **Problem:** Write a function to calculate a final discounted price. Enforce that the original base price is strictly positive and the discount percentage falls rigorously between 0 and 100 inclusive. Use assertions to halt execution if these rules are violated.
Solution:

1. **Step 1: Define the function and insert assertions before the computation.**

```
def apply_discount(price, discount_percent):
    assert price > 0, "Base price must be strictly greater than 0"
    assert 0 <= discount_percent <= 100, "Discount percentage must be [0, 100]"

    discount_amount = price * (discount_percent / 100.0)
    return price - discount_amount
```

2. **Step 2: Test with valid computational data.**

```
final_price = apply_discount(200, 15)
print(f"Final price: {final_price}")
```

Output: `Final price:  170.0`

3. **Step 3: Test with invalid data to trigger the assertion.**

```
try:
    apply_discount(200, 110)
except AssertionError as ae:
    print(f"Assertion failed: {ae}")
```

Output: `Assertion failed:  Discount percentage must be [0, 100]`

CHAPTER 4

Files Handling

4.1 Introduction to Files and Their Types

Methodology:

File Types Files are used to store data permanently on a secondary storage device. In Python files are mainly classified into two types:

1. **Text Files:** Store data in plain text format (like strings). Each line is terminated with a special character called EOL (End of Line) which is usually the newline character (`\n`).
2. **Binary Files:** Store data in binary format (0s and 1s) which is the exact same format as it is held in memory. Examples include images, audio files, and compiled code. They are not human-readable but are faster to process.

4.2 Opening and Closing Text File

Methodology:

File Operations Basics To work with files in Python you must follow these steps:

1. **Open:** Open the file using the `open()` function. It returns a file object.
2. **Process:** Read from or write to the file.
3. **Close:** Close the file using the `close()` method to free up system resources.

Syntax:

```
file_object = open("filename.txt", "mode")
# Perform operations
file_object.close()
```

Common modes:

- `'r'`: Read (default mode). Error if file does not exist.
- `'w'`: Write. Creates a new file or overwrites an existing file.
- `'a'`: Append. Adds data to the end of an existing file or creates a new file.

Worked Example:

Opening and Closing a Text File Write a Python program to open a file named `data.txt` in write mode and close it properly. Check the file status before and after closing.
Solution:

```
# Step 1: Open the file
f = open("data.txt", "w")

# Step 2: Verify if file is closed
print(f"Is file closed? {f.closed}")

# Step 3: Close the file
f.close()

# Step 4: Verify after closing
print(f"Is file closed? {f.closed}")
```

4.3 Handle file using 'with' statement

Methodology:

Using the with Statement The `with` statement provides a cleaner and safer way to handle files. It automatically takes care of closing the file once the block of code is executed even if an exception occurs during file processing.

1. Syntax: `with open(filename, mode) as file_object:`
2. Perform read/write operations inside the indented block.
3. No need to explicitly call `close()`; Python closes it automatically.

Worked Example:

File Handling with 'with' Statement Demonstrate how to write to a file and read from it using the `with` statement.

Solution:

```
# Writing to a file automatically closes it after the block
with open("sample.txt", "w") as f:
    f.write("Hello World!\n")
    f.write("Learning Python File Handling.\n")

# Reading from a file
with open("sample.txt", "r") as f:
    content = f.read()
    print("File Content:\n" + content)
```

4.4 Reading and Writing Files

Methodology:

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

1. `write(string)`: Inserts the string into the file.
2. `writelines(list)`: Inserts a list of strings into the file.

Reading Methods:

1. `read(size)`: Reads the specified number of bytes. If no size is provided it reads the entire file.
2. `readline()`: Reads one complete line from the file.

3. `readlines()`: Reads all lines and returns them as a list of strings.

Worked Example:

Reading and Writing Text Files Write a Python script to create a file `marks.txt` store three student names and their marks and then read the contents line by line.
Solution:

```
# Writing data using writelines
students = ["Alice: 85\n", "Bob: 90\n", "Charlie: 78\n"]
with open("marks.txt", "w") as file:
    file.writelines(students)

# Reading data line by line
print("Reading file contents:")
with open("marks.txt", "r") as file:
    for line in file.readlines():
        print(line.strip()) # strip() removes the trailing newline
```

4.5 Setting Offsets in File

Methodology:

File Pointers seek and tell In Python file objects maintain a pointer indicating the current position for reading or writing.

1. `tell()`: Returns the current position of the file pointer (in bytes from the beginning).
2. `seek(offset, whence)`: Changes the file pointer position.
    - `offset`: Number of bytes to move.
    - `whence`: Reference point. `0` (beginning of file default) `1` (current position) `2` (end of file).

Note: In Python 3 for text files whence values 1 and 2 are usually restricted unless offset is 0. They work perfectly in binary mode (`'rb'`).

Worked Example:

Using tell and seek Methods Write a program to demonstrate the use of `seek()` and `tell()` methods to read specific parts of a text file.
Solution:

```
with open("seek_demo.txt", "w") as f:
    f.write("Python Programming")

with open("seek_demo.txt", "r") as f:
    print("Initial position:", f.tell())

    # Read first 6 characters
    print("Read:", f.read(6))
    print("Current position:", f.tell())

    # Move pointer to the 7th byte (index 7)
    f.seek(7)
    print("Position after seek(7):", f.tell())
```

```
    print("Read from current:", f.read())
```

4.6 Object Serialization - Pickle Module

Methodology:

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

1. Import the module: `import pickle`
2. Open file in binary mode: `'wb'` for writing `'rb'` for reading.
3. `pickle.dump(object, file)`: Serialize and write the Python object to the file.
4. `pickle.load(file)`: Read from the file and deserialize it back into a Python object.

Worked Example:

Storing and Retrieving Python Objects Create a dictionary of student grades. Serialize it into a file using the `pickle` module and then retrieve and print it.
Solution:

```
import pickle

grades = {'Alice': 'A', 'Bob': 'B+', 'Charlie': 'A-'}

# Serialization (Writing)
with open("grades.dat", "wb") as f:
    pickle.dump(grades, f)
print("Data serialized successfully.")

# Deserialization (Reading)
with open("grades.dat", "rb") as f:
    loaded_grades = pickle.load(f)

print("Deserialized Data:", loaded_grades)
```

4.7 Introduction to Python JSON

Methodology:

JSON Operations JSON (JavaScript Object Notation) is a lightweight human-readable data-interchange format mostly used for text files and web APIs.

1. Import the module: `import json`
2. **Python to JSON (Serialization):**
   - `json.dumps(python_obj)`: Converts a Python object into a JSON formatted string.
   - `json.dump(python_obj, file)`: Writes a Python object directly to a JSON file.
3. **JSON to Python (Deserialization):**
   - `json.loads(json_string)`: Parses a JSON string and converts it into a Python dictionary.
   - `json.load(file)`: Reads a JSON file and parses it into a Python object.

Worked Example:

Converting Between JSON and Python Objects Write a program to demonstrate conversion from a Python dictionary to a JSON string and vice versa.
Solution:

```
import json

# 1. Python object (Dictionary) to JSON String
emp_dict = {"name": "John", "age": 30, "city": "New York"}
json_str = json.dumps(emp_dict)
print("JSON String:", json_str)
print("Type of json_str:", type(json_str))

# 2. JSON String back to Python object
parsed_dict = json.loads(json_str)
print("\nPython Dictionary:", parsed_dict)
print("Type of parsed_dict:", type(parsed_dict))
print("Employee Name:", parsed_dict["name"])
```

Worked Example:

Reading and Writing JSON Files Write a Python program to save a Python dictionary to a JSON file and then read it back.
Solution:

```
import json

data = {
    "employee": "John Doe",
    "skills": ["Python", "Machine Learning"],
    "experience_years": 5
}

# Writing Python object to JSON file
with open("data.json", "w") as f:
    json.dump(data, f, indent=4) # indent makes it readable

# Reading JSON file into Python object
with open("data.json", "r") as f:
    loaded_data = json.load(f)
```

```
print("Data from JSON file:")
print(loaded_data)
```

CHAPTER 5

Graphics with Turtle

5.1 Introduction to Turtle Graphics

Turtle graphics is a popular way for introducing programming to beginners. It was part of the original Logo programming language developed by Wally Feurzeig, Seymour Papert and Cynthia Solomon in 1967. Python includes a standard library called `turtle` that provides turtle graphics primitives. Imagine a robotic turtle starting at $(0, 0)$ in an x-y plane. You can give it commands like `turtle.forward(15)` to move ahead, or `turtle.right(25)` to turn, leaving a trail (a line) as it moves.

Methodology:

Initializing Turtle Graphics To begin drawing with the `turtle` module, follow these computational steps:

1. **Import the module:** Use `import turtle` to make the graphics functions available.
2. **Initialize the screen:** Create a canvas or screen for the turtle to move on using `screen = turtle.Screen()`.
3. **Create a turtle instance:** Instantiate a turtle object using `t = turtle.Turtle()`.
4. **Issue commands:** Call methods on the turtle object to move it and draw shapes.
5. **Keep the window open:** End the program with `turtle.done()` to ensure the drawing window does not close immediately after the drawing finishes.

5.2 Turtle Methods

The turtle object provides numerous methods to control its movement and drawing characteristics. By combining these basic commands, complex geometric shapes and patterns can be generated algorithmically.

Methodology:

Basic Movement and Rotation Methods These methods alter the position and orientation of the turtle:

1. `forward(d)` or `fd(d)`: Moves the turtle forward by `d` units in the direction it is currently heading.
2. `backward(d)` or `bk(d)`: Moves the turtle backward by `d` units.
3. `right(angle)` or `rt(angle)`: Rotates the turtle's orientation clockwise by `angle` degrees.
4. `left(angle)` or `lt(angle)`: Rotates the turtle's orientation counter-clockwise by `angle` degrees.
5. `goto(x, y)` or `setpos(x, y)`: Moves the turtle to the absolute coordinate (x, y).
6. `setheading(angle)` or `seth(angle)`: Sets the orientation of the turtle to `angle`. (0 is East, 90 is North, 180 is West, 270 is South).

Methodology:

Pen Control and Drawing Methods These methods control how the turtle draws its trail:

1. `penup()` or `pu()`: Lifts the pen. The turtle will not draw a line when it moves.
2. `pendown()` or `pd()`: Lowers the pen. The turtle will draw a line when moving.
3. `pensize(width)`: Changes the thickness of the line drawn by the turtle to `width`.
4. `color(colorstring)`: Changes the color of the turtle's pen (e.g., `"red"`, `"blue"`).
5. `circle(radius)`: Draws a circle with the specified `radius`. The center is `radius` units left of the turtle.
6. `dot(size, color)`: Draws a circular dot of the specified `size` and `color` at the current position.

Worked Example:

Drawing Basic Shapes **Problem:** Write a Python program to draw an equilateral triangle of side length 150 units.
Solution: An equilateral triangle has three equal sides and three internal angles of 60°. The turtle must turn by the exterior angle, which is $180° - 60° = 120°$.

```
import turtle

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

# Draw the first side
t.forward(150)
t.left(120)

# Draw the second side
t.forward(150)
t.left(120)

# Draw the third side
t.forward(150)
t.left(120)

turtle.done()
```

5.3 Turtle Screen Methods

The Screen object represents the window where the turtle draws. It provides methods to modify the canvas size, background color, and window title.

Methodology:

Screen Setup and Control Methods

1. `bgcolor(colorstring)`: Sets the background color of the turtle screen.
2. `title(titlestring)`: Sets the title of the turtle graphics window.
3. `setup(width, height)`: Sets the dimensions of the main window. Values can be pixels (integers) or fractions of the screen size (floats).

4. `clear()`: Deletes all drawings and resets all turtles to their initial state, leaving the background color unchanged.

5. `bye()`: Closes the turtle graphics window.

Worked Example:

Configuring the Graphics Window **Problem:** Write a Python program to create a 500x500 turtle graphics window with a black background and the title "My Drawing Canvas". Draw a yellow circle of radius 50 in the center.
Solution:

```
import turtle

# Screen configuration
screen = turtle.Screen()
screen.setup(500, 500)
screen.bgcolor("black")
screen.title("My Drawing Canvas")

# Turtle drawing
t = turtle.Turtle()
t.color("yellow")
t.circle(50)

turtle.done()
```

5.4 Turtle Programming - Loops and Conditional Statements

The true power of turtle graphics is unleashed when combined with control structures like loops and conditional statements. This allows for algorithmic generation of intricate patterns with minimal code.

Methodology:

Using Loops for Repetitive Patterns Algorithms for regular polygons or repeating patterns utilize `for` loops:

1. Identify the repeating segment of the drawing.

2. Determine the number of repetitions (N).

3. Enclose the drawing commands in a `for` loop iterating N times.

4. For regular polygons of N sides, the turn angle after each side is $\frac{360}{N}$ degrees.

Worked Example:

Drawing a Regular Polygon using a Loop **Problem:** Write a program to draw a regular octagon (8 sides) with side length 80 units using a loop.
Solution: The exterior angle for an octagon is $\frac{360}{8} = 45$ degrees.

```
import turtle

t = turtle.Turtle()

# Loop 8 times to draw the 8 sides
for _ in range(8):
    t.forward(80)
```

```
    t.left(45)

turtle.done()
```

Methodology:

Conditional Logic in Drawing Conditional statements (`if`, `elif`, `else`) can be used inside loops to dynamically alter drawing parameters based on the current iteration state or specific mathematical conditions.

Worked Example:

Alternating Colors with Conditionals **Problem:** Write a program to draw a dashed line of length 300, where every 20-unit dash alternates between red and blue.
Solution: We will loop 15 times ($15 \times 20 = 300$). We use the modulo operator to determine if the iteration index is even or odd, applying a different color accordingly.

```
import turtle

t = turtle.Turtle()
t.pensize(4)
t.penup()
t.goto(-150, 0) # Start from the left side

for i in range(15):
    # Conditional statement to alternate color
    if i % 2 == 0:
        t.color("red")
    else:
        t.color("blue")

    t.pendown()
    t.forward(15) # Draw dash
    t.penup()
    t.forward(5)  # Leave gap

turtle.done()
```

Worked Example:

Generating a Spiral Pattern **Problem:** Use a loop to draw a square spiral pattern.
Solution: A spiral is drawn by increasing the side length in each iteration while keeping the turn angle constant (90 degrees for a square spiral).

```
import turtle

t = turtle.Turtle()
t.speed(0) # Set to maximum drawing speed

side_length = 5

for i in range(50):
    t.forward(side_length)
    t.right(90)
    # Increment side length for the next iteration
    side_length = side_length + 5
```

```
turtle.done()
```

Formulae

Appendix: Key Formulae

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

Unit 5: Graphics with Turtle

Regular Polygon Drawing

In Python's Turtle graphics, regular polygons can be drawn using the `circle()` method by passing the number of sides to the `steps` parameter. The relationship is generalized as:

$$\text{steps} = n$$

where n represents the number of sides of the desired regular polygon (e.g., $n = 3$ for a triangle, $n = 6$ for a hexagon).