Python: Building Skills for Software Development

Dr Mahendra Singh Bora **[PhD (CS), MCA, PGDCA]**
[mahendra.singh.bora@gmail.com]

Mr Bhupendra Singh Latwal **[NET (CS), MCA, MTECH]**
[blatwal@gmail.com]

Mrs Shivangi Verma **[MCA, MTECH]**
[shivangi0007@gmail.com]

Published by
Singh Publication
78/77, New Ganesh Ganj,
Opposite Rajdhani Hotel, Aminabad Road
LUCKNOW- 226018 UTTAR PRADESH, INDIA
Email : info@singhpublication.com | https://singhpublication.com

Copyright © Author

All rights reserved. No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical including photocopy, recording or by any information storage and retrieval system, without permission in writing from the copyright owner.

ISBN
978-81-964979-1-0

DOI
10.5281/zenodo.10894298

First Published
March 2024

All disputes are subject to Lucknow jurisdiction only.

Printed At
Yellow Print,
G-4, Goel Market, Lekhraj Metro Station,
Indiranagar, Lucknow, India.
Ph: +91-7499403012
E-mail: yellowprints12@gmail.com

AUTHORS ARE FULLY LIABLE FOR ORIGINALITY AND WORDING

Every effort has been made to avoid errors or omissions in this publication. In spite of this, some errors might have crept in. Any mistake, error or discrepancy noted may be brought to our notice which shall be taken care of in the next edition. It is notified that neither the publisher nor the author or seller will be responsible for any damage or loss of action to anyone, of any kind, in any manner, therefrom. For binding mistakes, misprints or for missing pages etc., the publisher's liability is limited to replacement within one month of purchase by similar edition. All expenses in this condition are to be borne by the purchaser.

Content

Preface

This book is designed to be a comprehensive resource for both beginners and experienced programmers who want to learn or expand their knowledge of the Python programming language. Python is known for its simplicity and versatility, making it an ideal language for a wide range of applications, from web development to data science.

Who This Book Is For

Beginners: If you're new to programming, this book will provide you with a gentle introduction to Python, starting with the basics and gradually building your skills.

Intermediate Programmers: If you have some programming experience but want to dive into Python, this book will help you transition smoothly and master the language's advanced features.

Experienced Python Developers: Even if you're already familiar with Python, you'll find value in the book's in-depth coverage of advanced topics, best practices, and real-world projects.

What You Will Learn

- The book covers a wide range of Python topics, including:
- Python syntax and basic programming concepts
- Data structures like lists, dictionaries, and sets
- Object-oriented programming (OOP) and design principles
- File handling and working with external data
- Advanced Python topics like decorators and generators
- Web development with Python, using Flask and Django
- Data science and data analysis with Python libraries
- Testing, debugging, and deployment best practices
- Real-world projects to apply your knowledge

How This Book Is Organized

The book is divided into chapters, with each chapter focusing on a specific aspect of Python. We encourage you to read the chapters sequentially if you're new to Python, as they build upon each other.However, experienced programmers may find it useful to skip to specific chapters based on their interests or needs.

Code Examples

Throughout the book, you'll find numerous code examples and exercises. You can practice by typing out the code and experimenting with it in your Python environment. Code samples are available for download from our website.

Conventions Used in This Book

- Code examples are displayed in a monospaced font like this: print("Hello, World!").
- Important concepts and key terms are highlighted in bold.

Feedback and Corrections

We appreciate your feedback and any corrections you may discover. Please email us or contact us through our publisher to provide feedback or report any errors.

We hope you find this book to be a valuable resource on your journey to mastering Python. Whether you're looking to start a new career, enhance your skills, or simply have fun with programming, Python has something to offer you. Enjoy your learning journey!

Acknowledgements

First and foremost, I would like to express my deepest gratitude to the Python community. The open-source nature of Python and the collaborative spirit of its community have been instrumental in the creation of this book.

I would also like to thank my colleagues and friends in the programming world who have provided invaluable feedback and insights throughout the writing process. Your expertise and encouragement have been greatly appreciated.

Special thanks go to my editor and the publishing team for their patience, guidance, and hard work in bringing this book to life.

Lastly, I want to acknowledge all the readers of this book. Whether you are a beginner just starting your coding journey or an experienced developer looking to expand your skills, I hope this book will be a useful resource in your programming endeavors.

Thank you to everyone who contributed to this project in various ways. Your support has been invaluable, and we appreciate each and every one of you.

Warm regards,

Dr Mahendra Singh Bora **[PhD (CS), MCA, PGDCA]**
[mahendra.singh.bora@gmail.com]

Mr Bhupendra Singh Latwal **[NET(CS), MCA, MTECH]**
[blatwal@gmail.com]

Mrs Shivangi Verma **[MCA, MTECH]**
[shivangi0007@gmail.com]

Chapter 1
Introduction to Python Programming

What is Python

Python is a high-level, versatile, and easy-to-learn programming language known for its simplicity and readability. It was created by Guido van Rossum and first released in 1991. Python has gained immense popularity over the years and is widely used in various fields, including web development, data analysis, scientific computing, artificial intelligence, and more. Here are some key characteristics and uses of Python:

Readability: Python's syntax is designed to be clear and easy to understand, making it an excellent choice for beginners and experienced developers alike. The use of indentation (whitespace) for code blocks enhances code readability.

Versatility: Python is a general-purpose programming language, meaning it can be used for a wide range of applications. You can write scripts, build web applications, create desktop software, perform data analysis, and even develop games using Python.

Large Standard Library: Python comes with a comprehensive standard library that includes modules and packages for a variety of tasks, from working with files and data to web development and more. This extensive library reduces the need to write code from scratch and accelerates development.

Community and Ecosystem: Python has a large and active community of developers. There are countless open-source libraries and frameworks available for Python, making it easier to find solutions for specific tasks. Some popular libraries include NumPy, Pandas, Django, Flask, TensorFlow, and many others.

Cross-Platform: Python is available on various platforms, including Windows, macOS, and Linux. You can write code on one platform and run it on another without significant modifications.

Interpreted Language: Python is an interpreted language, meaning you don't need to compile your code before running it. This makes the development process more agile and user-friendly.

Dynamic Typing: Python uses dynamic typing, which means you don't need to declare the data type of a variable explicitly. Python determines the data type dynamically during runtime.

Open Source: Python is open-source and freely available. This makes it accessible to anyone who wants to use and contribute to its development.

Highly Extensible: Python can be easily extended with modules and packages written in other languages like C or C++. This allows for high performance and compatibility with existing code bases.

Scientific and Data Analysis: Python is widely used in scientific and data analysis, thanks to libraries like NumPy, Pandas, and Matplotlib. It's a popular choice for data scientists and researchers.

Web Development: Python offers various frameworks for web development, such as Django and Flask, which simplify the process of building web applications.

Artificial Intelligence and Machine Learning: Python is a go-to language for artificial intelligence and machine learning projects. Libraries like TensorFlow, Keras, and scikit-learn provide powerful tools for AI and ML development.

Automation and Scripting: Python is often used for automating tasks, creating scripts, and simplifying repetitive activities, making it a valuable tool for system administrators and DevOps professionals.

In summary, Python is a versatile and powerful programming language that is widely used in many fields due to its simplicity, readability, and the extensive ecosystem of libraries and frameworks available. It's an excellent choice for both beginners and experienced developers looking to solve a wide range of problems efficiently.

Installation and Setup

Setting up Python on your computer is typically a straightforward process, and there are different approaches to install and configure Python depending on your operating system. Here are general steps for installing and setting up Python:

Step 1: Choose a Python Version: Python has two major versions in use: Python 2 and Python 3. It's recommended to use Python 3, as Python 2 is no longer actively maintained. Choose the latest Python 3 version available.

Step 2: Download Python:

- Visit the official Python website at python.org and click on the "Downloads" section.
- Select the version of Python you want to install (e.g., Python 3.9.7).
- Choose the installer appropriate for your operating system (Windows, macOS, or Linux).

Step 3: Run the Installer: Note: The following steps may vary slightly depending on your operating system.

For Windows:

- Double-click the downloaded installer.
- Make sure to check the box that says "Add Python x.x to PATH" during installation to make Python accessible from the command line.
- Follow the installation prompts and complete the setup.

For macOS:

- Double-click the downloaded package file.
- Follow the installation instructions, and Python will be installed on your system.

For Linux:

- Open a terminal.
- Navigate to the directory where you downloaded the installer.
- Run the following commands (replace python3.x.x with the version you downloaded):

```
tar -xvf Python-3.x.x.tgz
cd Python-3.x.x
./configure
make
sudo make install
```

Step 4: Verify Installation:

To ensure that Python was installed correctly, open a command prompt or terminal and type:

```
python--version
```

This should display the version of Python you installed. If it doesn't, try running:

```
python3 --version
```

This will verify the installation of Python 3.

Step 5: Install a Text Editor or Integrated Development Environment (IDE):

You can write Python code in any text editor, but using an IDE designed for Python development can enhance your coding experience. Some popular Python IDEs include ***PyCharm, Visual Studio Code, and Jupyter Notebook.***

Your First Python Program

Your first Python program is often a simple "Hello, World!" program. Here's how you can write and run your first Python program:

Open a Text Editor: You can use any text editor to write your Python code. Notepad (on Windows), TextEdit (on macOS), or a code-focused editor like Visual Studio Code, PyCharm, or IDLE are good choices.

Write the Python Code: Open your text editor and write the following code:

```
print("Hello,World!")
```

This code uses the print() function to display the text "Hello, World!" on the screen.

Save the File:

- Give your file a name, such as hello.py. The .py extension indicates that this is a Python script.
- Choose a location to save your file.

Run the Python Program:

The steps for running your Python program depend on your operating system:

For Windows:

- Open the Command Prompt (you can search for "cmd" in the Start menu).
- Navigate to the directory where you saved your hello.py file using the cd command.
- Run the program by typing:

```
python hello.py
```

For macOS and Linux:

- Open the Terminal.
- Navigate to the directory where you saved your hello.py file using the cd command.
- Run the program by typing:

```
python3 hello.py
```

After running the program, you should see "Hello, World!" displayed on the screen. Congratulations! You've successfully written and executed your first Python program. This simple example is just the beginning. As

you continue your Python journey, you'll explore more complex programs and learn about the language's features and capabilities.

Python Development Environments

When working with Python, you have a variety of development environments (IDEs) and code editors to choose from. The choice of environment largely depends on your personal preferences and project requirements. Here are some popular Python development environments and code editors:

Integrated Development Environments (IDEs):

PyCharm:

PyCharm is a highly regarded IDE for Python development. It offers a wide range of features, including code completion, debugging, and support for web development with Django and Flask.

Visual Studio Code (VSCode):

VSCode is a lightweight, open-source code editor developed by Microsoft. It has an extensive library of Python extensions that provide features like IntelliSense, debugging, and integrated terminals.

Jupyter Notebook:

Jupyter Notebook is an open-source web application that allows you to create and share documents that contain live code, equations, visualizations, and narrative text. It's widely used in data science and scientific computing.

Spyder:

Spyder is a scientific IDE specifically designed for data science and scientific development. It comes with built-in support for popular scientific libraries like NumPy and Matplotlib.

PyDev:

PyDev is an IDE for Python that can be integrated with the Eclipse platform. It provides features like code analysis, debugging, and code completion.

Thonny:

Thonny is a beginner-friendly Python IDE designed to make learning Python easy. It comes with a built-in package manager, debugger, and simple interface.

Code Editors:

Sublime Text:

Sublime Text is a lightweight, highly customizable code editor. You can enhance it with Python-specific plugins and extensions.

Atom:

Atom is an open-source code editor developed by GitHub. It's highly customizable and has a large community, which means there are many Python-related packages available.

Notepad++:

Notepad++ is a popular text editor for Windows. While it's not an IDE, it supports Python syntax highlighting and can be used for basic Python scripting.

Geany:

Geany is a lightweight code editor with basic IDE features. It supports Python and allows for customizations.

IDLE (Python's Built-In IDE):

Python comes with its integrated development and learning environment called IDLE. While it's not as feature-rich as other IDEs, it's a good choice for beginners.

Each of these development environments and code editors has its own strengths and features. The best choice for you will depend on your specific needs, your familiarity with the tools, and your personal preferences. It's a good idea to try out a few of them to see which one you're most comfortable and productive with in your Python development.

Chapter 2
Python Basics

Variables and Data Types

In Python, variables are used to store and manipulate data. Python is a dynamically-typed language, which means you don't need to declare the data type of a variable explicitly. The data type is determined dynamically during runtime. Python supports various data types, including:

1. **Integers (int):** Whole numbers, such as -5, 0, 42.

   Example:

   ```
   age = 30
   ```

2. **Floating-Point Numbers (float):** Numbers with a decimal point, such as 3.14, -0.5.

   Example:

   ```
   temperature = 98.6
   ```

3. **Strings (str):** Text or sequences of characters enclosed in single (' '), double (" "), or triple (''' ' or """ ") quotes.

   Example:

   ```
   name = "ram”
   message = 'Hello, World!'
   ```

4. **Boolean (bool):** Represents either True or False. Boolean values are often used for conditional statements and logical operations.

   Example:

   ```
   is_student = True
   is_adult = False
   ```

5. **Lists:** Ordered collections of items. Lists can contain elements of different data types.

   Example:

   ```
   fruits = ["apple", "banana", "cherry"]
   numbers = [1, 2, 3, 4, 5]
   ```

6. **Tuples:** Similar to lists but immutable, meaning their values cannot be changed once set.

   Example:

   ```
   coordinates = (3, 4)
   ```

7. **Dictionaries:** Collections of key-value pairs. Each key is unique and associated with a value.

   Example:

   ```
   person = {"name": "Alice", "age": 30, "city": "New York"}
   ```

2. **Sets:** Unordered collections of unique items.

   Example:

   ```
   unique_numbers = {1, 2, 3, 4, 5}
   ```

3. **None Type (NoneType):** Represents the absence of a value or a null value.

   Example:

   ```
   result = None
   ```

You can assign values to variables and perform various operations with these data types. For example, you can perform arithmetic operations on integers and floating-point numbers, concatenate strings, use conditional statements with booleans, and more.

Here's a simple example that demonstrates the use of variables and data types:

```
# Variables and data types
name = "ram"
age = 30
height = 5.7
is_student = True
fruits = ["apple", "banana", "mango"]
person = {"name": "mohan", "age": 25}
# Printing variable values
print(name)          # "ram"
print(age)           # 30
print(height)        # 5.7
print(is_student)    # True
print(fruits)        # ["apple", "banana", "mango"]
print(person)        # {"name": "mohan", "age": 25}
```

Python's dynamic typing and rich set of data types make it a versatile language for various applications and data manipulation tasks.

Operators and Expressions

In Python, operators and expressions are fundamental concepts that allow you to perform various operations and calculations. Operators are symbols that represent operations, and expressions are combinations of values and operators that produce a result. Python supports a wide range of operators and allows you to create complex expressions. Here are some of the most commonly used operators and how they work:

Arithmetic Operators:

- **Addition (+)**: Adds two values.
- **Subtraction (-)**: Subtracts the right operand from the left operand.
- **Multiplication (*)**: Multiplies two values.
- **Division (/)**: Divides the left operand by the right operand, producing a floating-point result.
- **Integer Division (//)**: Divides the left operand by the right operand, producing an integer result.
- **Modulus (%)**: Returns the remainder after division.
- **Exponentiation (**)**: Raises the left operand to the power of the right operand.

Example:

```
x = 10
y = 3
addition = x + y        # 13
division = x / y        # 3.3333...
modulus = x % y         # 1
```

Comparison Operators:

- **Equal (==):** Compares if two values are equal.
- **Not Equal (! =):** Compares if two values are not equal.
- **Greater Than (>):** Checks if the left operand is greater than the right operand.

- **Less Than (<):** Checks if the left operand is less than the right operand.
- **Greater Than or Equal To (>=):** Checks if the left operand is greater than or equal to the right operand.
- **Less Than or Equal To (<=):** Checks if the left operand is less than or equal to the right operand.

Example:

```
a = 5
b = 7
is_equal = a == b # False
is_not_equal = a != b # True
```

Logical Operators:

- **Logical AND (and):** Returns True if both operands are True.
- **Logical OR (or):** Returns True if at least one of the operands is True.
- **Logical NOT (not):** Negates the value of the operand.

Example:

```
x = True
y = False
logical_and = x and y        # False
logical_or = x or y          # True
logical_not = not x          # False
```

Assignment Operators:

- **Assignment (=):** Assigns a value to a variable.
- **Increment (+=):** Adds the right operand to the left operand and assigns the result to the left operand.
- **Decrement (-=):** Subtracts the right operand from the left operand and assigns the result to the left operand.
- **Multiply (*=):** Multiplies the left operand by the right operand and assigns the result to the left operand.
- **Divide (/=):** Divides the left operand by the right operand and assigns the result to the left operand.

Example:

```
count = 0
count += 1              # Increment count by 1
```

Bitwise Operators: These operators perform operations on individual bits of integers.

- Bitwise AND &
- Bitwise OR |
- Bitwise XOR ^
- Bitwise NOT ~
- Left Shift <<
- Right Shift >>

Example:

```
a = 5                   #binary code - 101
b = 6                   #binary code - 110
c = a & b
d = a | b
e = a ^ b
f = ~a
g = b<<2
h = b>>2
```

Membership Operators:

- **in:** Returns True if a value is found in a sequence (e.g., a list, string, or tuple).
- **not in:** Returns True if a value is not found in a sequence.

Example:

```
fruits = ["apple", "banana", "cherry"]
is_apple_in_list = "apple" in fruits          #True
is_mango_in_list = "mango" not in fruits      #True
```

Identity Operators:

- **is:** Returns True if both variables refer to the same object.
- **is not:** Returns True if both variables refer to different objects.

Example:

```
x = [1, 2, 3]
y = x
is_same_object = x is y                    #True
```

Ternary Operator:

- **Conditional Expression x if condition else y**: Returns x if the condition is True, otherwise returns y.

Example:

```
age = 25
category = "Adult" if age >= 18 else "Minor"      # "Adult"
```

Operator Precedence:

Operators in Python have different levels of precedence. For example, multiplication (*) has higher precedence than addition (+). You can use parentheses to control the order of operations.

Example:

```
result = (5 + 2) * 3      # Parentheses take precedence, result is 21
```

These are some of the most commonly used operators in Python. You can combine them to create complex expressions and perform a wide range of operations in your Python programs.

Control Flow (if, else, and while)

Control flow in Python refers to the order in which statements and instructions are executed. Python provides various control flow constructs, including if statements for conditional execution and while loops for repeated execution. Here's an overview of these control flow constructs:

1. Conditional Statements (if, elif, and else):

Conditional statements are used to execute different blocks of code based on certain conditions. The primary conditional statement is the**if statement**, and it may be followed by zero or more **elif (short for "else if") and an optional else block**.

```
if condition1:
    # Code to execute if condition1 is True
```

```
elif condition2:
    # Code to execute if condition2 is True
else:
    # Code to execute if none of the above conditions are True
```

Example:

```
age = 25
if age < 18:
    print("You are a minor.")
elif age >= 18:
    print("You are an adult.")
else:
   print("Age is not defined.")
```

2. Loops (while and for):

Loops are used to repeatedly execute a block of code. In Python, you can use while and for loops.

while Loop:

A while loop repeatedly executes a block of code as long as a specified condition is True.

```
while condition:
    # Code to execute while the condition is True
```

Example:

```
count = 0
while count < 5:
    print(f"Count: {count}")
    count += 1
```

for Loop:

A for loop is used to iterate over sequences, such as lists, tuples, strings, or other iterable objects.

```
for variable in iterable:
    # Code to execute for each item in the iterable
```

Example:

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(f"Current fruit: {fruit}")
```

3. Break and Continue Statements:

- The break statement is used to exit a loop prematurely. It is often used to stop a loop when a certain condition is met.
- The continue statement is used to skip the current iteration of a loop and continue to the next one.

Example (using break):

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
for number in numbers:
    if number == 5:
        break              # Exit the loop when number is 5
    print(number)
```

Example (using continue):

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
for number in numbers:
    if number % 2 == 0:
        continue           # Skip even numbers
    print(number)
```

Control flow constructs are essential for making decisions, looping through data, and controlling the execution of your Python programs. They allow you to create more dynamic and interactive applications.

Functions and Modules

Functions and modules are essential concepts in Python that help you organize and modularize your code. They allow you to break down complex programs into smaller, more manageable parts, making your code more readable, maintainable, and reusable. Here's an overview of functions and modules in Python:

Functions:

Defining a Function:

To define a function in Python, you use the def keyword followed by the function name and a pair of parentheses. You can also specify parameters within the parentheses, which are the inputs to the function.

```
def greet():
        print("Hello, Ram!")
```

Calling a Function:

To execute a function, you call it by using its name followed by parentheses. If the function expects parameters, you provide them within the parentheses.

```
greet()
```

Return Values:

Functions can return values using the return statement. This allows a function to produce a result that can be used in other parts of your program.

```
def add():
        x=7
        y=10
        result = x + y
        return result
```

Function Parameters:

Functions can accept parameters, which are values passed into the function when it is called. These parameters can be used within the function's body.

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

Default Parameters:

You can assign default values to function parameters, making them optional. If a value is not provided when the function is called, the default value is used.

```
def greet(name="User"):
    print(f"Hello, {name}!")
```

Modules:

Creating a Module:

A module in Python is a file containing Python code. You can create your own modules by organizing related functions and variables within a **.py** file.

Example: Create a file named **my_module.py** with the following content:

```
def square(x):
    return x ** 2


def cube(x):
    return x ** 3
```

Using a Module:

To use functions and variables defined in a module, you need to import the module in your Python code. You can then access the module's content using dot notation.

```
import my_module
result = my_module.square(5)
```

Namespace:

When you import a module, it creates a separate namespace for the module's content. This prevents naming conflicts with other variables and functions in your code.

Renaming a Module:

You can give a module an alias using the as keyword, which makes it easier to reference the module.

```
import my_module as mm
result = mm.cube(4)
```

Importing Specific Items:

You can import specific functions or variables from a module instead of importing the entire module.

```
from my_module import square
result = square(6)
```

Functions and modules are fundamental to structuring and organizing code in Python. They help you achieve better code reusability, maintainability, and readability. By creating and using functions and modules effectively, you can build complex programs more efficiently.

Chapter 3
Data Structure in Python

Python offers a variety of built-in data structures that you can use to organize and manipulate data efficiently. These data structures are fundamental to many programming tasks and are crucial for solving a wide range of problems. Here are some of the most commonly used data structures in Python:

List

A list is a versatile and commonly used data structure that allows you to store and manipulate a collection of items. Lists are ordered, mutable, and can contain elements of different data types. This is one of the fundamental data structures in Python. Here's how you can work with lists:

Creating a List:

To create a list, you enclose a comma-separated sequence of items within square brackets []. The items can be of different data types, including numbers, strings, and other objects.

```
fruits = ["apple", "banana", "cherry"]
numbers = [1, 2, 3, 4, 5]
mixed = ["apple", 2, 3.14, True]
```

Accessing List Elements:

You can access individual elements of a list using indexing. Python uses 0-based indexing, so the first element is at index 0, the second at index 1, and so on.

```
fruits = ["apple", "banana", "cherry"]
print(fruits[0])        # "apple"
print(fruits[1])        # "banana"
```

You can also use negative indexing to access elements from the end of the list:

```
print(fruits[-1])       # "cherry"
```

Modifying Lists:

Lists are mutable, which means you can change their contents. You can modify, add, or remove elements.

Modifying an element by assigning a new value to a specific index:

```
fruits = ["apple", "banana", "cherry"]
fruits[1] = "orange"
print(fruits)              # ["apple", "orange", "cherry"]
```

Adding elements with the append() method:

```
fruits = ["apple", "banana"]
fruits.append("cherry")
print(fruits)              # ["apple", "banana", "cherry"]
```

Inserting elements with the insert() method:

```
fruits = ["apple", "banana"]
fruits.insert(1, "cherry")
print(fruits)              # ["apple", "cherry", "banana"]
```

Removing elements with the remove() method:

```
fruits = ["apple", "banana", "cherry"]
fruits.remove("banana")
print(fruits)              # ["apple", "cherry"]
```

Removing elements by index with the pop() method:

```
fruits = ["apple", "banana", "cherry"]
removed_fruit = fruits.pop (1)
```

 # Removes and returns the second element

```
print(removed_fruit)   # "banana"
print(fruits)          # ["apple", "cherry"]
```

Slicing lists to extract a portion of the list:

```
numbers = [1, 2, 3, 4, 5]
subset = numbers[1:4]
print(subset)          # Returns a new list [2, 3, 4]
```

List Operations:

Concatenating lists with the + operator:

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]
result = list1 + list2     # [1, 2, 3, 4, 5, 6]
```

Repeating a list with the * operator:

```
fruits = ["apple"]
fruits *= 3                    # Repeats the list three times
# Result: ["apple", "apple", "apple"]
```

List Methods:

Python provides several built-in list methods for various operations, such as sorting, counting, and finding elements. Some common list methods include **sort()**, **len()**, **count()**, and **index()**. You can refer to Python's official documentation for a complete list of list methods.

Lists are versatile and widely used in Python for storing and manipulating collections of data. They are a fundamental data structure and play a crucial role in many programming tasks.

Tuples

A tuple is a versatile data structure that is similar to a list but with a key difference: tuples are immutable, which means their elements cannot be modified after creation. Tuples are often used to represent collections of related data, and they can contain elements of different data types. Here's how you can work with tuples:

Creating a Tuple:

To create a tuple, you enclose a comma-separated sequence of items within parentheses (). Unlike lists, tuples are fixed and cannot be modified once created.

```
coordinates = (3, 4)
rgb_color = (255, 0, 0)
mixed_tuple = ("apple", 2, 3.14, True)
```

Accessing Tuple Elements:

You can access individual elements of a tuple using indexing, just like with lists. Python uses 0-based indexing.

```
coordinates = (3, 4)
x = coordinates[0]             # x will be 3
y = coordinates[1]             # y will be 4
```

You can also use negative indexing to access elements from the end of the tuple:

```
last_element = coordinates[-1]          # last_element will be 4
```

Tuple Unpacking:

You can assign the elements of a tuple to multiple variables in a single line, a process known as tuple unpacking.

```
coordinates = (3, 4)
x, y = coordinates          # x will be 3    # y will be 4
```

Modifying Tuples:

As mentioned earlier, tuples are immutable. You cannot change the values of elements or add or remove elements from a tuple. Attempting to do so will result in an error.

Tuple Operations:

Concatenating tuples with the + operator:

```
tuple1 = (1, 2, 3)
tuple2 = (4, 5, 6)
result = tuple1 + tuple2        # (1, 2, 3, 4, 5, 6)
```

Repeating a tuple with the * operator:

```
fruits = ("apple",)
fruits *= 3                 # Repeats the tuple three times
print(fruits)               # Result: ("apple", "apple", "apple")
```

Tuple Methods:

Tuples are simple data structures, so they have fewer built-in methods compared to lists. Some common methods include **count()** to count the occurrences of a specific element and **index()** to find the index of a specific element within the tuple.

When to Use Tuples:

Tuples are preferred in situations where you want to create a collection of values that should not be changed. Some common use cases for tuples include:

- Representing fixed, unchangeable data (e.g., coordinates).
- Returning multiple values from a function.
- Using tuples as keys in dictionaries (since they are immutable) when you want to create custom data structures for lookups.

While tuples lack the flexibility of lists, their immutability can be an advantage in scenarios where you need to ensure that the data remains constant.

Dictionaries

A dictionary is a versatile and fundamental data structure used to store and manage collections of key-value pairs. Dictionaries are also known as associative arrays or hash maps. They are unordered and mutable, which means you can change their contents after creation. Here's how you can work with dictionaries:

Creating a Dictionary:

To create a dictionary, you enclose a comma-separated sequence of key-value pairs within curly braces {}. Each key is associated with a value, and the key-value pairs are separated by colons (:).

```
person = {"name": "ram", "age": 30, "city": "ayodhya"}
student = {"student_id": 12345, "name": "mohi", "grades": [90, 85, 92]}
```

Accessing Dictionary Elements:

You can access the values associated with keys using the square bracket notation. Provide the key within square brackets to retrieve the associated value.

```
person = {"name": "ram", "age": 30, "city": "ayodhya"}
name = person["name"]                    # name will be "ram"
age = person["age"]                      # age will be 30
```

You can also use the **get()** method to access dictionary values. This method allows you to provide a default value in case the key does not exist in the dictionary.

```
city = person.get("city", "Unknown City")     # city will be "ayodhya"
country = person.get("country", "Unknown Country")
```

 # country will be "Unknown Country"

Modifying Dictionaries:

Dictionaries are mutable, which means you can change their contents. You can add, update, or remove key-value pairs.

Adding a new key-value pair:

```
person = {"name": "anand", "age": 14}
person["city"] = "nainital"
```

Updating an existing value by specifying the key:

```
person["age"] = 31                # Updates the age from 30 to 31
```

Removing a key-value pair using the del statement:

```
del person["age"]                 # Removes the "age" key-value pair
```

Dictionary Operations:

Checking if a key exists in a dictionary using the in keyword:

```
person = {"name": "palak", "city": "kashipur"}
if "age" in person:
    print("Age exists in the dictionary")
else:
    print("Age does not exist in the dictionary")
```

Getting the number of key-value pairs in a dictionary using the len() function:

```
person = {"name": "lakshay", "age": 14, "city": "bajpur"}
count = len(person)                    # count will be 3
```

Copying a dictionary using the copy() method:

```
original_dict = {"name": "palak", "age": 11, "city": "bajpur"}
copy_dict = original_dict.copy()
```

Dictionary Methods:

Python provides several built-in methods for working with dictionaries. Some common dictionary methods include **keys()**, **values()**, and **items()** to retrieve keys, values, and key-value pairs, respectively.

Dictionaries are widely used in Python for various tasks, such as representing structured data, managing configurations, and creating efficient lookup tables. They are particularly useful when you need to associate keys with values and retrieve values based on their keys.

Sets

A set is an unordered and mutable collection of unique elements. Sets are often used for tasks that involve mathematical operations like union, intersection, and difference. They are represented using curly braces {} or the **set()** constructor. Here's how you can work with sets:

Creating a Set:

To create a set, you can use curly braces {} and enclose a comma-separated sequence of elements, or you can use the **set()** constructor with an iterable (e.g., a list).

```
fruits = {"apple", "banana", "cherry"}
numbers = set([1, 2, 3, 4, 5])
```

Accessing Set Elements:

Sets are unordered, so you cannot access elements by index. You can, however, check for the presence of an element using the in keyword.

```
fruits = {"apple", "banana", "cherry"}
is_apple_in_set = "apple" in fruits # True
```

Modifying Sets:

Sets are mutable, which means you can add and remove elements.

Adding elements using the add() method:

```
fruits = {"apple", "banana"}
fruits.add("cherry")
```

Removing elements using the remove() or discard() method:

```
fruits = {"apple", "banana", "cherry"}
fruits.remove("banana")
# or
fruits.discard("cherry")
```

If you attempt to remove an element that doesn't exist in the set using **remove()**, it will raise a KeyError. **discard()**, on the other hand, won't raise an error.

Set Operations:

Sets support various set operations like union, intersection, difference, and symmetric difference.

Union (combining two sets):

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
union_result = set1.union (set2) # {1, 2, 3, 4, 5}
```

Intersection (common elements between two sets):

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
intersection_result = set1.intersection (set2) # {3}
```

Difference (elements in one set but not the other):

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
difference_result = set1.difference (set2) # {1, 2}
```

Symmetric Difference (elements in either set but not both):

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
symmetric_difference_result = set1.symmetric_difference (set2) # {1, 2, 4, 5}
```

Set Methods:

Python provides various built-in methods for sets, such as **add()**, **remove()**, **discard()**, **union()**, **intersection()**, **difference()**, **symmetric_difference()**, **clear()**, **copy()**, and more.

Sets are useful for various tasks, including removing duplicates from a list, checking for membership, and performing set operations like union and intersection. They are particularly efficient for membership testing because they use a hash-based data structure, ensuring that lookup operations are fast even for large collections.

Strings

A string is a sequence of characters, and it is one of the most commonly used data types. Strings are immutable, which means their contents cannot be changed after creation. They are often used to represent textual data, whether it's a single word, a sentence, or an entire document. Here's how you can work with strings:

Creating a String:

To create a string, you can enclose a sequence of characters within single (') or double (") quotation marks.

```
name = "ram"
message = 'Hello, World!'
```

You can also create multi-line strings using triple-quotes (''' or """), which are useful for docstrings and multi-line text.

```
multi_line = '''
This is a multi-line
string in Python.
'''
```

Accessing String Characters:

You can access individual characters in a string using indexing. Python uses 0-based indexing, meaning the first character is at index 0, the second character at index 1, and so on.

```
name = "mohan"
first_char = name[0]                    # 'm'
second_char = name[1]                   # 'o'
```

You can also use negative indexing to access characters from the end of the string.

```
last_char = name[-1]                    # 'n'
second_last_char = name[-2]             # 'a'
```

String Slicing:

You can extract substrings from a string using slicing. Slicing allows you to specify a range of indices to create a new string that includes the characters within that range.

```
text = "Hello, World!"
substring = text[7:12]                  # 'World'
```

You can omit the start or end index to slice from the beginning or to the end of the string, respectively.

String Concatenation:

You can concatenate strings using the + operator.

```
greeting = "Hello"
name = "anand"
message = greeting + ", " + name        # 'Hello, anand'
```

String Methods:

Python provides many built-in methods for manipulating and working with strings. Some common string methods include **upper()**, **lower()**, **strip()**, **replace()**, and **split()**. These methods allow you to perform operations like converting a string to uppercase, removing leading and trailing whitespace, replacing substrings, and splitting a string into a list of substrings based on a delimiter.

```
text = " Hello, Meena! "
text_upper = text.upper()                       # ' HELLO, MEENA! '
text_stripped = text.strip()                    # 'Hello, Meena!'
text_replaced = text.replace("Meena", "Mahi")   # ' Hello, Mahi! '
words = text.split(",")                         # [' Hello', ' Meena! ']
```

String Formatting:

You can format strings using different methods, including f-strings, string interpolation with **%**, and the **format()** method. F-strings are a common and modern way to format strings in Python.

```
name = "Palak"
age = 11
formatted_string = f"My name is {name} and I am {age} years old."
```

String Escapes:

Special characters in strings can be represented using escape sequences. For example, **"\n"** represents a newline character, and **"\t"** represents a tab character.

```
escaped_string = "This is a line with a newline\nand a tab\tcharacter."
```

Python provides a wide range of string methods and powerful string manipulation capabilities, making it easy to work with textual data in your programs. Strings are essential for tasks like text processing, data input and output, and building user interfaces.

Chapter 4
Object-Oriented Programming (OOP)

Classes and Objects

In object-oriented programming (OOP), classes and objects are fundamental concepts. They allow you to model and organize your code in a way that mimics the real world, making it more modular and maintainable. Here's an overview of classes and objects in Python:

1. Classes:

A class is a blueprint or template for creating objects. It defines the structure and behavior of objects of that type. Classes in Python are created using the class keyword. A class can include attributes (data members) and methods (functions). Here's how you define a simple class in Python:

```
class Dog:
def __init__(self, name, breed):
    self.name = name
self.breed = breed
    def bark(self):
        return "Woof!"
    def describe(self):
        return f"{self.name} is a {self.breed} dog."
```

- The **__init__** method is the **constructor**, which is called when an object is created. It initializes the attributes of the object.
- The **bark** and **describe** methods are functions associated with the class. They can be called on objects of the class.

2. Objects:

An object is an instance of a class. It represents a concrete, specific entity based on the class's blueprint. You create objects by calling the class as if it were a function. Here's how you create and work with objects of the Dog class:

```
dog1 = Dog("Chitti", "Pit Bull")
dog2 = Dog("Max", "German Shepherd")
print(dog1.name)       # "Chitti"
print(dog2.describe()) # "Max is a German Shepherd dog."
print(dog2.bark())     # "Woof!"
```

In the code above, **dog1** and **dog2** are objects created from the Dog class. You can access their attributes and call their methods as above.

Inheritance and Polymorphism

Inheritance and polymorphism are two key concepts in object-oriented programming (OOP) that enable code reusability, extensibility, and flexibility. They allow you to create hierarchies of classes and use objects of derived classes in a way that's compatible with their base classes. Let's delve deeper into these concepts in Python:

1. Inheritance:

Inheritance is the mechanism by which one class can inherit properties (attributes and methods) from another class. The class from which properties are inherited is known as the base class or parent class, and the class that inherits those properties is known as the derived class or child class.

In Python, you can create a derived class by defining a new class that inherits from a base class using the following syntax:

```
class BaseClass:
# Base class attributes and methods
class DerivedClass(BaseClass):
# Additional attributes and methods specific to the derived class
```

Here's an example:

```
class Animal:
    def __init__(self, species):
self.species = species
class Bird(Animal):
    def fly(self):
        return f"A {self.species} is flying."
```

In this example, the Bird class inherits the species attribute from the Animal class and adds its own fly method.

2. Polymorphism:

Polymorphism is a fundamental OOP concept that allows objects of different classes to be treated as objects of a common base class. This enables you to write more flexible and generic code. Polymorphism is achieved through method overriding and interfaces.

- **Method Overriding:**

Method overriding allows a derived class to provide a specific implementation for a method that is already defined in the base class. When a method is called on an object, the appropriate version of the method is executed based on the object's actual class. This is also known as dynamic method dispatch.

```
class Animal:
    def speak(self):
        return "Some generic animal sound."
class Dog(Animal):
    def speak(self):
        return "Woof!"
class Cat(Animal):
    def speak(self):
        return "Meow!"
# Polymorphism in action
animals = [Dog(), Cat()]
for animal in animals:
    print(animal.speak())
```

In the example above, we have a list of Animal objects that can include both Dog and Cat objects. When the speak method is called on each object, the appropriate overridden version of the method is executed.

- **Interfaces and Abstract Classes:**

In Python, you can create interfaces and abstract classes using the **"abc"** module. These classes define method signatures that must be implemented by any concrete (non-abstract) class that inherits from them. This enforces a certain structure and behavior for classes that implement the interface.

Here's a basic example:

```
from abc import ABC, abstractmethod
class Shape(ABC):
    @abstractmethod
    def area(self):
        pass
class Circle(Shape):
    def __init__(self, radius):
self.radius = radius
    def area(self):
        return 3.14 * self.radius * self.radius
```

In this example, the Shape class defines an abstract method area, which must be implemented by any concrete class that inherits from it.

In summary, inheritance and polymorphism are essential OOP concepts that help you create reusable and extensible code. Inheritance allows you to create hierarchies of classes with shared attributes and methods, while polymorphism allows you to treat objects of different classes in a uniform way, making your code more flexible and easier to maintain.

Encapsulation and Abstraction

Encapsulation and abstraction are two important principles in object-oriented programming (OOP) that help manage complexity, enhance code organization, and improve maintainability. They allow you to hide the internal details of a class while exposing a well-defined and easy-to-use interface. Let's explore these concepts in more detail:

1. Encapsulation:

Encapsulation is the concept of restricting access to certain components of an object or a class while exposing a well-defined interface. It helps in preventing unintended interference and ensures that data and behavior are encapsulated within a class. In Python, encapsulation can be achieved using access modifiers and property methods.

- **Access Modifiers:**

In Python, there are three access modifiers to control the visibility of attributes and methods:

Public (default): Members are accessible from anywhere.

Protected (underscore prefix _): Members are not intended for public use but can be accessed from outside the class.

Private (double underscore prefix __): Members are not accessible from outside the class.

Here's an example of encapsulation using access modifiers:

```
class Student:
    def __init__(self, name, age):
        self.name = name        # Public attribute
self._age = age             # Protected attribute
self.__grade = 'A'          # Private attribute
    def display(self):
        print(f"Name: {self.name}, Age: {self._age}, Grade: de}")
```

```
student = Student("ram", 25)
print(student.name)    # Accessing the public attribute
print(student._age)    # Accessing the protected attribute (not recommended)
print(student.__grade) # Accessing the private attribute will raise an AttributeError
```

Property Methods:

Property methods (getter and setter methods) allow you to control the access to an attribute and perform custom actions when getting or setting its value. This can be used to maintain data integrity and hide the implementation details.

```
class Circle:
    def __init__(self, radius):
self._radius = radius
    @property
    def radius(self):
        return self._radius
    @radius.setter
    def radius(self, value):
        if value < 0:
            raise ValueError("Radius cannot be negative.")
self._radius = value
circle = Circle(5)
print(circle.radius)      # Accessing the property
circle.radius = 10        # Setting the property (value checked by the setter)
```

2. Abstraction:

Abstraction is the process of simplifying complex systems by breaking them into smaller, more manageable parts while hiding unnecessary details. It focuses on defining a clear and concise interface to interact with an object, without exposing the internal implementation.

In Python, you can achieve abstraction by defining abstract classes and methods using the **"abc"**(Abstract Base Classes) module. Abstract classes cannot be instantiated, and they define abstract methods that must be implemented by concrete (non-abstract) subclasses.

Here's an example:

```
from abc import ABC, abstractmethod
class Shape(ABC):
    @abstractmethod
    def area(self):
        pass
class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius
    def area(self):
        return 3.14 * self.radius * self.radius
circle = Circle(5)
print(circle.area()) # Concrete implementation of the abstract method
```

In this example, the Shape class is an abstract class with an abstract method area. The Circle class is a concrete subclass that provides an implementation of the area method.

Abstraction allows you to define a clear and consistent interface for your classes, promoting code reuse and making your code more understandable and maintainable.

Both encapsulation and abstraction are crucial in OOP to create clean, modular, and maintainable code. Encapsulation helps protect the internal state of objects, while abstraction provides a well-defined and simplified interface to interact with those objects.

Exception Handling

Exception handling is a crucial aspect of programming that allows you to gracefully manage and recover from unexpected or exceptional situations that may occur during the execution of your code. In Python, exceptions are raised when an error or unusual event occurs, and you can use exception handling to handle these events. Here's an overview of how exception handling works in Python:

1. Basic Exception Handling:

In Python, exceptions are raised when something goes wrong during program execution. You can use the try, except, and optionally finally blocks to handle exceptions. Here's a basic example:

```
try:
# Code that might raise an exception
    x = 10 / 0              # this will raise a ZeroDivisionError
except ZeroDivisionError:
# Code to handle the exception
    print("Division by zero is not allowed.")
```

In this example, the code inside the try block raises a **ZeroDivisionError**, and **except** block handles it by printing an error message.

2. Handling Multiple Exceptions:

You can handle multiple exceptions by including multiple except blocks. This allows you to handle different types of exceptions differently.

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except ValueError:
    print("Invalid input. Please enter a valid number.")
except ZeroDivisionError:
    print("Division by zero is not allowed.")
```

3. The else Block:

You can use an else block after all except blocks to specify code that should be executed when no exceptions are raised.

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except ValueError:
    print("Invalid input. Please enter a valid number.")
except ZeroDivisionError:
    print("Division by zero is not allowed.")
else:
    print("Result is:", result)
```

4. The finally Block:

The finally block is used to define code that will be executed regardless of whether an exception is raised or not. This block is commonly used for cleanup operations, such as closing files or releasing resources.

```
try:
    file = open("example.txt", "r")
    # Perform file operations
except FileNotFoundError:
    print("File not found.")
finally:
file.close() # Always close the file, whether an exception raised or not
```

5. Custom Exceptions:

You can create your own custom exceptions by defining new classes that inherit from the base Exception class. This allows you to raise and catch specific exceptions tailored to your application's needs.

```
class CustomError(Exception):
    def __init__(self, message):
        super().__init__(message)
try:
    if some_condition:
        raise CustomError("This is a custom exception.")
except CustomError as e:
    print(e)
```

6. Handling Exceptions in a Function:

You can also handle exceptions within a function and propagate the exception to the caller or perform specific error handling within the function.

```
def divide(x, y):
    try:
        result = x / y
    except ZeroDivisionError:
        return "Division by zero is not allowed."
    return result
result = divide(10, 0)
print(result)
```

Exception handling is a critical aspect of writing robust and reliable code. It allows you to gracefully handle errors and unexpected situations, making your code more resilient and user-friendly. When designing your code, consider the types of exceptions that may occur and plan your exception handling strategy accordingly.

Chapter 5
File Handling

Reading and Writing Files

Reading and writing files is a common task in Python, and it allows you to interact with external data sources, like text files, CSV files, JSON files, and more. Here, I'll provide an overview of how to read and write text files in Python.

1. Reading Files:
You can read the contents of a text file in Python using the **open()** function in combination with different modes, such as **'r'** mode for reading. **Here's a basic example:**

```
# Opening a file in read mode
with open('example.txt', 'r') as file:
    content = file.read()
# Printing the contents of the file
print(content)
```

In the code above, **with** statement is used to open the file and automatically close it when the block is exited. The file's contents are then read into the content variable.

2. Writing Files:
To write data to a file, you can use the **'w'mode** with the open() function. If the file doesn't exist, it will be created. If it does exist, the content will be overwritten. **Here's a simple example:**

```
# Opening a file in write mode
with open('output.txt', 'w') as file:
file.write("Hello, world!\n")
file.write("This is a new line of text.")
```

In this example, we open 'output.txt' in write mode and write some text to it. If the file already exists, its previous content will be overwritten.

3. Appending to Files:
You can append data to an existing file using the **'a' (append) mode**. This will add new content to the end of the file without overwriting the existing content.

```
# Opening a file in append mode
with open('output.txt', 'a') as file:
file.write("\nThis is an appended line.")
```

4. Reading Line by Line:

If you need to process a file line by line, you can use for loop: with open('example.txt', 'r') as file:

```
for line in file:
    print(line)
```

This code reads the file line by line, making it useful for processing large files without loading the entire content into memory.

5. Handling Exceptions:

When working with files, it's important to handle exceptions, such as **FileNotFoundError** or **PermissionError**. You can use a try and except block to handle these exceptions gracefully.

```
try:
    with open('nonexistent_file.txt', 'r') as file:
        content = file.read()
    except FileNotFoundError:
        print("The file does not exist.")
```

This code attempts to open a file that doesn't exist, and the exception is caught and handled.

6. Closing Files Explicit

While using the with statement is recommended because it automatically closes the file, you can also close a file explicitly using the **close()** method:

```
file = open('example.txt', 'r')
content = file.read()
file.close()                    # Close the file explicitly
```

It's important to close files when you're done with them to free up system resources.

These are the basics of reading and writing text files in Python. You can also work with other types of files, such as CSV files, JSON files, and more, using appropriate libraries and methods tailored to those file formats

Working with CSV and JSON

Working with CSV (Comma-Separated Values) and JSON (JavaScript Object Notation) are common tasks in data processing and exchange. Python provides built-in modules to simplify reading and writing data in these formats. Let's explore how to work with CSV and JSON in Python:

Working with CSV:

1. Reading CSV Files:

To read data from a CSV file, you can use the csv module. Here's an example of reading a CSV file and printing its contents:

```
import csv
with open('data.csv', 'r') as file:
    csv_reader = csv.reader(file)
        for row in csv_reader:
                print(row)
```

You can also specify a delimiter other than a comma by passing the delimiter parameter to the **csv.reader()** function.

2. Writing to CSV Files:

To write data to a CSV file, you can use the csv.writer class. Here's an example of writing data to a CSV file:

```
import csv
data = [
    ["Name", "Age"],
    ["ram", 25],
    ["mohan", 30]
]
with open('output.csv', 'w', newline='') as file:
    csv_writer = csv.writer(file)
    csv_writer.writerows(data)
```

3. Using Dictionaries for CSV:

You can also read and write CSV files using dictionaries, which is a common format for tabular data. This is achieved using **csv.DictReader** and **csv.DictWriter**:

```
import csv
with open('data.csv', 'r') as file:
    csv_reader = csv.DictReader(file)
    for row in csv_reader:
        print(row)
data = [
{"Name": "palak", "Age": 12},
{"Name": "lakshay", "Age": 14}
]
with open('output.csv', 'w', newline='') as file:
    fieldnames = data[0].keys()
    csv_writer = csv.DictWriter(file, fieldnames=fieldnames)
    csv_writer.writeheader()
    csv_writer.writerows(data)
```

Working with JSON:

1. Reading JSON Files:

To read data from a JSON file, you can use the json module:

```
import json
with open('data.json', 'r') as file:
    data = json.load(file)
    print(data)
```

2. Writing to JSON Files:

To write data to a JSON file, you can use the **json.dump()** method:

```
import json
data = {
    "name": "ram",
    "age": 25
}
with open('output.json', 'w') as file:
    json.dump(data, file)
```

If you want to make the JSON file more human-readable, you can specify the indent parameter:

```
json.dump(data, file, indent=4)
```

3. Using JSON with Python Dictionaries:
In Python, dictionaries are similar in structure to JSON objects. You can easily convert between JSON and Python dictionaries using the **json** module:

```
import json
data = {
    "name": "anand",
    "age": 25
}
# Convert Python dictionary to JSON
json_data = json.dumps(data)
# Convert JSON to Python dictionary
python_data = json.loads(json_data)
```

Working with CSV and JSON data is important for data manipulation, file parsing, and data exchange. These formats are widely used for various data storage and interchange purposes, and Python provides convenient libraries to handle them efficiently.

Error Handling in File Operations

Error handling is crucial when working with file operations in Python. You need to anticipate and handle potential errors to ensure the robustness and reliability of your code. Here are some common errors you may encounter when performing file operations and how to handle them:

1. FileNotFoundError:
This error occurs when you attempt to open or manipulate a file that does not exist. You can handle it as follows:

```
try:
    with open('nonexistent_file.txt', 'r') as file:
        content = file.read()
except FileNotFoundError:
    print("The file does not exist.")
```

2. PermissionError:
This error occurs when you do not have the required permissions to access or modify a file. You can handle it like this:

```
try:
    with open('/root/protected_file.txt', 'w') as file:
        file.write("This is a protected file.")
```

```
except PermissionError:
print("Permission denied. You do not have access to this file.")
```

3. IOError (Input/Output Error):

This is a more general error that can occur for various reasons, such as attempting to open a directory as a file or trying to read a write-only file. You can handle it as follows:

```
try:
    with open('/dev/sda', 'r') as file:
        content = file.read()
except IOError as e:
    print(f"An IO error occurred: {e}")
```

4. Handling Multiple Errors:

You can handle multiple errors by including multiple **except** blocks, each specifically targeting a different error type. Here's an example:

```
try:
    with open('file.txt', 'r') as file:
        content = file.read()
except FileNotFoundError:
    print("The file does not exist.")
except PermissionError:
    print("Permission denied.")
except IOError as e:
    print(f"An IO error occurred: {e}")
```

5. Using finally Block:

You can use the finally block to execute cleanup operations regardless of whether an exception is raised or not. This is often used for closing files and releasing resources:

```
try:
    with open('file.txt', 'r') as file:
        content = file.read()
except FileNotFoundError:
    print("The file does not exist.")
finally:
file.close()            # Close the file, even if an error occurred
```

6. Using else Block:

The else block can be used to specify code that should be executed when no exceptions are raised. For example:

```
try:
    with open('file.txt', 'r') as file:
        content = file.read()
except FileNotFoundError:
    print("The file does not exist.")
else:
    print("File reading was successful.")
```

Handling errors in file operations is essential for making your code robust and user-friendly. It allows your program to respond gracefully to unexpected situations, ensuring that your code remains reliable and error-tolerant.

Chapter 6
Advanced
Python Topics

Decorators and Generators

Decorators:
Decorators are a powerful and flexible feature in Python that allow you to modify or enhance the behavior of functions or methods without changing their code. Decorators are often used for tasks like logging, authentication, authorization, and code profiling.
Here's a simple example of a decorator:

```
def my_decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
    def func()
        print("Something is happening after the function is called.")
    return wrapper

@my_decorator
def say_hello():
    print("Hello!")
say_hello()
```

In this example, my_decorator is a decorator function that takes another **function (func)** as its argument and returns a **new function (wrapper)** that wraps the original function. When you use the **@** symbol with the decorator name before a function definition, it indicates that the function should be decorated. In this case, **say_hello** is wrapped by **my_decorator**, which adds some behavior before and after the original function.
Generators:
Generators are a way to create iterators in Python. Unlike traditional functions that use return, generators use yield to produce a sequence of values lazily, one at a time. This is especially useful for working with large data sets, as it doesn't require storing the entire data in memory.
Here's a simple example of a generator:

```
def count_up_to(n):
i = 1
    while i<= n:
        yield i
i += 1
for num in count_up_to(5):
    print(num)
```

In this example, count_up_to is a generator function that yields numbers from 1 to n. When you iterate over it using a for loop, the numbers are produced one at a time.

Generators can also be used to create infinite sequences or to efficiently process large data sets without consuming excessive memory.

Decorators and generators are powerful and versatile features in Python that can significantly enhance the **readability, reusability, and performance** of your code. **Decorators** are commonly used for **cross-cutting concerns like logging and authentication**, while **generators** are ideal for lazy evaluation and dealing with **large data sets**.

Context Managers

Context managers in Python are a convenient way to manage resources, such as files, network connections, or database connections, ensuring that they are properly acquired and released, even if exceptions occur during their use. Context managers are typically used with the **with statement**, which simplifies the setup and teardown of resources. Python's standard library provides the **contextlib module** to create **custom context managers**, and it also includes some **built-in context managers**.

Here's how to use context managers with the with statement:

1. Using Built-In Context Managers:

Python provides built-in context managers for common tasks, like opening and closing files or dealing with network connections. For example, when working with files, you can use the **open()** function as a context manager:

```
with open('example.txt', 'r') as file:
    content = file.read()
# File is automatically closed when exiting the block
# File is already closed here
```

In this example, the **with** statement ensures that the file is properly closed when the block is exited, even if an exception occurs.

2. Creating Custom Context Managers:
You can create your own context managers by defining classes with (**__enter__**) and**(__exit__) methods**. The **__enter__** method is responsible for resource setup, and the **__exit__** method is responsible for resource cleanup. Here's an example:

```
class MyContextManager:
    def __enter__(self):
        print("Entering the context")
# Resource setup (e.g., open a file)
        return self
# Optionally return an object to be used within the context
    def __exit__(self, exc_type, exc_value, traceback):
        print("Exiting the context")
# Resource cleanup (e.g., close a file)
with MyContextManager() as cm:
    print("Inside the context")
# Exiting the context
```

In this example, when the **with block** is entered, the **__enter__** method is called, and when the block is exited, the **__exit__** method is called, ensuring proper resource management.

3. Using contextlib for Simpler Context Managers:
The contextlib module in Python's standard library provides tools to create context managers more easily, especially for simpler cases. You can use the **contextlib.contextmanager** decorator to define a generator-based context manager. **Here's an example:**

```
from contextlib import contextmanager
@contextmanager
def my_context_manager():
    print("Entering the context")
# Resource setup
    yield #The control is yielded to the with block
    print("Exiting the context")
# Resource cleanup
with my_context_manager():
    print("Inside the context")
# Exiting the context
```

In this example, the yield statement serves as the point where the control is temporarily transferred to the **with block**. When the block is exited, **execution continues after the yield statement**.

Context managers help ensure **resource management, clean code,anderror handling**. They are widely used in Python to handle tasks like file I/O, database connections, and network operations. When you use context managers, you can be confident that resources will be acquired and released correctly, making your code more robust and maintainable.

Multithreading and Multiprocessing

Multithreading and multiprocessing are techniques in Python for concurrent execution of code, which can help improve the performance of your applications, particularly when dealing with CPU-bound or I/O-bound tasks. These techniques enable you to execute multiple tasks in parallel, taking advantage of multi-core processors. Here's an overview of both concepts:

Multithreading:

Multithreading involves using multiple threads within a single process to perform tasks concurrently. Python's threading module is used for this purpose. However, due to Python's Global Interpreter Lock (GIL), multithreading is generally not suitable for CPU-bound tasks (tasks that require significant processing power) in Python. It's more effective for I/O-bound tasks where threads spend time waiting for I/O operations to complete.

Here's a simple example using the threading module:

```
import threading
def print_numbers():
    for i in range(1, 6):
        print(f"Number: {i}")

def print_letters():
    for letter in 'abcde':
        print(f"Letter: {letter}")
# Create two threads
t1 = threading.Thread(target=print_numbers)
t2 = threading.Thread(target=print_letters)
```

```
# Start the threads
t1.start()
t2.start()
# Wait for both threads to finish
t1.join()
t2.join()
print("Both threads are done.")
```

In this example, two threads are created to print numbers and letters concurrently.

Multiprocessing:

Multiprocessing involves using multiple processes, each with its own Python interpreter and memory space, to execute tasks in parallel. Python's multiprocessing module is used for this purpose. Unlike multithreading, multiprocessing can fully utilize multiple CPU cores and is suitable for CPU-bound tasks.

Here's a simple example using the multiprocessing module:

```
import multiprocessing
def square(number, result, index):
    result[index] = number * number
if __name__ == '__main__':
    numbers = [1, 2, 3, 4, 5]
    results = multiprocessing.Array('i', len(numbers))
    processes = []
    for i, number in enumerate(numbers):
        process = multiprocessing.Process(target=square,
                    args=(number, results, i))
        processes.append(process)
        process.start()
    for process in processes:
        process.join()
        print(list(results))
```

In this example, we use multiple processes to calculate the squares of numbers in parallel and store the results in an array.

Key Differences:

- Multithreading uses multiple threads within a single process, while multiprocessing uses multiple separate processes.
- Multithreading is subject to Python's **Global Interpreter Lock (GIL)**, which can limit its effectiveness for CPU-bound tasks. Multi-processing bypasses the GIL and can utilize multiple CPU cores.
- Multiprocessing requires inter-process communication mechanisms for sharing data between processes, such as multiprocessing.Queue and multiprocessing.Array.

When choosing between multithreading and multiprocessing, consider the nature of your task. If you have CPU-bound tasks that can benefit from parallel processing, consider using multiprocessing. For I/O-bound tasks, multithreading may be more appropriate. Keep in mind that Python's GIL limits the effectiveness of multithreading for certain scenarios, so multiprocessing is often the preferred choice for CPU-bound tasks.

Regular Expressions

Regular expressions, often abbreviated as regex or regexp, are a powerful tool for pattern matching and text manipulation. They provide a concise and flexible way to search, match, and manipulate text strings based on patterns. In Python, the **re module** is used to work with regular expressions.

Here are some key concepts and examples of using regular expressions in Python:

1. Basic Patterns:

. (dot): Matches any single character except a newline.

^: Anchors the regex at the start of the string.

$: Anchors the regex at the end of the string.

***:** Matches 0 or more occurrences of the preceding character.

+: Matches 1 or more occurrences of the preceding character.

?: Matches 0 or 1 occurrence of the preceding character.

[]: Matches any single character within the brackets.

|: Acts like an OR operator.

2. Using the re Module:

```
import re
pattern = r'^[A-Za-z]+$'
text = 'HelloWorld'
# Check if the entire string consists of letters only
if re.match(pattern, text):
    print(f"{text} matches the pattern.")
else:
    print(f"{text} does not match the pattern.")
```

3. Matching and Searching:

```
import re
pattern = r'\b\d{4}-\d{4}-\d{4}\b' # Matches a aadhar number pattern
text = 'ram: 1234-4578-3123, mohan: 9873-6534-4323'
matches = re.findall(pattern, text)
print(matches)
```

4. Groups and Capture:

```
import re
pattern = r'(\w+): (\d+)-(\d+)-(\d+)' # Matches name and aadhar number pattern
text = 'ram: 1233-1245-6789, mohan: 9887-6765-4321'
matches = re.findall(pattern, text)
for match in matches:
    name, part1, part2, part3 = match
    print(f"{name}'s aadhar: {part1}-{part2}-{part3}")
```

5. Substitution:

```
import re
pattern = r'\b(\w)(\w*)\b' # Matches individual words
text = 'regular expressions are powerful'
# Capitalize the first letter of each word
result = re.sub(pattern, lambda match: match.group(1).upper() +
        match.group(2), text)
print(result)
```

These are just a few examples of what you can do with regular expressions in Python. Regular expressions are a vast and powerful topic, and they can be used for tasks such as **validation**, **searchandreplace**, **text extraction**, **and more**. If you're new to regular expressions, it may take some time to become familiar with the syntax and techniques, but they are a valuable tool for working with text data.

Working with Dates and Times

Working with dates and times in Python is facilitated by the **datetime module**, which provides classes for working with **dates**, **times**, **and timedeltas**. Here's an overview of how to work with dates and times in Python:

1. Current Date and Time:
You can obtain the current date and time using the datetime class:

```
from datetime import datetime
current_datetime = datetime.now()
print("Current Date and Time:", current_datetime)
```

2. Formatting Dates and Times:
You can format dates and times as strings using the **strftime method**, which stands for **"string format time"**:

```
from datetime import datetime
current_datetime = datetime.now()
# Format as string
formatted_date = current_datetime.strftime("%Y-%m-%d %H:%M:%S")
print("Formatted Date:", formatted_date)
```

The format codes used in the **strftime** method are placeholders for various components like year, month, day, hour, minute, and second.

3. Parsing Strings to Dates:
You can parse strings to obtain datetime objects using the **strptime method**:

```
from datetime import datetime
date_string = "2023-11-09 12:30:00"
```

```
parsed_date = datetime.strptime(date_string, "%Y-%m-%d %H:%M:%S")
print("Parsed Date:", parsed_date)
```

Make sure to provide the correct format code corresponding to the structure of your date string.

4. Time Delta:

A timedelta represents the difference between two dates or times:

```
from datetime import datetime, timedelta
current_datetime = datetime.now()
future_datetime = current_datetime + timedelta(days=7)
print("Current Date and Time:", current_datetime)
print("Future Date and Time:", future_datetime)
```

In this example, a timedelta of 7 days is added to the current date and time.

5. Working with Time Zones:

For working with time zones, you can use the **pytz library**:

```
from datetime import datetime
import pytz
# Set the time zone
tz = pytz.timezone('America/New_York')
current_datetime = datetime.now(tz)
print("Current Date and Time in New York:", current_datetime)
```

6. Arithmetic with Dates:

You can perform arithmetic operations with dates, such as finding the difference between two dates:

```
from datetime import datetime
date1 = datetime(2023, 11, 9)
date2 = datetime(2023, 11, 1)
difference = date1 - date2
print("Difference in Days:", difference.days)
```

This calculates the difference in days between date1 and date2.

Working with dates and times can involve complex scenarios, such as handling daylight saving time, leap years, and different calendar systems. The datetime module in Python provides a solid foundation for these tasks, and additional libraries like pytz can enhance your capabilities, especially when dealing with time zones.

Virtual Environment

Virtual environments in Python are a way to create isolated environments for your projects, allowing you to manage dependencies and avoid conflicts between different projects. The **venv module** is the built-in tool for creating virtual environments in Python 3.3 and newer versions.

Here's a basic guide on working with virtual environments:

1. Creating a Virtual Environment:

To create a virtual environment, open a terminal or command prompt and navigate to your project's directory. Then, run the following command:

python -m venv venv

This command creates a virtual environment named **"venv"** in your project directory.

2. Activating the Virtual Environment:

After creating the virtual environment, you need to activate it. On Windows, use:

venv\Scripts\activate

On macOS and Linux, use:

source venv/bin/activate

When the virtual environment is activated, your command prompt or terminal prompt will change, indicating that you are now working within the virtual environment.

3. Installing Dependencies:

With the virtual environment activated, you can install dependencies specific to your project. For example:

pip install package_name

This installs the package only in the virtual environment, keeping your global Python environment clean.

4. Deactivating the Virtual Environment:

When you're done working in the virtual environment, you can deactivate it using the following command:

deactivate

5. Using requirements.txt:

You can create a **requirements.txt** file to specify the dependencies for your project. It helps in sharing and replicating your environment. To generate a requirements.txt file, use:

pip freeze > requirements.txt

To install dependencies from a requirements.txt file, use:

pip install -r requirements.txt

6. Virtual Environment Best Practices:

- Always use virtual environments for your projects to avoid conflicts between dependencies.
- Include the venv directory in your project's .gitignore or equivalent file to avoid versioning the virtual environment.
- Share your requirements.txt file with your project so others can easily recreate the environment.

Using virtual environments is a best practice in Python development, especially when working on multiple projects or collaborating with others. It ensures that each project has its own isolated environment, preventing dependency clashes and making it easier to manage project-specific requirements.

Chapter 7
Python
Standard Library

The Python Standard Library is a collection of modules and packages that come with the Python programming language. These modules provide a wide range of functionality, from working with data types and structures to handling networking, file I/O, and much more. Here are some key categories and examples of modules from the Python Standard Library:

1. **Data Types and Structures:**
- **collections:** Provides alternatives to built-in types like lists and dictionaries, such as `**Counter**`, `**defaultdict**`, and `**namedtuple**`.
- **json:** Enables encoding and decoding JSON data.
- **math:** Offers mathematical functions and constants.
- **random:** Generates random numbers and performs random selections.

2. **File and Directory Access:**
- **os:** Provides a way to interact with the operating system, including file and directory operations.
- **shutil:** Offers higher-level file operations, such as copying and archiving.
- **glob:** Helps find files using wildcard patterns.

3. **Networking:**
- **socket:** Implements low-level network communication.
- **http.server and socketserver:** Facilitate building simple HTTP servers.
- **urllib:** Allows working with URLs.

4. **Threading and Multiprocessing:**
- **threading and multiprocessing:** Support concurrent programming using threads and processes.
- **queue:** Provides thread-safe FIFO queues for communication between threads.

5. **Time and Date:**
- **datetime:** Offers classes for working with dates and times.
- **time:** provides functions for working with time, such as measuring execution time.

6. **Regular Expressions:**
- **re:** Implements regular expression operations.

7. **Testing:**
- **unittest:** The built-in testing framework for writing and running tests.

8. **Web and Internet Data:**
- **urllib.request:** Fetches data from URLs.
- **http.client:** Implements an HTTP client.

9. **Compression and Archiving:**
- **zipfile:** Provides tools to create, read, write, append, and list a ZIP file.
- **tarfile:** Allows working with tar archive files.

10. **Cryptography:**
- **hashlib:** Implements hash functions.
- **ssl:** Supports TLS/SSL protocols for secure network communication.

11. **Data Serialization:**
- **pickle:** Serializes and deserializes Python objects.
- **json:** Encodes and decodes JSON data.

12. **Command-Line Argument Parsing:**
- **argparse:** Helps parse command-line arguments.

13. **Miscellaneous:**
- **platform:** Provides an interface to interact with the underlying platform's identifying data.
- **logging:** Implements a flexible logging system.

These are just a few examples of the modules available in the Python Standard Library. The standard library is extensive, covering a broad range of topics and providing tools for various programming tasks. When working on a project, it's beneficial to explore the standard library to leverage existing functionality and reduce the need for external dependencies.

Commonly Used Modules (e.g., os, sys, math, random)

The Python Standard Library is a collection of modules and packages that come with the Python programming language. These modules provide a wide range of functionality, from working with data types and structures to handling networking, file I/O, and much more. Here are some key categories and examples of modules from the Python Standard Library:

1. os (Operating System Interface):

Purpose: Provides a way to interact with the operating system, allowing you to perform tasks like file and directory operations.

Common Functions:

os.getcwd(): Get the current working directory.

os.listdir(): List files and directories in a given path.

os.path.join(): Join one or more path components intelligently.

```
import os
# Get the current working directory
current_dir = os.getcwd()
print("Current Directory:", current_dir)
# List files and directories in the current directory
file_list = os.listdir(current_dir)
print("Files and Directories:", file_list)
```

2. sys (System-Specific Parameters and Functions):

Purpose: Provides access to some variables used or maintained by the interpreter and functions that interact strongly with the interpreter.

Common Functions:

sys.argv: List of command-line arguments.

sys.exit(): Terminate the program.

sys.path: List of directories where Python looks for modules.

```
import sys
# Print command-line arguments
print("Command-line arguments:", sys.argv)
# Exit the program with a message
sys.exit("Exiting the program.")
```

3. math (Mathematical Functions):

Purpose: Provides mathematical functions and constants.

Common Functions:

math.sqrt(x): Return the square root of x.

math.sin(x), math.cos(x), math.tan(x): Trigonometric functions.

math.pi: A mathematical constant representing Pi.

```
import math
# Calculate the square root
square_root = math.sqrt(25)
print("Square Root:", square_root)
# Calculate the sine of an angle
angle_sin = math.sin(math.radians(30))
print("Sine of 30 degrees:", angle_sin)
```

4. random (Random Number Generators):

Purpose: Provides functions for generating pseudo-random numbers.

Common Functions:

random.random(): Return the next random floating-point number in the range [0.0, 1.0).

random.randint(a, b): Return a random integer N such that a <= N <= b.

random.choice(seq): Return a random element from the non-empty sequence.

```
import random
# Generate a random number between 0 and 1
random_number = random.random()
print("Random Number:", random_number)
# Generate a random integer between 1 and 10 (inclusive)
random_integer = random.randint(1, 10)
print("Random Integer:", random_integer)
# Choose a random element from a list
fruits = ['apple', 'orange', 'banana', 'grape']
random_fruit = random.choice(fruits)
print("Random Fruit:", random_fruit)
```

5. datetime (Date and Time):
Purpose: Supplies classes for working with dates and times.
Common Classes:
datetime.datetime: Represents a date and time.
datetime.date: Represents a date without time.
datetime.time: Represents a time without date.
Common Functions:
datetime.now(): Returns the current local date and time.
datetime.strptime(date_string, format): Parses a string representing a date and time.

```
from datetime import datetime
# Get the current date and time
current_datetime = datetime.now()
print("Current Date and Time:", current_datetime)
# Parse a date string
date_string = "2023-11-09"
parsed_date = datetime.strptime(date_string, "%Y-%m-%d")
print("Parsed Date:", parsed_date)
```

6. json (JSON Encoding and Decoding):
Purpose: Provides methods for encoding and decoding JSON data.
Common Functions:
json.dumps(obj): Serialize obj to a JSON formatted str.
json.loads(s): Deserialize s (a str, bytes, or bytearray instance).

```
import json
# Create a dictionary
data = {'name': 'John', 'age': 30, 'city': 'New York'}
# Convert the dictionary to a JSON string
json_string = json.dumps(data)
print("JSON String:", json_string)
# Parse the JSON string back to a dictionary
parsed_data = json.loads(json_string)
print("Parsed Data:", parsed_data)
```

7. subprocess (Subprocess Management):
Purpose: Allows the spawning of additional processes and provides interfaces for communicating with them.

Common Functions:

subprocess.run(command, ...): Run the command with arguments.

```
import subprocess
# Example 1: Run a Shell Command and Capture Output
command = "ls -l"
# Run the command and capture the output
result = subprocess.run(command, shell=True, stdout=subprocess.PIPE, text=True)
# Print the output
print("Command Output:")
print(result.stdout)
# Example 2: Run a Python Script as a Subprocess
python_script = "print('Hello from subprocess!')"
# Run the Python script and capture the output
result = subprocess.run(["python", "-c", python_script], stdout=subprocess.PIPE, text=True)
# Print the output
print("\nPython Script Output:")
print(result.stdout)
```

Example 1 runs the ls -l command using the subprocess.run function. The **stdout=subprocess.PIPE**parameter captures the standard output of the command, and text=True ensures that the output is returned as a string. The result is then printed.

Example 2 runs a simple Python script using the **-c** option. The script prints "Hello from subprocess!" to the console. The output of the script is captured and printed.

8. re (Regular Expressions):

Purpose: Provides a set of functions that allows us to search a string for a match.

Common Functions:

re.search(pattern, string): Searches the string for a match and returns a match object if there's a match.

re.findall(pattern, string): Finds all occurrences of the pattern in the string.

- **Basic Pattern Matching:**

```
import re
# Search for a pattern in a string
pattern = r'\b\w+oo\w+\b'
text = "The cat in the room says meow"
match = re.search(pattern, text)
if match:
    print("Found:", match.group())
else:
    print("Pattern not found")
```

- **Find All Matches:**

```
import re
# Find all occurrences of a pattern in a string
pattern = r'\b\w+oo\w+\b'
text = "The cat in the room says meow and the dog outside says
woof"
matches = re.findall(pattern, text)
print("Matches:", matches)
```

- **Capture Groups:**

```
import re
# Use parentheses for capturing groups
pattern = r'(\d{2})/(\d{2})/(\d{4})'
date_string = "05/20/2023"
match = re.match(pattern, date_string)
if match:
    month, day, year = match.groups()
    print(f"Month: {month}, Day: {day}, Year: {year}")
else:
    print("Invalid date format")
```

- **Replace and Substitution:**

```
import re
# Replace a pattern in a string
pattern = r'\bcat\b'
text = "The black cat is on the mat. Another cat is sleeping."
replacement = "dog"
new_text = re.sub(pattern, replacement, text)
print("Original Text:", text)
print("Modified Text:", new_text)
```

- **Case-Insensitive Matching:**

```
import re
# Perform case-insensitive matching
pattern = re.compile(r'python', re.IGNORECASE)
text = "Python is a popular programming language. python is also used."
matches = pattern.findall(text)
print("Matches:", matches)
```

- **Anchors and Boundaries:**

```
import re
# Use anchors and boundaries
pattern = r'\bword\b'
text = "This word is a keyword. Anotherword is not."
matches = re.findall(pattern, text)
print("Matches:", matches)
```

Examples

1. Testing examples

Testing is a crucial aspect of software development to ensure that your code works as expected and to catch any potential issues early on. In Python, the 'unittest' module provides a built-in testing framework. **Here's an example demonstrating basic unit testing:**

Suppose you have a simple function that adds two numbers in a file called **`math_operations.py`:**

```
# math_operations.py
def add_numbers(a, b):
    return a + b
```

Now, you can create a test file, e.g., **`test_math_operations.py`**, to write unit tests for this function:

```
# test_math_operations.py
import unittest
from math_operations import add_numbers
class TestMathOperations(unittest.TestCase):
    def test_add_numbers(self):
# Test the add_numbers function
    result = add_numbers(3, 4)
self.assertEqual(result, 7, "Incorrect addition result")
if __name__ == '__main__':
unittest.main()
```

In this example:

- A test class **`TestMathOperations`** is created that inherits from **`unittest.TestCase`.**
- A test method **`test_add_numbers`** is defined to check if the **`add_numbers`** function returns the correct result.
- The **`assertEqual`** method is used to verify that the result of **`add_numbers(3, 4)`** is equal to 7. If not, an error message is displayed.

To run the tests, execute the test file:
python test_math_operations.py

If the tests pass, you'll see an output indicating that the test ran successfully. If there are issues, the test framework will provide information about which tests failed and why.
Besides **`assertEqual`, `unittest`** provides other assertion methods like **`assertTrue`, `assertFalse`, `assertRaises`**, etc., depending on the type of test you want to perform.
It's worth noting that there are other testing frameworks available for Python, such as **`pytest`** and **`nose2`**, each with its own features and syntax. The choice of testing framework often depends on personal preference and project requirements.

Network Programming (sockets)

Network programming with sockets in Python allows you to create applications that can communicate over a network using the Internet Protocol (IP). Sockets provide a low-level interface for network communication. Here's a simple example of a client-server architecture using sockets:

1. Server Side:

```
import socket
# Create a socket object
server_socket = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
# Bind the socket to a specific address and port
host = '127.0.0.1' # Localhost
port = 12345
```

```
server_socket.bind((host, port))
# Listen for incoming connections
server_socket.listen(5)
print(f"Server listening on {host}:{port}")
while True:
# Establish a connection with the client
client_socket, addr = server_socket.accept()
    print(f"Got connection from {addr}")
# Send a welcome message to the client
    message = "Welcome to the server!"
client_socket.send(message.encode('utf-8'))
# Receive data from the client
    data = client_socket.recv(1024).decode('utf-8')
    print(f"Received data: {data}")
# Close the connection
client_socket.close()
```

2. Client Side:

```
import socket
# Create a socket object
client_socket = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
# Connect to the server
host = '127.0.0.1' # Localhost
port = 12345
client_socket.connect((host, port))
# Receive the welcome message from the server
message = client_socket.recv(1024).decode('utf-8')
print(f"Received message from the server: {message}")
# Send data to the server
data_to_send = "Hello, server!"
client_socket.send(data_to_send.encode('utf-8'))
# Close the connection
client_socket.close()
```

In this example:

- The server creates a socket using **socket.socket()**, binds it to a specific address and port using **bind()**, and listens for incoming connections using **listen()**.
- The client creates a socket, connects to the server using **connect()**, and communicates with the server by sending and receiving data.

To run this example:

- Save the server code in a file named **server.py**.
- Save the client code in a file named **client.py**.
- Open two terminal windows and run the server in one and the client in the other.

This simple example demonstrates a basic client-server interaction using sockets. In a real-world scenario, you would handle errors, implement data serialization, and manage more complex communication protocols depending on your application's requirements.

Chapter 8
Web Development with Python

Introduction to Web Development

Web development is the process of creating and maintaining websites or web applications. It involves various aspects, including front-end development, back-end development, and the integration of databases. Web development encompasses a wide range of technologies, languages, and frameworks. Here's an introduction to the key components of web development:

1. Front-End Development:

a. HTML (Hypertext Markup Language):

Description: HTML is the standard markup language for creating the structure and content of web pages. It defines the elements and their attributes, such as headings, paragraphs, links, images, and forms.

b. CSS (Cascading Style Sheets):

Description: CSS is used for styling HTML elements. It controls the layout, appearance, and presentation of web pages. With CSS, you can define colors, fonts, spacing, and responsive designs.

c. JavaScript:

Description: JavaScript is a scripting language that adds interactivity to web pages. It enables the creation of dynamic content, client-side validation, and the manipulation of the Document Object Model (**DOM**).

d. Front-End Frameworks (e.g., React, Angular, Vue):

Description: Front-end frameworks provide pre-built components and tools for building user interfaces. They enhance the development process and facilitate the creation of interactive and responsive web applications.

2. Back-End Development:

a. Server-Side Languages (e.g., Python, Node.js, Ruby, PHP, ASP.NET):

Description: Server-side languages handle the logic and processing on the server. They interact with databases, perform business logic, and generate dynamic content before sending it to the client's browser.

b. Server-Side Frameworks (e.g., Django, Flask, Express, Ruby on Rails):

Description: Frameworks provide a structured way to build server-side applications. They include tools and conventions for handling routing, middleware, and database interactions, making development more efficient.

c. Databases (e.g., MySQL, PostgreSQL, MongoDB):

Description: Databases store and manage the data used by web applications. They allow for the retrieval, storage, and manipulation of information. Different types of databases, such as relational and NoSQL databases, serve various needs.

3. Full-Stack Development:

a. Full-Stack Developers:

Description: Full-stack developers have expertise in both front-end and back-end development. They can work on the entire web application stack, from designing user interfaces to implementing server logic and database interactions.

4. Web Development Workflow:

a. Version Control (e.g., Git):

Description: Version control systems like Git help developers track changes to their code, collaborate with others, and manage different versions of their projects.

b. IDEs (Integrated Development Environments):

Description: IDEs provide a development environment with features such as code highlighting, debugging tools, and version control integration, enhancing the coding experience.

c. Build Tools (e.g., Webpack, Gulp):

Description: Build tools automate tasks such as bundling, minification, and transpilation of code. They optimize assets for deployment and improve the overall performance of web applications.

5. Web Hosting and Deployment:

a. Web Hosting Services (e.g., Heroku, AWS, DigitalOcean):

Description: Web hosting services allow developers to deploy and run their web applications on servers accessible over the internet.

b. Continuous Integration and Continuous Deployment (CI/CD):

Description: CI/CD practices involve automating the testing, building, and deployment of code changes. This ensures a more streamlined and efficient development process.

Conclusion:

Web development is a dynamic field that evolves with new technologies and trends. Whether you're building a personal website, an e-commerce platform, or a complex web application, understanding both front-end and back-end development is essential for creating effective and user-friendly experiences on the web.

Flask and Django Frameworks

Flask and Django are both popular web frameworks for Python, but they have different philosophies, use cases, and levels of complexity. Let's explore each of them:

Flask:

Philosophy:

Flask follows a micro-framework philosophy, providing the essentials for building web applications without imposing a strict structure.

It is designed to be lightweight and flexible, allowing developers to choose components and libraries based on their needs.

Key Features:

Routing: Define routes for different URLs and handle HTTP requests.

Templates: Use Jinja2 templating engine for rendering dynamic content.

ORM Integration: Flask can be used with various ORMs (Object-Relational Mapping), such as SQLAlchemy.

Extensions: Flask has a modular design with numerous extensions for adding functionalities like authentication, forms, and more.

Use Cases:

Flask is suitable for small to medium-sized applications or when a minimalistic approach is preferred.

It is often chosen for prototyping, APIs, and projects with specific requirements that benefit from its simplicity.

Example Code:

```
from flask import Flask
app = Flask(__name__)
@app.route('/')
def hello_world():
    return 'Hello, World!'
if __name__ == '__main__':
app.run(debug=True)
```

Django:

Philosophy:

Django follows the "batteries-included" philosophy, providing a comprehensive set of features out of the box.

It follows the "Don't Repeat Yourself" (DRY) and "Convention Over Configuration" principles, promoting code organization and reducing boilerplate.

Key Features:

Admin Interface: An automatically generated admin interface for managing application data.

ORM (Object-Relational Mapping): Built-in ORM for database interactions.

Authentication and Authorization: Includes a user authentication system with built-in security features.

Template Engine: Uses Django's template engine for rendering dynamic content.

Forms: Simplifies form handling and validation.

Use Cases:

Django is suitable for larger projects and applications with complex requirements.

It is commonly used for content management systems, e-commerce platforms, and any project where a full-stack framework with built-in features is advantageous.

Example Code:

```
from django.http import HttpResponse
from django.urls import path
from django.shortcuts import render
```

```
def index(request):
    return render(request, 'index.html', {'message': 'Hello, World!'})

urlpatterns = [
    path('', index, name='index'),
]
```

Which One to Choose?

Flask:

Choose Flask if you prefer a lightweight framework, want more flexibility in choosing components, and are working on smaller projects or prototypes.

Django:

Choose Django if you want a full-stack framework with built-in features, rapid development capabilities, and a more opinionated structure. It's well-suited for larger, more complex applications.

Both Flask and Django have active communities, extensive documentation, and are widely used in the Python web development ecosystem. The choice between them depends on the specific requirements of your project and your development preferences.

Building a Simple Web Application

Building a simple web application involves creating both the front-end and back-end components. In this example, I'll **use Flask** as the web framework and HTML for the front-end. The application will consist of a single page that allows the user to enter their name, and it will display a personalized greeting.

1. Install Flask:

Make sure you have Flask installed. If not, you can install it using:

```
pip install flask
```

2. Create the Flask Application:

Create a file named **`app.py`** with the following content:

```
from flask import Flask, render_template, request
```

```
app = Flask(__name__)
@app.route('/', methods=['GET', 'POST'])
def index():
    if request.method == 'POST':
        name = request.form['name']
        return render_template('greet.html', name=name)
    return render_template('index.html')
if __name__ == '__main__':
app.run(debug=True)
```

3. Create HTML Templates:

Create a folder named `templates` in the same directory as **`app.py`**. Inside this folder, create two files: **`index.html`** and **`greet.html`.**

index.html:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Simple Web App</title>
</head>
<body>
<h1>Welcome to the Simple Web App!</h1>
<form method="post" action="/">
<label for="name">Enter your name:</label>
<input type="text" id="name" name="name" required>
<button type="submit">Submit</button>
</form>
</body>
</html>
```

greet.html:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
```

```
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Greeting</title>
</head>
<body>
<h1>Hello, {{ name }}!</h1>
<p>Thank you for using the Simple Web App.</p>
</body>
</html>
```

4. Run the Application:
In the terminal, run the Flask application:
python app.py
Visit **`http://localhost:5000`** in your web browser. You'll see the homepage where you can enter your name. After submitting the form, you'll be redirected to a personalized greeting page.
This example demonstrates a basic web application using **Flask, HTML, and form handling**. Depending on your needs, you can expand the application by adding more routes, incorporating CSS for styling, and integrating additional features.
Building a simple web application using Django involves creating a project, defining models, setting up views, and creating templates. In this example, I'll guide you through building a basic web application that allows users to enter their name and displays a personalized greeting. Let's get started:

1. Install Django:
Make sure you have Django installed. If not, you can install it using:
pip install django

2. Create a Django Project and App:
In your terminal, run the following commands:
 # Create a Django project
django-admin startprojectmywebapp
 # Navigate to the project directory
cd mywebapp
 # Create a Django app
python manage.py startappgreetapp

3. Define Models:

In the **`greetapp/models.py`** file, define a simple model to store user names:

```
# greetapp/models.py
from django.db import models
class Greeting(models.Model):
    name = models.CharField(max_length=100)
    def __str__(self):
        return self.name
```

4. Run Migrations:

Run the following commands to apply migrations and create the database:

```
python manage.py makemigrations
python manage.py migrate
```

5. Set Up Views:

In the `greetapp/views.py` file, create views to handle rendering the form and processing user input:

```
# greetapp/views.py
from django.shortcuts import render, redirect
from .models import Greeting
def index(request):
    if request.method == 'POST':
        name = request.POST['name']
Greeting.objects.create(name=name)
        return redirect('greet')
    return render(request, 'index.html')
def greet(request):
    greetings = Greeting.objects.all()
    return render(request, 'greet.html', {'greetings': greetings})
```

6. Set Up URLs:

Create `urls.py` in the `greetapp` directory and define the URLs:

```
# greetapp/urls.py
from django.urls import path
from .views import index, greet
```

```
urlpatterns = [
    path('', index, name='index'),
    path('greet/', greet, name='greet'),
]
```

Include these URLs in the main `urls.py` file in the `mywebapp` directory:

```
# mywebapp/urls.py
from django.contrib import admin
from django.urls import include, path
urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('greetapp.urls')),
]
```

7. Create Templates:

Create `templates` folder in the `greetapp` directory and add the following HTML templates.

index.html:

```
<!-- greetapp/templates/index.html -->
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Simple Web App</title>
</head>
<body>
<h1>Welcome to the Simple Web App!</h1>
<form method="post" action="{% url 'index' %}">
    {% csrf_token %}
<label for="name">Enter your name:</label>
<input type="text" id="name" name="name" required>
<button type="submit">Submit</button>
</form>
</body>
</html>
```

greet.html:

```
<!-- greetapp/templates/greet.html -->
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Greetings</title>
</head>
<body>
<h1>Greetings</h1>
    {% for greeting in greetings %}
<p>Hello, {{ greeting.name }}!</p>
    {% endfor %}
</body>
</html>
```

8. Run the Development Server:

In the terminal, run the following command to start the development server:

python manage.py runserver

Visit **`http://localhost:8000`** in your web browser. You'll see the homepage where you can enter your name. After submitting the form, you'll be redirected to a page displaying personalized greetings.

This example demonstrates a basic web application using Django, including model definition, views, templates, and URL routing. Depending on your needs, you can further enhance the application by adding more features and improving the user interface.

Working with Databases (SQL and NoSQL)

Working with databases is a fundamental aspect of web development, allowing you to store, retrieve, and manipulate data for your applications. Two primary types of databases are SQL (relational databases) and NoSQL (non-relational databases). Here's an overview of both

SQL Databases:

1. SQLite (Lightweight SQL Database):

a. Introduction:

- SQLite is a C library that provides a lightweight disk-based database.
- It doesn't require a separate server process and allows access to the database using a nonstandard variant of the SQL query language.

b. Python Integration:

- Python comes with built-in support for SQLite through the **`sqlite3`** module.

c. Example Usage:

```
import sqlite3
# Connect to a database (creates a new file if it doesn't exist)
conn = sqlite3.connect('example.db')
# Create a cursor object to interact with the database
cursor = conn.cursor()
# Create a table
cursor.execute('''
    CREATE TABLE IF NOT EXISTS users (
        id INTEGER PRIMARY KEY,
        name TEXT,
        age INTEGER
    )
''')
# Insert data into the table
cursor.execute('INSERT INTO users (name, age) VALUES (?, ?)',
('ram singh', 25))
# Commit the changes and close the connection
conn.commit()
conn.close()
```

 # 2. PostgreSQL (Advanced SQL Database):

a. Introduction:

- PostgreSQL is a powerful, open-source relational database system.
- It supports advanced SQL features and is known for its extensibility and standards compliance.

b. Python Integration:

- The **`psycopg2`** library is commonly used for connecting to PostgreSQL in Python.

c. Example Usage:

```
import psycopg2
# Connect to a PostgreSQL database
conn = psycopg2.connect(
    host="your_host",
    user="your_user",
    password="your_password",
    database="your_database"
)
# Create a cursor object
cursor = conn.cursor()
# Execute SQL queries
cursor.execute('SELECT * FROM users')
result = cursor.fetchall()
# Commit changes and close the connection
conn.commit()
conn.close()
```

NoSQL Databases:

1. MongoDB (Document-Oriented NoSQL Database):

a. Introduction:

- MongoDB is a widely used document-oriented NoSQL database.
- It stores data in flexible, JSON-like documents, allowing for dynamic schema designs.

b. Python Integration:

- The **`pymongo`** library is commonly used for connecting to MongoDB in Python.

c. Example Usage:

```
from pymongo import MongoClient
# Connect to MongoDB
client = MongoClient("your_mongodb_connection_string")
# Access a database and collection
db = client['mydatabase']
collection = db['mycollection']
# Insert a document
document = {"name": "ram singh", "age": 25}
result = collection.insert_one(document)
```

```
# Query the collection
query_result = collection.find({"age": {"$gte": 21}})
# Iterate through the query results
for document in query_result:
    print(document)
# Close the connection
client.close()
```

2. Redis (In-Memory Data Structure Store):

a. Introduction:

- Redis is an in-memory data structure store, often used as a cache or message broker.
- It supports various data structures such as strings, hashes, lists, sets, and more.

b. Python Integration:

- The `redis` library is commonly used for connecting to Redis in Python.

c. Example Usage:

```
import redis

# Connect to Redis
client = redis.StrictRedis(host='localhost', port=6379, db=0)
# Set a key-value pair
client.set('example_key', 'example_value')
# Get the value by key
value = client.get('example_key')
print(value.decode('utf-8'))
# Close the connection
client.close()
```

Conclusion:

The choice between SQL and NoSQL databases depends on your application's specific requirements. SQL databases are suitable for applications with structured data and complex queries, while NoSQL databases offer flexibility and scalability for applications with dynamic and unstructured data. The examples provided demonstrate basic interactions with both SQL and NoSQL databases using Python.

Chapter 9
Data Science and Python

Data science is a multidisciplinary field that uses scientific methods, processes, algorithms, and systems to extract insights and knowledge from structured and unstructured data. Python has become a popular programming language in the field of data science due to its versatility, ease of learning, and a rich ecosystem of libraries and tools. Here's an overview of data science and the role of Python in this domain:

Key Components of Data Science:

1. **Data Collection:**
   - Gathering relevant data from various sources, such as databases, APIs, CSV files, and more.

2. **Data Cleaning and Preprocessing:**
   - Handling missing values, removing outliers, and transforming data into a suitable format for analysis.

3. **Exploratory Data Analysis (EDA):**
   - Analyzing and visualizing data to understand patterns, relationships, and distributions.

4. **Feature Engineering:**
   - Creating new features from existing data or transforming features to improve model performance.

5. **Model Building:**
   - Developing machine learning models to make predictions, classifications, or identify patterns.

6. **Model Evaluation:**
   - Assessing the performance of models using metrics and validation techniques.

7. **Model Deployment:**
   - Integrating models into production systems for real-world applications.

8. **Communication and Visualization:**
   - Presenting findings and insights to stakeholders through reports, dashboards, and visualizations.

Python in Data Science:

Python is widely used in data science for several reasons:

1. **Rich Ecosystem:**
- Python has a vast ecosystem of libraries and frameworks specifically designed for data science, such as NumPy, pandas, Matplotlib, Seaborn, Scikit-Learn, TensorFlow, and PyTorch.

2. **Ease of Learning:**
- Python's syntax is clear and readable, making it accessible to beginners and facilitating collaboration among team members.

3. **Community Support:**
- The Python data science community is active and collaborative, providing a wealth of resources, tutorials, and solutions to common challenges.

4. **Versatility:**
- Python is a general-purpose programming language, allowing data scientists to seamlessly integrate data analysis, machine learning, and other tasks in a single environment.

5. **Integration with Big Data Technologies:**
- Python integrates well with big data technologies such as Apache Spark, making it suitable for handling large-scale datasets.

 # Example Data Science Workflow in Python:

```
# Import necessary libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
# Load dataset
url = 'https://raw.githubusercontent.com/openai/gpt-3.5-
turbo/main/examples/summarization/input.txt'
data = pd.read_csv(url, delimiter='\t', names=['X', 'y'])
```

```
# Exploratory Data Analysis (EDA)
sns.scatterplot(x='X', y='y', data=data)
plt.title('Scatter Plot of X vs y')
plt.show()
# Data Preprocessing
X = data['X'].values.reshape(-1, 1)
y = data['y'].values
#Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
# Model Building
model = LinearRegression()
model.fit(X_train, y_train)
# Model Evaluation
y_pred = model.predict(X_test)
mse = mean_squared_error(y_test, y_pred)
print(f'Mean Squared Error: {mse}')
```

In this example, we load a dataset, perform exploratory data analysis, preprocess the data, and build a simple linear regression model using Scikit-Learn. This is just a small part of a typical data science workflow, and Python libraries provide tools for every step of the process.

Whether you are working on data analysis, machine learning, or any other aspect of data science, Python's versatility and rich ecosystem make it a powerful choice for data scientists.

NumPy and NumPy Arrays

NumPy (Numerical Python):

NumPy is a powerful Python library for numerical and mathematical operations. It provides support for large, multi-dimensional arrays and matrices, along with a collection of high-level mathematical functions to operate on these arrays.

Key Features of NumPy:

1. **Arrays:**

- NumPy arrays are the core data structure in the library.
- They are similar to Python lists but offer more functionality and efficiency for numerical operations.

2. **Vectorized Operations:**

- NumPy supports vectorized operations, which means that operations can be performed on entire arrays without the need for explicit loops.

3. **Broadcasting:**

- Broadcasting allows NumPy to perform operations on arrays of different shapes and sizes.

4. **Mathematical Functions:**

- NumPy provides a wide range of mathematical functions for operations like linear algebra, Fourier analysis, random number generation, etc.

5. **Integration with Other Libraries:**

- NumPy integrates seamlessly with other libraries like SciPy, Matplotlib, and pandas.

NumPy Arrays:

NumPy arrays are homogeneous, multi-dimensional, and memory-efficient data structures. They can be created using lists, tuples, or other arrays.

Creating NumPy Arrays:

```
import numpy as np
# Create a 1D array from a list
arr_1d = np.array([1, 2, 3, 4, 5])
# Create a 2D array from a list of lists
arr_2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
# Create an array of zeros
zeros_array = np.zeros((3, 4))
# Create an array of ones
ones_array = np.ones((2, 3))
# Create a range of values
range_array = np.arange(0, 10, 2) # start, stop, step
# Create a linearly spaced array
linspace_array = np.linspace(0, 1, 5) # start, end, number of points
# Create a random array
random_array = np.random.rand(2, 3) # random values from a uniform distribution
```

NumPy Array Operations:

```
# Element-wise operations
arr = np.array([1, 2, 3, 4])
result = arr + 2 # [3, 4, 5, 6]
# Vectorized operations
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
result = arr1 + arr2 # [5, 7, 9]
# Array broadcasting
matrix = np.array([[1, 2, 3], [4, 5, 6]])
scalar = 2
result = matrix * scalar # [[2, 4, 6], [8, 10, 12]]
# Array indexing and slicing
arr = np.array([1, 2, 3, 4, 5])
subset = arr[1:4] # [2, 3, 4]
# Reshape array
arr = np.array([1, 2, 3, 4, 5, 6])
reshaped_arr = arr.reshape((2, 3))
```

These examples provide a glimpse into the capabilities of NumPy and its array operations. **NumPy is widely used in scientific computing, data analysis, machine learning,** and other domains where numerical operations are prevalent.

Data Manipulation with Pandas

Pandas is a popular Python library for data manipulation and analysis. It provides data structures like Series and DataFrame, which are designed for efficient and intuitive handling of structured data. Here's an overview of common data manipulation tasks using Pandas:

1. Loading Data:

Pandas supports various file formats, such as CSV, Excel, SQL databases, and more.

```
import pandas as pd
# Load a CSV file into a DataFrame
df = pd.read_csv('example.csv')
# Load an Excel file into a DataFrame
df_excel = pd.read_excel('example.xlsx')
# Connect to a SQL database and read data
```

```
import sqlite3
conn = sqlite3.connect('example.db')
query = 'SELECT * FROM table_name'
df_sql = pd.read_sql_query(query, conn)
```

2. Exploratory Data Analysis (EDA):

Pandas provides functions to explore and understand the structure of your data.

```
# Display the first few rows of the DataFrame
print(df.head())
# Get basic statistics for numerical columns
print(df.describe())
# Check for missing values
print(df.isnull().sum())
# Filter and subset data
subset = df[df['Column'] > 50]
# Group by and aggregate
grouped_data = df.groupby('Category')['Value'].mean()
```

3. Data Cleaning:

Pandas helps in cleaning and preprocessing data by handling missing values, duplicates, and outliers.

```
# Drop rows with missing values
df_cleaned = df.dropna()
# Fill missing values with a specific value
df_filled = df.fillna(0)
# Remove duplicate rows
df_no_duplicates = df.drop_duplicates()
# Remove outliers using z-score
from scipy.stats import zscore
df_no_outliers = df[(np.abs(zscore(df['Column'])) < 3)]
```

4. Data Transformation:

Pandas facilitates the transformation of data, including creating new columns, applying functions, and reshaping data.

```
# Create a new column based on existing columns
df['New Column'] = df['Column1'] + df['Column2']
# Apply a function element-wise
df['Column'] = df['Column'].apply(lambda x: x*2)
```

 # Pivot table for reshaping data

```
pivot_table = df.pivot_table(index='Category', columns='Month', values='Value', aggfunc='mean')
```

 # Melt to convert wide format to long format

```
df_long = pd.melt(df, id_vars=['ID'], value_vars=['Jan', 'Feb'], var_name='Month', value_name='Value')
```

5. Merging and Concatenating:

Combine data from multiple sources using merge and concatenate operations.

 # Concatenate DataFrames vertically

```
df_concat = pd.concat([df1, df2])
```

 # Merge DataFrames based on a common column

```
df_merged = pd.merge(df1, df2, on='KeyColumn', how='inner')
```

6. Handling Dates and Times:

Pandas provides functionality for working with dates and times.

 # Convert a column to datetime format

```
df['Date'] = pd.to_datetime(df['Date'])
```

 # Extract year, month, day from datetime column

```
df['Year'] = df['Date'].dt.year
df['Month'] = df['Date'].dt.month
df['Day'] = df['Date'].dt.day
```

7. Handling Categorical Data:

Encode and handle categorical variables.

 # Convert categorical column to numerical using one-hot encoding

```
df_encoded = pd.get_dummies(df, columns=['Category'], prefix='Category')
```

 # Map categorical values to numerical values

```
mapping = {'Low': 1, 'Medium': 2, 'High': 3}
df['Priority'] = df['Priority'].map(mapping)
```

Pandas is a powerful tool for data manipulation and analysis in Python. Its intuitive syntax and rich functionality make it a go-to choice for working with structured data in various data science and analysis projects.

Data Visualization with Matplotlib and Seaborn

Matplotlib and Seaborn are popular Python libraries for data visualization. Matplotlib is a comprehensive 2D plotting library, and Seaborn is built on top of Matplotlib, providing a high-level interface for drawing attractive statistical graphics. Here's an overview of basic data visualization using Matplotlib and Seaborn:

Matplotlib:

```
# Line Plot:
import matplotlib.pyplot as plt
# Sample data
x = [1, 2, 3, 4, 5]
y = [2, 4, 6, 8, 10]
# Create a line plot
plt.plot(x, y, label='Line Plot')
# Add labels and title
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Line Plot Example')
# Show legend
plt.legend()
# Show the plot
plt.show()
# Scatter Plot:
# Create a scatter plot
plt.scatter(x, y, label='Scatter Plot', color='red', marker='o')
# Add labels and title
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Scatter Plot Example')
# Show legend
plt.legend()
# Show the plot
plt.show()
```

Seaborn:

```
# Distribution Plot:
import seaborn as sns
# Create a distribution plot
sns.histplot(data=df, x='Column', kde=True, color='skyblue')
# Add labels and title
plt.xlabel('X-axis')
plt.ylabel('Frequency')
plt.title('Distribution Plot Example')
# Show the plot
plt.show()
# Box Plot:
# Create a box plot
sns.boxplot(data=df, x='Category', y='Value', palette='Set2')
# Add labels and title
plt.xlabel('Category')
plt.ylabel('Value')
plt.title('Box Plot Example')
# Show the plot
plt.show()
```

Combining Matplotlib and Seaborn:

While Seaborn simplifies the creation of certain plots, Matplotlib can still be used for customization.

```
# Create a scatter plot with Seaborn
sns.scatterplot(x='Column1', y='Column2', data=df, hue='Category',
palette='viridis')
# Add labels and title using Matplotlib
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Scatter Plot with Seaborn')
# Show the legend
plt.legend()
# Show the plot
plt.show()
```

These are just simple examples, and both Matplotlib and Seaborn offer a wide range of customization options for creating more complex and informative visualizations. Whether you need basic line plots or advanced statistical graphics, these libraries provide tools to meet your data visualization needs in Python.

Chapter 10
Testing and Debugging

Testing and debugging are critical aspects of the software development process to ensure that your code functions correctly and is free of errors.

Writing Tests with unit test

Unittest is the built-in testing framework in Python. It provides a set of tools for constructing and running tests, and it follows the xUnit style. Here's an overview of writing tests with `unittest`:

Basic Structure of a Test:

1. Test Class:

- Create a test class that inherits from `unittest.TestCase`.
- Each test is a method within this class.

```
import unittest
class MyTests(unittest.TestCase):
    def test_example(self):
# Your test code here
self.assertEqual(1 + 1, 2)
```

2. Assertions:

- Use assertion methods like `assertEqual`, `assertTrue`, `assertFalse`, etc., to check if the expected conditions are met.
- If an assertion fails, the test fails.

 # Example Test Case:

1. Consider a simple function that adds two numbers:

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

Now, let's write a test case for this function:

```
# test_my_module.py
import unittest
from my_module import add
class TestAddFunction(unittest.TestCase):
    def test_add_positive_numbers(self):
        result = add(3, 4)
self.assertEqual(result, 7)
    def test_add_negative_numbers(self):
        result = add(-2, 5)
```

```
self.assertEqual(result, 3)
    def test_add_zero(self):
        result = add(0, 0)
        self.assertEqual(result, 0)
if __name__ == '__main__':
    unittest.main()
```

Running Tests:

1. Command Line:

- Run the tests from the command line:

```
python -m unittest test_my_module.py
```

2. Test Discovery:

- If you have multiple test modules, you can use test discovery:

```
python -m unittest discover
```

Test Fixtures:

`unittest` supports the use of test fixtures to set up and tear down resources for your tests. Fixtures are functions or methods that are run before or after each test method.

```
import unittest
class MyTests(unittest.TestCase):
    def setUp(self):
# Code to run before each test method
        pass
def tearDown(self):
# Code to run after each test method
        pass
    def test_example(self):
# Your test code here
self.assertEqual(1 + 1, 2)
```

Skipping Tests:

You can skip certain tests using the `@unittest.skip` decorator or conditionally skip them using `unittest.skipIf` or `unittest.skipUnless`.

```
import unittest
```

```
class MyTests(unittest.TestCase):
    @unittest.skip("Skipping this test")
    def test_example(self):
        self.assertEqual(1 + 1, 2)
    @unittest.skipIf(True, "Skipping this test conditionally")
    def test_another_example(self):
        self.assertEqual(2 * 2, 4)
```

Conclusion:
`unittest` provides a robust and built-in framework for writing and running tests in Python. While other testing frameworks like `pytest` and `nose` offer additional features and flexibility, `unittest` is widely used and is part of the standard library. Choose the testing framework that best fits your project's requirements and your personal preferences.

Best Practices

Best practices in software development aim to enhance code quality, maintainability, and collaboration. Here are some general best practices for writing Python code:

1. **Code Readability:**
- Follow PEP 8(https://www.python.org/dev/peps/pep-0008/) for Python style guide conventions.
- Use meaningful variable and function names.
- Write comments for complex sections of code but strive for self-explanatory code.

2. **Modularization:**
- Break down your code into small, reusable functions or classes.
- Use modules and packages to organize code logically.

3. **Docstrings:**
- Include docstrings for modules, classes, and functions to provide documentation.
- Follow PEP 257(https://www.python.org/dev/peps/pep-0257/) for docstring conventions.

4. **Testing:**
- Write unit tests for your code using frameworks like `unittest`, `pytest`, or `nose`.
- Aim for comprehensive test coverage to ensure the correctness of your code.
- Run tests regularly and automate testing where possible.

5. **Version Control:**
- Use version control systems like Git.
- Commit frequently with clear and concise commit messages.
- Branch your code for features or bug fixes.

6. **Virtual Environments:**
- Use virtual environments (e.g., `venv` or `virtualenv`) to isolate project dependencies.
- Include a `requirements.txt` file for specifying project dependencies.

7. **Error Handling:**
- Use try-except blocks for handling exceptions.
- Log errors to help diagnose issues in production.
- Avoid using bare `except:` clauses; specify the exception type whenever possible.

8. **Use List Comprehensions:**
- Utilize list comprehensions for concise and readable code when creating lists.

```
squares = [x**2 for x in range(10)]
```

9. **Generators:**
- Use generators for memory-efficient iteration when dealing with large datasets.

```
def square_numbers(n):
    for i in range(n):
        yield i**2
```

10. Avoid Global Variables:
- Minimize the use of global variables; prefer passing parameters to functions.
- Use constants (uppercase) for variables that should not be modified.

11. Consistent Naming Conventions:
- Follow consistent naming conventions for variables, functions, and classes.
- Use snake_case for variables and functions, and CamelCase for classes.

12. Continuous Integration:
- Use continuous integration tools (e.g., Travis CI, Jenkins, GitHub Actions) to automate testing and ensure code quality.

13. Optimize Imports:
- Only import what you need to avoid cluttering the namespace.
- Group imports according to PEP 8 recommendations.

14. Security Best Practices:
- Be mindful of security considerations (e.g., input validation, avoiding SQL injection).
- Regularly update dependencies to patch security vulnerabilities.

15. Performance Optimization:
- Profile your code using tools like `cProfile` to identify bottlenecks.
- Optimize critical sections based on profiling results.

16. Consistent Formatting:
- Use an automated code formatter like `black` to maintain consistent formatting.

These best practices contribute to writing maintainable, scalable, and error-free Python code. Following them helps improve collaboration among team members and facilitates the long-term maintenance of your codebase.

Chapter 11
Deployment and Packaging

Packaging Your Python Application

Packaging a Python application involves organizing your code and resources into a distributable format that can be easily installed and distributed to users or other developers. Here's an overview of the process:

1. Project Structure:

Maintain a well-organized directory structure for your project. A typical structure might include:

```
your_project/
├── your_package/
│   ├── __init__.py
│   ├── module1.py
│   └── module2.py
├── README.md
├── setup.py
└── requirements.txt
```

- **`your_package`:** Contains your actual Python code (modules, packages).
- **`README.md`:** Documentation for your project.
- **`setup.py`:** Script for packaging and distribution.
- **`requirements.txt`:** List of dependencies.

2. Creating setup.py:

- **setup.py** is a Python script that contains information about your package and how it should be installed.

```
from setuptools import setup, find_packages
setup(
    name='your_package_name',
    version='1.0.0',
    packages=find_packages(),
```

```
install_requires=['dependency1', 'dependency2'],
entry_points={
        'console_scripts': [
                'your_script_name=your_package.module:
                main_function',
        ],
    },
    author='Your Name',
    author_email='your@email.com',
    description='Description of your package',
    url='https://github.com/your_username/your_package',
)
```

- Replace placeholders **(`your_package_name`, `dependency1`, etc.)** with your package's information.
- **`entry_points`:** Define any command-line scripts associated with your package.

3. **Adding __init__.py:**
- Include a**__init__.py** file in your package directories to indicate that they are Python packages.

4. **Documentation:**
- Provide comprehensive documentation, including a **README.md** file, **usage instructions**, and examples.

5. **Version Control:**
- Use version control **(e.g., Git)** to manage your project and make it accessible for distribution.

6. **Building and Distributing:**
- Use tools like setuptools or wheel to build your package.

```
# Create a source distribution
python setup.py sdist
# Create a wheel distribution
python setup.py bdist_wheel
```

7. Upload to Package Index (PyPI):
Publish your package on PyPI for easy installation by others.
 # Upload to PyPI using Twine
pip install twine
twine upload dist

8. Installation:
To install your package from PyPI:
pip install your_package_name

9. Testing Installation:
Create a new virtual environment and test the installation of your package:
 # Create a virtual environment
python -m venv myenv
source myenv/bin/activate **# Activate the virtual environment**
 # Install your package
pip install your_package_name

10. Continuous Integration (CI):
- Set up CI/CD pipelines (e.g., GitHub Actions, Travis CI) to automate package building and testing.

Packaging your Python application involves making it easily installable and distributable. By following above steps and best practices, you can create a well-structured, documented, and easily installable package for your Python project.

Deploying Python Applications

Deploying Python applications involves making your application available for use by end-users or making it accessible on servers or cloud platforms. The deployment process varies based on the type of application (**web, desktop, API, etc**.) and the hosting environment. Here's an overview of deploying different types of Python applications:

1. Web Applications:

a. Using Web Frameworks (e.g., Flask, Django):

Deployment to a Web Server:

- Use application servers like Gunicorn, uWSGI, or ASGI servers for deploying Flask or Django applications.
- Deploy behind a reverse proxy server like Nginx or Apache for handling client requests.

Cloud Platform Deployment:

- Host applications on cloud platforms like AWS, Google Cloud Platform, or Heroku.
- Platforms often provide specific deployment guides for popular frameworks.

b. Serverless Deployment:

Deploying as Serverless Functions:

- Utilize serverless platforms like AWS Lambda, Azure Functions, or Google Cloud Functions.
- Frameworks like **Zappa** (for Flask/Django) or Serverless Framework simplify deployment to serverless environments.

2. Desktop Applications:

- Use packaging tools like PyInstaller, cx_Freeze, or Py2exe to create standalone executables for Windows, macOS, or Linux platforms.
- Distribute the compiled executables to end-users or through platforms like the Microsoft Store, Apple App Store, or Snapcraft (for Linux).

3. APIs:

- Deploy APIs using frameworks like Flask or FastAPI on web servers or cloud platforms.
- Secure the API endpoints using authentication mechanisms (e.g., **JWT, OAuth**).

4. **Continuous Integration/Continuous Deployment (CI/CD):**
- Set up CI/CD pipelines (using tools like **Jenkins, GitLab CI/CD, GitHub Actions**) to automate the build, test, and deployment processes.
- Automate deployment to your hosting environment when changes are pushed to version control.

5. **Docker Containers:**
- Containerize your application using Docker for consistency across different environments.
- Deploy Docker containers to container orchestration platforms like Kubernetes or Docker Swarm.

6. **Database Deployment:**
- Configure and deploy databases separately based on the type of application (SQL, NoSQL).
- Use managed database services provided by cloud platforms for easier management and scalability.

7. **Monitoring and Logging:**
- Implement logging and monitoring tools (**e.g., Prometheus, Grafana, ELK Stack**) to track application performance and errors in production environments.
- Set up alerts for critical issues.

8. **Security Considerations:**
- Secure sensitive data using encryption and follow best practices for user authentication and authorization.
- Regularly update dependencies and libraries to patch security vulnerabilities.

Conclusion:

Deploying Python applications involves various steps based on the application type and the hosting environment. It's crucial to follow best practices, automate where possible, and ensure that your application is secure and performs well in production environments. Each deployment may have its specific requirements, so refer to platform-specific documentation or guidelines for a smoother deployment experience.

Virtual Environments for Isolation

Virtual environments in Python are used to create isolated environments with their own Python installations and package dependencies. They allow you to work on multiple projects with different dependency requirements without conflicts. Here's an overview of using virtual environments:

1. Creating Virtual Environments:

Using venv (Built-in):

- Create a new virtual environment in a directory:

```
python -m venv myenv
```

- Activate the virtual environment:

Windows:

```
myenv\Scripts\activate
```

Unix or MacOS:

```
source myenv/bin/activate
```

2. Managing Packages:

- Install packages within the virtual environment using **pip**:
  pip install package_name
- To freeze installed packages into a **requirements.txt** file:
  pip freeze > requirements.txt
- Install dependencies from a requirements.txt file:
  pip install -r requirements.txt

3. Deactivating the Virtual Environment:

- To deactivate the virtual environment:
  deactivate

4. Benefits of Virtual Environments:

- **Isolation:** Each environment has its own set of dependencies, avoiding conflicts between different projects.
- **Portability:** Virtual environments can be easily shared and recreated on different systems.
- **Dependency Management:** Facilitates clean installation and management of project-specific dependencies.

4. Benefits of Virtual Environments:

- **Isolation:** Each environment has its own set of dependencies, avoiding conflicts between different projects.
- **Portability:** Virtual environments can be easily shared and recreated on different systems.
- **Dependency Management:** Facilitates clean installation and management of project-specific dependencies.

5. Using Other Tools:

virtualenv:

An alternative to venv, virtualenv is a third-party package for creating virtual environments.

```
pip install virtualenv
virtualenv myenv
```

- Activation and usage are similar to **venv**.

pipenv:

- A higher-level tool that combines package management and virtual environment creation.

```
pip install pipenv
pipenv install package_name
```

- Manages both package installation and virtual environments through a Pipfile and Pipfile.lock.

conda:

From the Anaconda distribution, conda manages environments and packages, particularly useful for data science applications.

```
conda create --name myenv python=3.8
conda activate myenv
```

Chapter 12
Advanced
Python Concepts

Metaclasses

Metaclasses in Python are classes responsible for creating classes. They offer a way to modify the behavior of class creation and control how classes are defined. Understanding metaclasses is an advanced topic in Python and is used in specific scenarios.

Here's an overview of metaclasses:

1. Basics of Metaclasses:

- In Python, everything is an object, including classes. A class itself is an instance of a metaclass.
- The default metaclass in Python is type. When you create a class, Python implicitly uses type as the metaclass.
- Metaclasses allow you to customize how classes are created by defining the __**new**__ and __**init**__ methods. __**new**__ is used for creating the object, while __**init**__ initializes it.

2. Creating Metaclasses:

You can create a custom metaclass by subclassing type:

```
class CustomMeta(type):
    def __new__(cls, name, bases, dct):
# Modify or customize the class creation process here
        return super().__new__(cls, name, bases, dct)
    def __init__(self, name, bases, dct):
        super().__init__(name, bases, dct)
# Additional initialization if needed
```

3. Using Metaclasses:

To use a custom metaclass, you define a class and specify the metaclass using the metaclass keyword argument:

```
class MyClass(metaclass=CustomMeta):
# Class definition here
pass
```

4. Use Cases of Metaclasses:

- **Framework Creation:**Metaclasses can be used to create frameworks where classes automatically register themselves or enforce certain behaviors.

- **API Design:** They can be utilized to enforce rules, validate class attributes, or modify class behavior at the time of creation.
- **Singleton Pattern:**Metaclasses can be used to implement the Singleton design pattern by controlling the instantiation of classes.
- **ORMs (Object-Relational Mappers):** Some ORMs use metaclasses to map class attributes to database columns.

5. Considerations:

Metaclasses are powerful but can make code less readable and more complex. They should be used sparingly when simpler solutions aren't feasible.

Overusing metaclasses can lead to code that's difficult to understand and maintain, so they're usually reserved for advanced scenarios.

Example:

Here's a simple example demonstrating the usage of a metaclass:

```
class CustomMeta(type):
    def __new__(cls, name, bases, dct):
        dct['custom_attr'] = 100
        # Adding a custom attribute to classes
        return super().__new__(cls, name, bases, dct)
class MyClass(metaclass=CustomMeta):
pass
print(MyClass.custom_attr) # Output: 100
```

This example illustrates how the metaclassCustomMeta modifies the class creation process by adding a custom attribute custom_attr to classes created with it.

Metaclasses are a powerful tool in Python, but they're generally considered advanced and might not be necessary for most everyday programming tasks. Understanding them can be beneficial for scenarios where customization of class creation is required.

Design Pattern in Python

Design patterns are reusable solutions to common problems in software design. They provide templates and guidelines to solve specific problems effectively in a flexible and maintainable way. Python supports various design patterns, and understanding them can significantly improve your code's structure and maintainability. Here are some commonly used design patterns in Python:

1. **Creational Design Patterns:**

a. **Singleton Pattern:**

Ensures a class has only one instance and provides a global point of access to it.

b. **Factory Method Pattern:**

Defines an interface for creating an object but allows subclasses to alter the type of objects that will be created.

c. **Abstract Factory Pattern:**

Provides an interface to create families of related or dependent objects without specifying their concrete classes.

2. **Structural Design Patterns:**

a. **Adapter Pattern:**

Allows objects with incompatible interfaces to collaborate by converting the interface of one class into another interface that clients expect.

b. **Decorator Pattern:**

Adds behavior to objects dynamically without affecting other objects of the same class.

c. **Facade Pattern:**

Provides a unified interface to a set of interfaces in a subsystem, implifying their usage.

3. **Behavioral Design Patterns:**

a. **Observer Pattern:**

Defines a one-to-many dependency between objects where changes in one object trigger updates in other objects.

b. **Strategy Pattern:**

Defines a family of algorithms, encapsulates each one, and makes them interchangeable. Clients can choose the appropriate algorithm.

c. **Command Pattern:**

Encapsulates a request as an object, thereby allowing parameterization of clients with queues, requests, and operations.

Implementation in Python:

Each pattern has a specific implementation tailored to the problem it solves. Python's flexibility allows for elegant and concise implementations:

Singleton Pattern Example:

```
class Singleton:
    _instance = None
    def __new__(cls):
        if not cls._instance:
            cls._instance = super().__new__(cls)
            return cls._instance
singleton1 = Singleton()
singleton2 = Singleton()
print(singleton1 is singleton2) # Output: True (both variables refer to the same instance)
```

Observer Pattern Example:

```
class Subject:
    def __init__(self):
        self._observers = []
    def attach(self, observer):
        self._observers.append(observer)
    def notify(self, message):
        for observer in self._observers:
            observer.update(message)
class Observer:
    def update(self, message):
        print(f"Received message: {message}")
        subject = Subject()
        observer1 = Observer()
        observer2 = Observer()
        subject.attach(observer1)
        subject.attach(observer2)
        subject.notify("Hello Observers!")
```

Conclusion:
Design patterns help solve recurring problems in software development by providing proven solutions. They improve code readability, maintainability, and scalability. While these patterns are powerful, it's essential to apply them judiciously, considering the context and specific requirements of your project, to avoid over-engineering or unnecessary complexity.

Functional Programming in Python

Functional programming (FP) is a paradigm that treats computation as the evaluation of mathematical functions and avoids changing state or mutable data. Python supports functional programming concepts and offers features that enable functional programming practices. Here's an overview of functional programming in Python:

1. **First-Class Functions:**
   In Python, functions are first-class citizens, meaning they can be:
- Assigned to variables.
- Passed as arguments to other functions.
- Returned as values from other functions.

2. **Lambda Functions:**
- Lambda functions (anonymous functions) can be defined using the `**lambda**` keyword:

```
square = lambda x: x ** 2
print(square(5)) # Output: 25
```

- Lambda functions are often used in functional programming paradigms to create simple functions on-the-fly.

3. **Higher-Order Functions:**
Python supports higher-order functions, which are functions that take other functions as arguments or return them as results.

```
def apply_operation(func, x, y):
return func(x, y)
def add(a, b):
return a + b
result = apply_operation(add, 4, 5)
print(result) # Output: 9
```

4. Map, Filter, and Reduce:

- **map() Function**: Applies a function to all items in an input list.

```
number = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x ** 2, numbers))
print(squared) # Output: [1, 4, 9, 16, 25]
```

- **filter() Function:** Filters elements based on a given function.

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

- **reduce() Function (in the `functools` module):** Applies a rolling computation to sequential pairs of values.

```
from functools import reduce
numbers = [1, 2, 3, 4, 5]
product = reduce(lambda x, y: x * y, numbers)
print(product) # Output: 120 (1 * 2 * 3 * 4 * 5)
```

5. Immutable Data and Avoiding Side Effects:

Functional programming encourages the use of immutable data structures to avoid side effects.

Python has immutable types like tuples and sets, and immutable data can be used to prevent unintended changes.

6. Generator Functions and Iterators:

Generator functions (`**yield**` keyword) and iterators (`**iter()**` and `**next()**`) allow lazy evaluation and can be used for efficient handling of sequences and data streams.

Conclusion:

Functional programming concepts in Python enable a more declarative and expressive coding style. While Python is not a purely functional language, it supports functional programming paradigms, allowing developers to write code that is more concise, reusable, and easier to reason about in certain scenarios. Embracing functional programming can lead to cleaner and more modular code, especially in cases where immutability and higher-order functions are beneficial.

Chapter 13
Real-World Project

Building a Command-Line Tool

Building a command-line tool in Python involves creating an application that can be executed from the terminal or command prompt, accepting user input as arguments or options. Here's a basic overview of building a command-line tool using Python:

1. Choose a Framework or Library (Optional):

a. **Argparse (Standard Library):**

Python's `argparse` module is part of the standard library and allows parsing command-line arguments and options.

b. Click, docopt, Fire, etc. (Third-party Libraries):

These libraries simplify building command-line interfaces in Python and provide additional features for argument parsing and handling.

2. Define the Command-Line Interface:

Using `argparse`:

Define a parser and add arguments and options:

```
import argparse
parser = argparse.ArgumentParser(description='Description of your command-line tool')
parser.add_argument('arg1', help='Description of argument 1')
parser.add_argument('--option', help='Description of an optional argument', default='default_value')
```

Using other libraries:

Different libraries have their syntax for defining commands and options. Refer to their documentation for specific usage.

3. Implement Functionality:

Define functions that correspond to the command-line operations.

```
def run_command(arg1, option):
# Implement functionality using provided arguments and options
    print(f'Executing command with arg1: {arg1} and option: {option}')
```

4. Parse Command-Line Arguments and Execute:

Using `argparse`:

Parse arguments and execute corresponding functions based on user input:

```
args = parser.parse_args()
run_command(args.arg1, args.option)
```

Using other libraries:

Execute commands or call functions based on user input according to the library's documentation.

5. Packaging and Distribution (Optional):

Package your tool using tools like `**setuptools**` or `**pyinstaller**` to create distributable packages or executable files.

Example (Using `argparse`):

```
import argparse
def run_command(arg1, option):
    print(f'Executing command with arg1: {arg1} and option: {option}')
def main():
    parser = argparse.ArgumentParser(description='Description of
    your command-line tool')
    parser.add_argument('arg1', help='Description of argument 1')
    parser.add_argument('--option', help='Description of an optional
    argument', default='default_value')
    args = parser.parse_args()
    run_command(args.arg1, args.option)
if __name__ == "__main__":
    main()
```

Running the Tool:

Save the script and execute it from the terminal or command prompt, passing required arguments and options:

```
python script_name.py arg_value --option option_value
```

Conclusion:

Building a command-line tool in Python involves defining a command-line interface, parsing arguments, implementing functionality, and handling user input. Different libraries or frameworks offer varying levels of convenience and features for building robust and user-friendly command-line interfaces. Choose a suitable approach based on the complexity and requirements of your tool.

Developing a Web Application using Django

Developing a web application using Django involves setting up the Django framework, defining models, views, templates, and configuring URLs to create a fully functional web application. Here are the steps to create a basic web application using Django:

1. Install Django:
Install Django using pip:
pip install django

2. Create a Django Project:
Use the Django command-line tool to create a new project:
django-admin startprojectproject_name

3. Create an App:
In Django, an app is a web application. Create an app within your project:
cd project_name
python manage.py startappmyapp

4. Define Models:
Edit the **`models.py`** file in your app directory (`myapp`) to define data models using Django's ORM (Object-Relational Mapping):

```
# Example model
from django.db import models
class Item(models.Model):
    name = models.CharField(max_length=100)
    description = models.TextField()
    def __str__(self):
        return self.name
```

5. Create Database Tables:
Run database migrations to create database tables based on your models:
python manage.py makemigrations
python manage.py migrate

6. Define Views:

Create views in your app's **`views.py`** file to handle HTTP requests and generate responses:

```
# Example view
from django.shortcuts import render
from .models import Item
def item_list(request):
    items = Item.objects.all()
    return render(request, 'item_list.html', {'items': items})
```

7. Create Templates:

Create HTML templates in the **`templates`** directory within your app to render dynamic content:

Example template (`item_list.html`):

```
<!DOCTYPE html>
<html>
<head>
    <title>Item List</title>
</head>
<body>
<h1>Items:</h1>
<ul>
{% for item in items %}
    <li>{{ item.name }}</li>
{% endfor %}
</ul>
</body>
</html>
```

8. Configure URLs:

Define URL patterns to map views in your app's **`urls.py`** file:

```
# Example URL configuration in your app's urls.py
from django.urls import path
from .views import item_list
urlpatterns = [
path('items/', item_list, name='item_list'),
]
```

9. Run the Development Server:
Start the Django development server:
python manage.py runserver

10. Access the Application:
Visit `http://127.0.0.1:8000/items/` in your web browser to access the application.

Conclusion:
This is a basic guide to create a web application using Django. Django offers many features like authentication, admin panel, form, etc., which can be incorporated into your application based on your requirements. Django's documentation provides in-depth information on various aspects of building web applications using Django.

Date Analysis and Visualization Project

A data analysis and visualization project involves analyzing datasets to gain insights and presenting those findings through visualizations. Python offers various libraries like Pandas, Matplotlib, Seaborn, and others, which are commonly used for data analysis and visualization. Here's a high-level overview of creating a data analysis and visualization project using Python:

1. **Define the Problem and Goals:**
   Identify the problem statement and set goals for what insights or conclusions you aim to derive from the data.
2. **Acquire and Preprocess Data:**

- Data Collection: Obtain the dataset from reliable sources or databases.
- Data Cleaning: Handle missing values, outliers, and inconsistencies in the data. Convert data types if necessary.

3. **Exploratory Data Analysis (EDA):**
   Use Pandas to explore and understand the data:

- Descriptive Statistics: Summary statistics, distributions, etc.
- Data Visualization: Create basic plots to visualize relationships and trends in the data.

A data analysis and visualization project involves analyzing datasets to gain insights and presenting those findings through visualizations. Python offers various libraries like Pandas, Matplotlib, Seaborn, and others, which are commonly used for data analysis and visualization. Here's a high-level overview of creating a data analysis and visualization project using Python:

4. Data Mainpulation and Transformation:
Perform necessary transformations or aggregations on the data using Pandas or NumPy:
- Filtering, sorting, grouping, merging datasets, etc.

5. Advanced Analysis:
Conduct in-depth analysis or apply statistical methods to derive insights:
Correlation analysis, regression, hypothesis testing, etc.

6. Visualization:
Use Matplotlib, Seaborn, or other libraries to create informative and visually appealing plots:
- Scatter plots, histograms, bar plots, heatmaps, etc.

7. Storytelling and Presentation:
- Organize the insights gained into a coherent narrative or story.
- Create a report or presentation using Jupyter Notebooks, PowerPoint, or other tools to convey findings effectively.

 # Example Workflow (using Python Libraries):

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# 1. Data Acquisition and Preprocessing
data = pd.read_csv('dataset.csv')
# Data cleaning, handling missing values, etc.

# 2. Exploratory Data Analysis (EDA)
# Descriptive statistics
print(data.describe())
```

```
# Data visualization
plt.figure(figsize=(8, 6))
sns.scatterplot(x='feature1', y='feature2', data=data)
plt.title('Relationship between Feature 1 and Feature 2')
plt.show()

# 3. Data Manipulation and Transformation
# Perform data transformations or aggregations

# 4. Advanced Analysis
# Apply statistical methods or conduct in-depth analysis

# 5. Visualization
plt.figure(figsize=(8, 6))
sns.histplot(data['feature3'], bins=20)
plt.title('Distribution of Feature 3')
plt.show()

# 6. Storytelling and Presentation
# Organize findings into a report or presentation
```

Conclusion:

A data analysis and visualization project in Python involves several stages from data acquisition to storytelling. Python's libraries offer a robust ecosystem for data analysis, manipulation, visualization, and reporting, enabling comprehensive exploration and communication of insights from the data. Adjust the workflow based on your specific project requirements and data characteristics.

Chapter 14
Appendices

Python 2 vs. Python 3

Python 2 and Python 3 are two different versions of the Python programming language. Here are some key differences between the two:

Python 2:

- **Legacy Version:** Python 2 was released in 2000 and has been the standard version of Python for many years.
- **End of Life:** Python 2 reached its end-of-life on January 1, 2020, and is no longer officially maintained.
- **Print Statement:** In Python 2, the `print` statement is used without parentheses: `print "Hello, World!"`.
- **Handling:** String handling and Unicode support are different from Python 3, which sometimes led to encoding/decoding issues.
- **Division:** In Python 2, dividing two integers would return an integer if both numbers were integers (`5 / 2` would return `2`). To get a floating-point result, you'd have to use `from __future__ import division` or use `5.0 / 2` to force floating-point division.
- **Library Support:** While many libraries were developed for Python 2, newer libraries and updates are primarily targeted at Python 3.

Python 3:

- **Current Version:** Python 3 was released in 2008 and is the actively developed and maintained version.
- **Print Function:** In Python 3, the `print` statement was replaced with a print function requiring parentheses: `print("Hello, World!")`.
- **Unicode Handling:** Python 3 handles strings as Unicode by default, simplifying handling of text and characters.
- **Division:** In Python 3, division of two integers returns a float by default (`5 / 2` would return `2.5`). Integer division can be done using `5 // 2`.
- **Library Support:** Newer libraries and updates are focused on Python 3, while some older libraries might not be fully compatible.

Which Version to Use:

- **Python 3:** It's strongly recommended to use Python 3 for all new projects. It offers many improvements over Python 2, including better Unicode support, cleaner syntax, and ongoing support from the Python community.
- **Migration:** For projects still using Python 2, it's advisable to migrate to Python 3, as Python 2 is no longer maintained, making it vulnerable to security risks and lacking updates or new features.

Conclusion:

Python 3 is the current and recommended version of Python for all new projects. It offers numerous improvements and is the version that continues to receive active support and updates from the Python Software Foundation. If possible, it's highly recommended to migrate any existing Python 2 code to Python 3.

Python Resources

Certainly! Here's a list of valuable Python resources that can help beginners and advanced users alike to learn, practice, and deepen their understanding of Python:

Online Learning Platforms:

1. **Coursera:** Offers Python courses from top universities like University of Michigan, Rice University, etc.
2. **edX:** Provides Python courses from MIT, Harvard, and other prestigious institutions.
3. **Udemy:** Hosts numerous Python courses catering to different skill levels and specializations.
4. **Codecademy:** Provides an interactive platform to learn Python through coding exercises.
5. **SoloLearn:** Offers free Python courses and a mobile app for learning on-the-go.

Books:

1. **"Automate the Boring Stuff with Python" by Al Sweigart:** Great for beginners and covers practical applications.

2. **"Python Crash Course" by Eric Matthes:** Covers Python fundamentals and project-based learning.
3. **"Fluent Python" by Luciano Ramalho:** For more experienced Python programmers wanting to deepen their understanding.
4. **"Effective Python: 90 Specific Ways to Write Better Python" by Brett Slatkin:** Focuses on best practices and idiomatic Python coding.

Documentation and Tutorials:

1. **Python Official Documentation:** An essential resource for understanding the Python language and its libraries.
2. **Real Python:** Offers tutorials, articles, and courses suitable for all skill levels.
3. **GeeksforGeeks Python:** Provides tutorials, code snippets, and articles for Python programming.
4. **W3Schools Python Tutorial:** Beginner-friendly tutorials covering Python basics.

Practice Platforms:

1. **LeetCode:** Offers coding challenges to practice Python and other programming languages.
2. **HackerRank:** Provides coding challenges and exercises for Python.
3. **Exercism:** Focuses on improving coding skills through mentoring and community collaboration.

Community and Forums:

1. **Stack Overflow:** An active community to ask and answer programming questions related to Python.
2. **Reddit (r/learnpython, r/python):** Subreddits where Python enthusiasts share resources and help each other.

Miscellaneous:

1. **GitHub Repositories:** Explore Python projects on GitHub to learn from others' code and contribute.
2. **PyPI (Python Package Index):** Repository for Python libraries - explore and use various Python packages.
3. **YouTube Channels:** Channels like Corey Schafer, Sentdex, and freeCodeCamp.org offer Python tutorials and guides.

Conclusion:

These resources cater to various learning styles and levels of expertise. Depending on your preferences and goals, exploring a combination of these resources can significantly enhance your Python skills and knowledge. Remember that consistent practice and application are key to mastering Python programming.

Glossary of Terms

Certainly! Here's a glossary of commonly used terms in Python programming:

A

Algorithm: A sequence of well-defined instructions or steps to solve a specific problem or perform a task.

B

Boolean: A data type that represents two values: `True` or `False`.

C

Class:A blueprint or template for creating objects that define attributes and behaviors.

Conditional Statements: Statements that execute based on certain conditions (`if`, `else`, `elif`) in the code.

CSV (Comma-Separated Values): A file format for storing tabular data where each line represents a row, and columns are separated by commas.

D

Decorator: A function that modifies the behavior of another function without directly changing its code.

Dictionary: A data structure in Python that stores key-value pairs `{key: value}`.

E

Exception: An error that occurs during the execution of a program due to unforeseen circumstances.

Expression: A combination of values, variables, and operators that evaluates to a single value.

F

Function: A block of reusable code that performs a specific task.

I

Immutable: An object whose state cannot be modified after it is created (e.g., tuples and strings).

L

List: A mutable data structure in Python that stores a collection of items in a specific order `[item1, item2, ...]`.

M

Module: A file containing Python code that can be imported and used in other Python programs.

O

Object: An instance of a class that encapsulates data and behavior.

Operator: Symbols or keywords that perform operations on operands (e.g., `+`, `-`, `*`, `/`).

P

Package: A collection of modules that can be imported together.

Polymorphism: The ability of objects to take on different forms or behaviors based on their context.

Pythonic: Writing code that follows the idiomatic style and best practices of Python programming.

R

Recursion: A technique where a function calls itself to solve a problem by breaking it down into smaller subproblems.

Regular Expression: A sequence of characters defining a search pattern to match patterns in strings.

S

Set: A collection of unique elements `{item1, item2, ...}`.

String: A sequence of characters enclosed in quotes `' '` or `" "`.

T

Tuple: An immutable ordered collection of elements `(item1, item2, ...)`.

Type Casting/Conversion: Changing the data type of an object to another data type.

V

Variable: A name that refers to a value stored in memory.

This glossary covers some fundamental terms used in Python programming. Understanding these terms is essential for learning and effectively writing Python code.

Lab Practice

Lab Practice

1. /*Simple program to display "HelloWorld" */

```
print("Hello, world!")
```

2. /*Maximum of two numbers in Python */

```
# Define two numbers num1 = 10 num2 = 20
# Find the maximum of the two numbers
maximum = max(num1, num2)
# Display the maximum
print("The maximum number is:", maximum)
```

3./*Python Program for factorial of a number*/

```
def factorial_iterative(n):
    factorial = 1
    if n < 0:
        return "Factorial does not exist for negative numbers"
    elif n == 0:
        return 1
    else:
        for i in range(1, n + 1):
            factorial *= i
        return factorial
# Input the number
number = int(input("Enter a number: "))
result=factorial_iterative(number)
print("Factorial of", number, "is:", result)
```

4. /*Python Program for factorial of a number using recursive*/

```
def factorial_recursive(n):
    if n < 0:
        return "Factorial does not exist for negative numbers"
    elif n == 0 or n == 1:
        return 1
    else:
        return n * factorial_
        recursive(n - 1)
# Input the number
number = int(input("Enter a number: "))
result=factorial_recursive(number)
print("Factorial of", number, "is:", result)
```

5. /*Python Program for factorial of a number using recursive*/

```
def simple_interest(principal, rate, time):
# Simple interest formula: SI = #(P * R *T) / 100
interest = (principal * rate * time) / 100
return interest
# Input principal amount, rate of interest, and time period
principal_amount=float(input("Enter the principal amount: "))
interest_rate=float(input("Enter the interest rate: "))
time_period= float(input("Enter the time period (in years): "))
# Calculate the simple interest
simple_interest_amount=simple_interest(principal_amount, interest_rate, time_period)
# Display the simple interest
print("Simple Interest:", simple_interest_amount)
```

6. /*Python Program for compound interest*/

```
def
compound_interest(principal, rate, time, frequency):
# Compound interest formula:
A = P * (1 + r/n)^(nt)
    amount = principal * (pow((1 + rate / (frequency * 100)), (frequency * time)))
    interest = amount - principal
return interest
# Input principal amount, rate of interest, time period, and frequency of compounding
principal_amount=float(input(" Enter the principal amount: "))
interest_rate=float(input("Enter the interest rate: "))
time_period= float(input("Enter the time period (in years): "))
compounding_frequency=int(input("Enter the frequency of compounding per year: "))
# Calculate the compound interest
compound_interest_amount = compound_interest(principal_mount, interest_rate, time_period,
compounding_frequency)
# Display the compound interest
print("Compound Interest:", compound_interest_amount)
```

7. /*Python Program for Program to find area of a circle*/

```
def
calculate_circle_ara(radius):
# Formula to calculate the area of a circle: A = π * r^2
    pi = 3.14159
# Approximation of Pi
    area = pi * (radius ** 2)
    return area
# Input radius of the circle
radius = float(input("Enter the radius of the circle: "))
# Calculate the area of the circle
circle_area=calculate_circle_ara(radius)
# Display the area of the circle
print(f"The area of the circle with radius {radius} is: {circle_area}")
```

8. /*Python Program for n-th Fibonacci number*/

```
def fibonacci_recursive(n):
    if n <= 0:
        return "Invalid input. Please enter a positive integer."
    elif n == 1:
        return 0
    elif n == 2:
        return 1
    else:
        return
fibonacci_recursive(n - 1) + fibonacci_recursive(n - 2)
# Input the value of 'n'
n_value = int(input("Enter the value of 'n' to find the nth Fibonacci number: "))
```

```
# Calculate and display the nth Fibonacci number using recursion
result_recursive=fibonacci_recursive(n_value)
print(f"The {n_value}th Fibonacci number using recursive approach is: {result_recursive}")
```

9. /*Program to print ASCII Value of a character*/

```
# Input a character
character = input("Enter a character: ")
# Get the ASCII value of the character
ascii_value = ord(character)
# Display the ASCII value
print(f"The ASCII value of '{character}' is: {ascii_value}")
```

10. /*Python Program to find sum of array*/

```
def sum_of_array(arr):
# Initialize the sum to zero
    array_sum = 0

# Iterate through the array and add each element to the sum
    for element in arr:
        array_sum += element
    return array_sum
# Input the array elements
arr = list(map(int, input("Enter the elements of the array separated by space: ").split()))
# Calculate the sum of the array elements
result = sum_of_array(arr)
# Display the sum of the array
print("The sum of the array elements is:", result)
```

11. /*Python Program to find largest element in an array*/

```
def find_largest_element(arr):
    if not arr:
        return "Array is empty"
# Initialize the maximum element as the first element of the array
max_element = arr[0]
# Iterate through the array to find the largest element
    for element in arr:
        if element >max_element:
            max_element=element
    return max_element
# Input the array elements
arr = list(map(int, input("Enter the elements of the array separated by space: ").split()))
# Find the largest element in the array
largest=find_largest_element(arr)
# Display the largest element in the array
print("The largest element in the array is:", largest)
```

12. /*Python Program for array rotation*/

```
def rotate_array(arr, rotation):
    length = len(arr)
    rotation %= length # Adjust rotation if it's greater than the array length
```

```
# Rotate the array elements
arr[:] = arr[-rotation:] + arr[:-rotation]
    return arr
# Input the array elements
arr = list(map(int, input("Enter the elements of the array separated by space: ").split()))
# Input the number of rotations
num_rotations=int(input("Enter the number of rotations: "))
# Perform array rotation
rotated_array=rotate_array(arr, num_rotations)
# Display the rotated array
print("Array after rotation:", rotated_array)
```

13. /*Python Program to Split the array and add the first part to the end*/

```
def split_and_add(arr, split_position):
    if split_position< 0 or split_position>= len(arr):
        return "Invalid split position"
# Split the array and add the first part to the end
    return arr[split_position:] + arr[:split_position]
# Input the array elements
arr = list(map(int, input("Enter the elements of the array separated by space: ").split()))
# Input the split position
split_position = int(input("Enter the split position: "))
# Perform splitting and adding the first part to the end
result_array= split_and_add(arr, split_position)
# Display the resulting array
print("Array after splitting and adding the first part to the end:", result_array)
```

14. /*Python Program to check if given array is Monotonic*/

```
def is_monotonic(arr):
    increasing = decreasing = True
# Check for non-increasing
    for i in range(1, len(arr)):
        if arr[i] >arr[i - 1]:
            decreasing = False
            break
# Check for non-decreasing
    for i in range(1, len(arr)):
        if arr[i] <arr[i - 1]:
            increasing = False
            break
# If either increasing or decreasing is True, array is monotonic
    return increasing or decreasing
# Input the array elements
arr = list(map(int, input("Enter the elements of the array separated by space: ").split()))
# Check if the array is monotonic
if is_monotonic(arr):
    print("The array is monotonic")
else:
    print("The array is not monotonic")
```

15. /*Python program to interchange first and last elements in a list*/

```
def interchange_first_last(lst):
    if len(lst) < 2:
        return "List should have at least two elements for interchange"
    # Swap the first and last elements using tuple unpacking
lst[0], lst[-1] = lst[-1], lst[0]
    return lst
# Input the list elements
input_list = list(map(int, input("Enter the elements of the list separated by space: ").split()))
# Interchange the first and last elements in the list
result_list=interchange_first_last(input_list[:])
# Display the list after interchange
print("List after interchanging first and last elements:", result_list)
```

16. /*Python program to swap two elements in a list*/

```
def swap_elements(lst, idx1, idx2):
    if 0 <= idx1 <len(lst) and 0 <= idx2 <len(lst):
# Swap the elements at idx1 and idx2
lst[idx1], lst[idx2] = lst[idx2], lst[idx1]
    return lst
else:
    return "Invalid indices. Please enter valid indices within the list range."
# Input the list elements
input_list = list(map(int, input("Enter the elements of the list separated by space: ").split()))
# Input the indices to swap
index1 = int(input("Enter the first index to swap: "))
index2 = int(input("Enter the second index to swap: "))
# Swap elements at specified indices in the list
result_list=swap_elements(input_list[:], index1, index2)
# Display the list after swapping elements
print("List after swapping elements:", result_list)
```

17. /*Python program to find second largest number in a list*/

```
def second_largest(lst):
    if len(lst) < 2:
        return "List should have at least two elements"
max_num = max(lst[0], lst[1])
second_max = min(lst[0], lst[1])
    for i in range(2, len(lst)):
        if lst[i] >max_num:
second_max = max_num
max_num = lst[i]
eliflst[i]>second_max and lst[i] != max_num:
second_max = lst[i]
    if second_max == float('-inf'):
 return "There is no second largest element" else: return second_max
```

```
# Input the list elements
input_list = list(map(int, input("Enter the elements of the list separated by space: ").split()))
# Find the second largest number in the list
result=second_largest(input_list)
# Display the second largest number in the list
print("The second largest number in the list is:", result)
```

18. /*Python program to print even numbers in a list*/

```
def print_even_numbers(lst):
    even_numbers = [num for num in lst if num % 2 == 0]
    if len(even_numbers) == 0:
        return "No even numbers found in the list"
    else:
        return even_numbers
# Input the list elements
input_list = list(map(int, input("Enter the elements of the list separated by space: ").split()))
# Print the even numbers from the list
result=print_even_numbers(input_list)
# Display the even numbers
if isinstance(result, list):
    print("Even numbers in the list are:", result)
else:
    print(result)
```

19. /*Remove multiple elements from a list in Python*/

```
# Original list
original_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
# Elements to remove (e.g., removing elements from index 2 to 5)
start_index = 2
end_index = 6 # Exclude the last index you want to remove
# Remove elements using slicing
updated_list=original_list[:start_index]+original_list[end_index:]
print("Updated list after removing elements:", updated_list)
```

20. /*Program to print duplicates from a list of integers*/

```
def find_duplicates(lst):
    frequency = {}
    duplicates = []
    for num in lst:
        if num in frequency:
            frequency[num] += 1
        else:
            frequency[num] = 1
    for key, value in frequency.items():
        if value > 1:
duplicates.append(key)
    return duplicates
# Example list with duplicates
input_list = [1, 2, 2, 3, 4, 4, 5, 5, 6, 7, 8, 9, 9, 9]
```

```
# Find and print duplicates in the list
duplicate_values=find_duplicates(input_list)
if duplicate_values:
    print("Duplicate values in the list are:", duplicate_values)
else:
    print("No duplicates found in the list")
```

21. /*Python program to find Cumulative sum of a list*/

```
def cumulative_sum(lst):
cumulative_result = []
    cumulative = 0
    for num in lst:
        cumulative += num
cumulative_result.append(cumulative)
    return cumulative_result
# Example list of integers
input_list = [1, 2, 3, 4, 5]
# Calculate and print the cumulative sum of the list
result=cumulative_sum(input_list)
print("Cumulative sum of the list:", result)
```

22. /*Sort the values of first list using second list*/

```
def sort_list_by_second_list(list1, list2):
    combined= list(zip(list2, list1))
combined.sort()
    sorted_list1 = [element[1] for element in combined]
    return sorted_list1
# Example lists
first_list = [3, 1, 5, 4, 2]
second_list = [9, 7, 2, 8, 3]
# Sort the values of the first list using the second list
sorted_values=sort_list_by_second_list(first_list, second_list)
print("Sorted values of the first list using the second list:", sorted_values)
```

23. /*Python program to add two Matrices*/

```
def add_matrices(matrix1, matrix2):
    if len(matrix1) != len(matrix2) or len(matrix1[0]) != len(matrix2[0]):
    return "Matrices should have the same dimensions for addition"
result_matrix = []
    for i in range(len(matrix1)):
        row = []
        forj in range(len(
            matrix1[0])):
row.append(matrix1[i][j] + matrix2[i][j])
result_matrix.append(row)
    return result_matrix
# Example matrices
matrix1 = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
matrix2 = [
    [9, 8, 7],
    [6, 5, 4],
    [3, 2, 1]
]
```

```
# Add the two matrices
result = add_matrices(matrix1, matrix2)
# Display the result of matrix addition
if isinstance(result, str):
    print(result)
else:
    print("Resultant Matrix after addition:")
    for row in result:
    print(row)
```

24. /*Python program to multiply two matrices*/

```
def multiply_matrices(matrix1, matrix2):
    rows_m1 = len(matrix1)
    cols_m1 = len(matrix1[0])
    rows_m2 = len(matrix2)
    cols_m2 = len(matrix2[0])

    if cols_m1 != rows_m2:
        return "Cannot multiply matrices. Number of columns in the first matrix should be equal to the number of rows in the second matrix."
result_matrix = [[0 for _ in range(cols_m2)] for _ in range(rows_m1)]
    for i in range(rows_m1):
        for j in range(cols_m2):
for k in range(cols_m1):
result_matrix[i][j] += matrix1[i][k] * matrix2[k][j]
    return result_matrix
# Example matrices
matrix1 = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
matrix2 = [
    [9, 8, 7],
    [6, 5, 4],
    [3, 2, 1]
]
# Multiply the two matrices
result=multiply_matrices(matrix1, matrix2)
# Display the result of matrix multiplication
if isinstance(result, str):
    print(result)
else:
    print("Resultant Matrix after multiplication:")
    for row in result:
    print(row)
```

25. /*Transpose a matrix in Single line in Python*/

```
# Example matrix
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
# Transpose the matrix in a single line using list comprehension and zip
transpose_matrix = [list(row) for row in zip(*matrix)]
# Display the transposed matrix
for row in transpose_matrix:
    print(row)
```

- **zip(*matrix) transposes the matrix by unpacking matrix into arguments for zip(). It effectively rearranges rows into columns and columns into rows.**
- **list(row) for row in zip(*matrix) uses list comprehension to convert the resulting transposed tuples into lists.**

26. /* Python program to check if a string is palindrome or not*/

```
def is_palindrome(s):
# Removing spaces and converting to lowercase for case-insensitive comparison
    s = s.replace(" ", "").lower()
# Compare the original string with its reverse
    return s == s[::-1]
# Input from the user
user_input = input("Enter a string: ")
if is_palindrome(user_input):
    print("The string is a palindrome.")
else:
    print("The string is not a palindrome.")
```

27. /* Python program to check if a Substring is Present in a Given String*/

```
defis_substr_present(main_str, substr):
    # Check if the substring is present in the main string
    return substr in main_str
# Input from the user
main_string = input("Enter the main string: ")
substring = input("Enter the substring to check: ")
ifis_substr_present(main_str, substr):
    print(f"The substring '{substring}' is present in the main string.")
else:
    print(f"The substring '{substring}' is not present in the main string.")
```

28. /* Python program to find the frequency of each word in a given string*/

```
def word_frequency(string):
# Removing punctuation and converting to lowercase
    string = ''.join(char.lower() if char.isalnum() or char.isspace() else '' for char in string)
    # Split the string into words
    words = string.split()
# Count the frequency of each word using a dictionary
    frequency = {}
    for word in words:
        if word in frequency:
            frequency[word] += 1
        else:
            frequency[word] = 1
    return frequency
```

```
# Input from the user
input_string = input("Enter a string: ")
# Get the word frequency
frequency_dict = word_frequency(input_string)
# Print the word frequency
print("Word Frequency:")
for word, count in frequency_dict.items():
    print(f"{word}: {count}")
```

29. /* Python program to print even length words in a string*/

```
def print_even_length_words(string):
# Removing punctuation and converting to lowercase
    string = ''.join(char.lower() if char.isalnum() or char.isspace() else ' ' for char in string)
# Split the string into words
    words = string.split()
# Print even-length words
    print("Even-length words:")
    for word in words:
        if len(word) % 2 == 0:
            print(word)
# Input from the user
input_string = input("Enter a string: ")
# Print even-length words in the string
print_even_length_words(input_string)
```

30. /* Python program to accept the strings which contains all vowels*/

```
def _all_vowels(s):
    # Convert the string to lowercase for case-insensitive comparison
    s = s.lower()
# Check if the string contains all vowels
    return all(vowel in s for vowel in 'aeiou')
# Input from the user
input_string = input("Enter a string: ")
# Check if the string contains all vowels
if _all_vowels(input_string):
    print("The string contains all vowels.")
else:
    print("The string does not contain all vowels.")
```

31. /* Program to remove a key from dictionary*/

```
def remove_key(dictionary, key_to_remove):
    # Use pop() to remove the specified key
    dictionary.pop(key_to_remove, None)
# Example usage:
my_dict = {'a': 1, 'b': 2, 'c': 3}
# Remove key 'b'
remove_key(my_dict, 'b')
# Print the updated dictionary
print(my_dict)
```

32. /* Python program Merging two Dictionaries*/

```
def merge_dicts(dict1, dict2):
```

```
    # Create a copy of dict1 to avoid modifying it directly
merged_dict = dict1.copy()
# Update the copy with the contents of dict2
merged_dict.update(dict2)
    return merged_dict
# Example usage:
    dict1 = {'a': 1, 'b': 2}
    dict2 = {'b': 3, 'c': 4}
# Merge the dictionaries
result_dict = merge_dicts(dict1, dict2)
# Print the merged dictionary
print(result_dict)
```

33. /* Python program to Convert key-values list to flat dictionary*/

```
def list_to_dict(key_value_list):
# Initialize an empty dictionary
    flat_dict = {}
    # Iterate through the list in pairs
    for key, value in key_value_list:
# Add key-value pairs to the dictionary
    flat_dict[key] = value
    return flat_dict
# Example usage:
key_value_list = [('a', 1), ('b', 2), ('c', 3)]
# Convert the list to a flat dictionary
result_dict = list_to_dict(key_value_list)
# Print the resulting dictionary
print(result_dict)
```

34. /* Python program Remove all duplicates words from a given sentence*/

```
def remove_duplicates(sentence):
# Split the sentence into words
    words = sentence.split()
# Use a set to store unique words
unique_words = set()
# List to store the result
result_words = []
# Iterate through the words
    for word in words:
     # Check if the word is not in the set
      if word not in unique_words:
# Add the word to the set and result list
       unique_words.add(word)
    result_words.append(word)
    # Join the result list into asentence
result_sentence=''.join(result_words)
    return result_sentence
# Example usage:
input_sentence = "This is a sample sentence with some duplicate words. This is a sample sentence."
# Print the merged dictionary
print(result_dict)
```

33. /* Python program to Convert key-values list to flat dictionary*/

```
def list_to_dict(key_value_list):
# Initialize an empty dictionary
flat_dict = {}
    # Iterate through the list in pairs
    for key, value in key_value_list:
# Add key-value pairs to the dictionary
flat_dict[key] = value
    return flat_dict
# Example usage:
key_value_list = [('a', 1), ('b', 2), ('c', 3)]
# Convert the list to a flat dictionary
result_dict = list_to_dict(key_value_list)
# Print the resulting dictionary
print(result_dict)
```

34. /* Python program Remove all duplicates words from a given sentence*/

```
def remove_duplicates(sentence):
    # Split the sentence into words
    words = sentence.split()
    # Use a set to store unique words
    unique_words = set()
# List to store the result
result_words = []
# Iterate through the words
    for word in words:
        # Check if the word is not in the set
        if word not in unique_words:
# Add the word to the set and result list
unique_words.add(word)
result_words.append(word)
    # Join the result list into a sentence
result_sentence=''.join(result_words)
    return result_sentence
# Example usage:
input_sentence = "This is a sample sentence with some duplicate words. This is a sample sentence."
# Remove duplicates
result_sentence = remove_duplicates(input_sentence)
# Print the result
print(result_sentence)
```

35. /* Python program to convert number into words */

First, you need to install the **inflect** library if you haven't already:

pip install inflect

```
import inflect
def number_to_words(number):
    p = inflect.engine()
    return p.number_to_words(number)
```

```
# Example usage:
input_number = 123456

# Convert the number to words
result=number_to_words(input_number)
# Print the result
print(result)
```

36. /* Python program Convert a list of Tuples into Dictionary*/

```
def list_of_tuples_to_dict(tuple_list):
    # Use dict() constructor to convert the list of tuples to a dictionary
    result_dict = dict(tuple_list)
    return result_dict
# Example usage:
tuple_list = [('a', 1), ('b', 2), ('c', 3)]
# Convert the list of tuples to a dictionary
result_dict = list_of_tuples_to_dict(tuple_list)
# Print the resulting dictionary
print(result_dict)
```

37. /* Python program Least Frequent Character in String*/

```
def least_freq_char(input_str):
# Create a dictionary to store the frequency of each character
char_frequency = {}
# Count the frequency of each character in the string
    for char in input_str:
      if char in char_frequency:
char_frequency[char] += 1
    else:
char_frequency[char] = 1
# Find the least frequent characterleast_frequent_char=min(char_frequency,key=char_frequency.get)
return least_frequent_char
my_string = "hello world"
result = least_freq_char(my_string)
print(f"The least frequent character is: {result}")
```

38. /* Python program Maximum frequency character in String*/

```
def max_freq_chr(input_str):
    # Create a dictionary to store the frequency of each character
    char_frequency = {}
# Count the frequency of each character in the string
    for char in input_str:
      if char in char_frequency:
       char_frequency[char]+= 1
    else:
      char_frequency[char] = 1
# Find the character with the maximum frequency
max_frequency_char =
```

```
max(char_frequency,key=char
_frequency.get)
 return max_frequency_char

my_string = "hello world"
result =
max_freq_chr(my_string)
print(f"The character with the
maximum frequency is:
{result}")
```

39. /* Python program to check if a string contains any special character*/

```
def
has_special_chrs(input_str):
    # Define a set of special
characters
special_characters =
set("!@#$%^&*()
_+=[]{}|;:\",.<>?/")
    # Check if the string
contains any special
characters
    for char in input_str: if char in
special_characters:
        return True
    return False
    my_string = "Hello! How are
you?"
result = has_special_chrs
(my_string)

if result:
    print("The string contains
special characters.")
else:
    print("The string does not
contain any special
characters.")
```

40. /* Python program to Generating random strings until a given string is generated*/

```
import random
import string

def gen_random_string(length):
    return
''.join(random.choice(string.asc
ii_letters + string.digits) for _ in
range(length))

def gen_until_targ(targ_string):
generated_string = ""
    attempts = 0
    while generated_string !=
target_string:
generated_string
gen_random_string(len(target_
string))
    attempts += 1
    print(f"Attempt {attempts}:
{generated_string}")

    print(f"\nTarget string
'{target_string}' generated after
{attempts} attempts.")

target_string = "Hello123"
gen_until_targ(targ_string)
```

41. /* Python program to Check if a given string is binary string or not*/

```
def is_binary_string(input_str):
    # Check if each character is either '0' or '1'
    for char in input_str:
        if char not in ('0', '1'):
            return False
    return True

binary_string = "101010101"

result = is_binary_string(binary_string)

if result:
    print("The string is a binary string.")
else:
    print("The string is not a binary string.")
```

42. /* Python program to find uncommon words from two Strings*/

```
def find_uncommon(str1, str2):
# Tokenize the strings into words
    words_str1 = set(str1.split())
    words_str2 = set(str2.split())

    # Find uncommon words
uncommon_words = words_str1.symmetric_difference(words_str2)

    return uncommon_words

string1 = "This is the first string"
string2 = "This is the second string with some different words"
result = find_uncommon(string1, string2)

print("Uncommon words:", result)
```

43. /* Python program to Check for URL in a String*/

```
import re
def contains_url(input_str):
# Regular expression to match URLs
url_pattern = re.compile(r'https?://\S+|www\.\S+')
# Search for the URL pattern in the input string
    match = re.search(url_pattern, input_str)
    return bool(match)
my_string = "Visit my website at https://www.example.com for more information."

if contains_url(my_string):
    print("The string contains a URL.")
else:
    print("No URL found in the string.")
```

44. /* Python program to find the sum of all items in a dictionary */

```
def sum_of_vals(dictionary):
    return sum(dictionary.values())
```

```
my_dict = {'a': 10, 'b': 20, 'c': 30, 'd': 40}
result = sum_of_val(my_dict)
print(f"The sum of all values in the dictionary is: {result}")
```

45. /* Python program to sort list of dictionaries by values– Using lambda function*/

```
# List of dictionaries
my_list_of_dicts = [
    {'name': 'lakshay', 'age': 30, 'score': 85},
    {'name': 'mohan', 'age': 25, 'score': 92},
    {'name': 'palak', 'age': 35, 'score': 78}
]
# Sort the list of dictionaries by the 'score' value using lambda function
sorted_list = sorted(my_list_of_dicts, key=lambda x: x['score'])
# Print the sorted list
for item in sorted_list:
    print(item)
```

46. /* Python program to Append Dictionary Keys and Valuesin dictionary*/

```
def apd_keys_vals(dict1, dict2):
result_dict = dict1.copy()
    for key, value in dict2.items():
result_dict[key] = value
return result_dict
# Example dictionaries
dict1 = {'a': 1, 'b': 2}
dict2 = {'c': 3, 'd': 4}
# Append keys and values from dict2 to dict1
result = apd_keys_vals (dict1, dict2)
print("Dictionary 1:", dict1)
print("Dictionary 2:", dict2)
print("Dictionary:", result)
```

47. /* Python program to Handling missing keys in Python dictionaries*/

```
my_dict = {'a': 1, 'b': 2}
key = 'c'
if key in my_dict:
    value = my_dict[key]
else:
    value = 'Key not found'

print(value)
```

48. /* Python dictionary with keys having multiple inputs*/

```
# Using tuples as keys
multi_input_dict = {('a', 1): 'Value1', ('b', 2): 'Value2', ('c', 3): 'Value3'}

# Accessing values using tuples as keys
print(multi_input_dict[('a', 1)]) # Output: Value1
print(multi_input_dict[('b', 2)]) # Output: Value2

# Using lists as keys
multi_input_dict_list = {['x', 10]: 'ValueX', ['y', 20]: 'ValueY', ['z', 30]: 'ValueZ'}
```

```
# Note: Lists cannot be used as dictionary keys because they are mutable
# You might get an error or unexpected behavior if you try to use a list as a key
# But you can convert the lists to tuples before using them as keys
list_key = tuple(['x', 10])
print(multi_input_dict_list[list_key]) # Output: ValueX
```

49. /* Check if binary representations of two numbers are anagram*/

To check if the binary representations of two numbers are anagrams, you can follow these steps:

1. Convert both numbers to binary strings.
2. Compare the binary strings for equality.

```
defanagrams(num1, num2):
# Convert numbers to binary strings
    binary_str1 = bin(num1)[2:]
    binary_str2 = bin(num2)[2:]
# Check if the sorted binary strings are the same
    return sorted(binary_str1) == sorted(binary_str2)
num1 = 7
num2 = 4
if anagrams(num1, num2):
    print(f"The binary representations of {num1} and {num2} are anagrams.")
else:
    print(f"The binary representations of {num1} and {num2} are not anagrams.")
```

50. /* Counting the frequencies in a list using dictionary in Python*/

```
def count_frequencies(input_list):
# Initialize an empty dictionary to store frequencies
frequency_dict = {}
# Iterate through the list
    for element in input_list:
        # If the element is already a key in the dictionary, increment its count
if element in frequency_dict:
frequency_dict[element] += 1
    # If the element is not a key, add it to the dictionary with a count of 1
else:
frequency_dict[element] = 1
return frequency_dict
my_list = [1, 2, 2, 3, 3, 3, 4, 4, 4, 4]
result = count_frequencies(my_list)
print("List:", my_list)
print("Frequencies:", result)
```

50. /* Python program to Find the size of a Tuple*/

```
import sys
def tuple_size(input):
    return sys.getsizeof(input)
```

```
my_tuple = (1, 2, 3, 'a', 'b', 'c', True, False, None)
# Get the size of the tuple
size = tuple_size(my_tuple)
print("Tuple:", my_tuple)
print("Size of the tuple:", size, "bytes")
```

51. /* Python program to Find Maximum and Minimum K elements in Tuple*/

```
import heapq def max_min_k_elements(in, k):
# Finding K largest elements
    max= heapq.nlargest(k, in)
# Finding K smallest elements
    min = heapq.nsmallest(k, in)
    return max, min

my_tuple = (3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5)
# Specify the value of K
k = 3
# Get the K largest and K smallest elements
max_elements, min_elements= max_min_k_elements(my_tuple, k)

print("Tuple:", my_tuple)
print(f"{k} Largest Elements:", max_elements)
print(f"{k} Smallest Elements:", min_elements)
```

52. /* Python program Remove Tuples of Length K*/

```
def rm_tuples_of_length_k(list, k):
    return [tup for tup in tuple_list if len(tup) != k]
list_of_tuples = [(1, 2), ('a', 'b', 'c'), (3, 4, 5), ('x', 'y'), ('p', 'q', 'r')]
# Specify the length K
k = 3
# Remove tuples of length K
result = rm_tuples_of_length_k(list_of_tuples, k)

print("Original List of Tuples:", list_of_tuples)
print(f"Tuples with length {k} removed:", result)
```

53. /* Create a list of tuples from given list having number and its cube in each tuple*/

```
deftuples_cube(input_list):
# Use a list comprehension to create tuples with number and its cube
tuples_list = [(num, num ** 3) for num in input_list]
    return tuples_list
original_list = [1, 2, 3, 4, 5]
# Create a list of tuples with each tuple containing a number and its cube
result=tuples_cube(original_list)
print("Original List:", original_list)
print("List of Tuples (Number, Cube):", result)
```

54. /* Python program Join Tuples if similar initial element*/

```
def
join_tuples_with_similar_initial
_element(tuple_list):
# Use a dictionary to group tuples by their initial elements
grouped_tuples = {}
    for tup in tuple_list:
initial_element = tup[0]
    if initial_element in
grouped_tuples:
grouped_tuples[initial_element]
.append(tup)
    else:
grouped_tuples[initial_element]
= [tup]

    # Concatenate tuples with similar initial elements
result_list=[tuple(sum(grouped
_tuples[key], ())) for key in
grouped_tuples]

    return result_list

# Example usage
original_list = [(1, 'a'), (2, 'b'), (1,
'c'), (3, 'd'), (2, 'e'), (4, 'f')]

# Join tuples with similar initial elements
result_list =
join_tuples_with_similar_initial
_element(original_list)

print("Original List of Tuples:",
original_list)
print("Joined Tuples with
Similar Initial Elements:",
result_list)
```

55. /* Python program Extract digits from Tuple list*/

```
def extract_digits(tuple_list):
    # Use list comprehension to extract digits from each tuple
digit_list = [''.join(filter(str.isdigit,
str(item))) for tup in tuple_list for
item in tup]

    return digit_list
tuple_list = [(1, 'abc', 23), ('x', 45,
'yz'), (67, 'pqr', '89')]
# Extract digits from the tuple list
result= extract_digits(tuple_list)
print("Original List of Tuples:",
tuple_list)
print("Extracted Digits:", result)
```

56. /* Python Program for Binary Search */

```
def binary_search(arr, low,
high, target):
    if low <= high:
        mid = (low + high) // 2
        # Check if target is present at the middle
    if arr[mid] == target:
        return mid
# If target is smaller, search in the left half
elif arr[mid] > target:
    return binary_search(arr,
low, mid - 1, target)
```

```
# If target is larger, search in the right half
        else:
            return binary_search(arr, mid + 1, high, target)

    else:
# Element is not present in the array
        return -1
arr = [2, 3, 4, 10, 40]
target = 10
result = binary_search(arr, 0, len(arr) - 1, target)
if result != -1:
    print(f"Element {target} is present at index {result}")
else:
    print(f"Element {target} is not present in the array")
```

57. /* Python Program for Linear Search*/

```
def linear_search(arr, target):
    for i in range(len(arr)):
        if arr[i] == target:
            return i# Return the index if the target is found

    return -1 # Return -1 if the target is not found
arr = [2, 5, 8, 12, 16, 23, 38, 42]
target = 16
result = linear_search(arr, target)

if result != -1:
    print(f"Element {target} is present at index {result}")
else:
    print(f"Element {target} is not present in the array")
```

58. /* Python Program for Insertion Sort*/

```
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
# Move elements of arr[0..i-1] that are greater than key to one position ahead of their current position
    while j >= 0 and key <arr[j]:
arr[j + 1] = arr[j]
    j -= 1

arr[j + 1] = key

arr = [12, 11, 13, 5, 6]
print("Original Array:", arr)
insertion_sort(arr)
print("Sorted Array:", arr)
```

59. /* Python program to get Current Date and Time*/

```
from datetime import datetime
# Get the current time
current_datetime = datetime.now()
# Format and print the current time
formatted_datetime = current_datetime.strftime("%Y-%m-%d %H:%M:%S")
print("Formatted Date Time:", formatted_datetime)
```

60. /* Python program to find difference between current time and given time*/

```
from datetime import datetime, timedelta
# Given time (replace with your own time)
given_time_str = "2024-01-27 12:30:00"
# Convert the given time string to a datetime object
given_time = datetime.strptime(given_time_str, "%Y-%m-%d %H:%M:%S")
# Get the current time
current_time = datetime.now()

# Calculate the difference between the current time and the given time
time_difference = current_time - given_time

print("Given Time:", given_time)
print("Current Time:", current_time)
print("Time Difference:", time_difference)
```

61./*How to convert timestamp string to datetime object in Python?*/

```
from datetime import datetime
# Example timestamp string
timestamp_str = "2024-01-27 15:30:00"
# Define the format of the timestamp string
timestamp_format = "%Y-%m-%d %H:%M:%S"
# Convert the timestamp string to a datetime object
datetime_object = datetime.strptime(timestamp_str, timestamp_format)

print("Timestamp String:", timestamp_str)
print("Datetime Object:", datetime_object)
```

62./*Find number of times every day occurs in a Year*/

```
import calendar
def count_days_in_year(year):
    # Initialize a dictionary to store the count of each day of the week
    day_counts = {
        'Monday': 0,
        'Tuesday': 0,
        'Wednesday': 0,
        'Thursday': 0,
        'Friday': 0,
        'Saturday': 0,
        'Sunday': 0
    }

    # Iterate through each month of the year
    for month in range(1, 13):
    # Get the matrix representing the month's calendar
    month_matrix = calendar.monthcalendar(year, month)
```

```
    # Iterate through each week of the month
    for week in month_matrix:
    # Iterate through each day of the week
    for day, day_number in enumerate(week):
# Check if the day is in the current month
        if day_number != 0:
            # Increment the count for the corresponding day of the week
day_name = calendar.day_name[day]
day_counts[day_name] += 1

    return day_counts

year = 2024
result = count_days_in_year(year)

# Print the result
for day, count in result.items():
    print(f"{day}: {count} occurrences")
```

63./*Python Program to Check if String Contain Only Defined Characters using Regex*/

```
import re
defonly_defined_chars(input,defined_chars):
# Define the regex pattern
    pattern = f'^[{re.escape(defined_chars)}]+$'
# Use re.match to check if the entire string matches the pattern
    match = re.match(pattern, input)
    return match is not None

defined_characters = 'abcde'
test_string1 = 'abc'
test_string2 = 'abcf123'

result1 = only_defined_chars(test_string1, defined_characters)
result2 = only_defined_chars(test_string2, defined_characters)

print(f"Test String 1: {test_string1} - Contains only defined characters: {result1}")
print(f"Test String 2: {test_string2} - Contains only defined characters: {result2}")
```

64./*Python program to Count Uppercase, Lowercase, special character and numeric values using Regex*/

```
import re
def count_characters(input_string):
    # Define regex patterns for different character types
uppercase_pattern = re.compile(r'[A-Z]')
lowercase_pattern = re.compile(r'[a-z]')
```

```
special_character_pattern = re.compile(r'[^A-Za-z0-9]')
numeric_pattern = re.compile(r'[0-9]')
    # Count occurrences using regex patterns
uppercase_count = len(re.findall(uppercase_pattern, input_string))
lowercase_count = len(re.findall(lowercase_pattern, input_string))
special_character_count = len(re.findall(special_characte_pattern, input_string))
numeric_count = len(re.findall(numeric_pattern, input_string))
    return uppercase_count, lowercase_count, special_character_count, numeric_count

test_string = "Hello123! How are you today?"

uppercase, lowercase, special_character, numeric = count_characters(test_string)

print("Uppercase Count:", uppercase)
print("Lowercase Count:", lowercase)
print("Special Character Count:", special_character)
print("Numeric Count:", numeric)
```

65./*Python Program to Check if email address valid or not*/

```
import re

def is_valid_email(email):
    # Define the email pattern using a regular expression
    email_pattern = re.compile(r'^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+$')
    # Use re.match to check if the entire email matches the pattern
    match = re.match(email_pattern, email)

    return match is not None

test_email1 = "user@example.com"
test_email2 = "invalid-email"

result1 = is_valid_email(test_email1)
result2 = is_valid_email(test_email2)

print(f"Email '{test_email1}' is valid: {result1}")
print(f"Email '{test_email2}' is valid: {result2}")
```

66./*Categorize Password as Strong or Weak using Regex in Python*/

```
import re
def categorize_pwd(password):
    # Define regex patterns for different character types
    uppercase_pattern = re.compile(r'[A-Z]')
    lowercase_pattern = re.compile(r'[a-z]')
    numeric_pattern = re.compile(r'[0-9]')
    special_character_pattern = re.compile(r'[!@#$%^&*()_+{}\[\]:;<>,.?~\\/-]')

    # Check if the password meets the criteria for a strong password
    has_uppercase = bool(re.search(uppercase_pattern, password))
    has_lowercase = bool(re.search(lowercase_pattern, password))
    has_numeric = bool(re.search(numeric_pattern, password))
    has_special_character = bool(re.search(special_character_pattern, password))

    # Check if the password is strong
    is_strong = has_uppercase and has_lowercase and has_numeric and has_special_character

    return "Strong" if is_strong else "Weak"

pwd1 = "StrongPassword123!"
pwd2 = "weakpassword"

result1 = categorize_pwd(pwd1)
result2 = categorize_pwd(pwd2)

print(f"Password '{test_password1}' is categorized as: {result1}")
print(f"Password '{test_password2}' is categorized as: {result2}")
```

67./*Python program to read file word by word*/

```
def read_file(file_path):
try:
    with open(file_path, 'r') as file:
# Iterate through each line in the file
    for line in file:
# Split the line into words
words = line.split()
# Iterate through each word in the line
        for word in words:
            print(word)
    except FileNotFoundError:
        print(f"File '{file_path}' not found.")
    except Exception as e:
        print(f"An error occurred: {e}")

file_path = 'example.txt'
```

```
# Replace with the path to your file
read_file(file_path)
```

68./*Python Program Get number of characters, words, spaces and lines in a file*/

```
def get_file_statistics(file_path):
    try:
        with open(file_path, 'r') as file:
# Read the entire content of the file
    content = file.read()
# Count the number of characters, words, spaces, and lines
num_characters = len(content)
num_words = len(content.split())
num_spaces = content.count(' ')
num_lines = content.count('\n') + 1
# Adding 1 to count the last line
        print(f"Number of characters: {num_characters}")
        print(f"Number of words: {num_words}")
        print(f"Number of spaces: {num_spaces}")
        print(f"Number of lines: {num_lines}")

    except FileNotFoundError:
        print(f"File '{file_path}' not found.")
    except Exception as e:
    print(f"An error occurred: {e}")

file_path = 'example.txt'
# Replace with the path to your file
get_file_statistics(file_path)
```

69./*Python Program to Eliminate repeated lines from a file*/

```
def rmv_dup_lines(input_file, output_file):
    try:
        with open(input_file, 'r') as input_file, open(output_file, 'w') as output_file:
unique_lines_set = set()
    # Iterate through each line in the input file
    for line in input_file:
    # Check if the line is not already in the set
    if line not in unique_lines_set:
# Write the line to the output file
output_file.write(line)
# Add the line to the set
unique_lines_set.add(line)
    except FileNotFoundError:
        print(f"File '{input_file}' not found.")
    except Exception as e:
        print(f"An error occurred: {e}")
input_file_path = 'input.txt'
# Replace with the path to your input file
output_file_path = 'output.txt'
```

```
# Replace with the desired output file path
rmv_dup_lines(input_file_path, output_file_path)
```

70./*Python Program to read List of Dictionaries from File*/

```
import json
def read_list_of_dicts_from_file(file_path):
try:
    with open(file_path, 'r') as file:
        # Load the JSON data from the file
        data = json.load(file)
        # Print the list of dictionaries
        print("List of Dictionaries:")
        for item in data:
            print(item)
    except FileNotFoundError:
        print(f"File '{file_path}' not found.")
    except json.JSONDecodeError as e:
    print(f"Error decoding JSON in '{file_path}': {e}")
    except Exception as e:
        print(f"An error occurred: {e}")
    file_path = 'data.json'
# Replace with the path to your JSON file
read_list_of_dicts_from_file(file_path)
```

Note :- save the file "data.json"

```
[
    {"name": "lakshay", "age": 14, "city": "nainital"},
    {"name": "palak", "age": 11, "city": "haldwani"},
    {"name": "meena", "age": 38, "city": "kashipur"}
]
```

71./*Python Program to merge two files into a third file*/

```
def merge_files(file1_path, file2_path, output_file_path):
    try:
        with open(file1_path, 'r') as file1, open(file2_path, 'r') as file2, open(output_file_path, 'w') as output_file:
    # Read content from the first file and write to the output file
output_file.write(file1.read())
# Add a newline character to separate the content of the two files
output_file.write('\n')
# Read content from the second file and write to the output file
output_file.write(file2.read())
    print(f"Merged files '{file1_path}' and '{file2_path}' into '{output_file_path}'")

    except FileNotFoundError:
        print(f"One or more files not found.")
```

```
except Exception as e:
        print(f"An error occurred: {e}")

file1_path = 'file1.txt'
# Replace with the path to your first input file
file2_path = 'file2.txt'
# Replace with the path to your second input file
output_file_path = 'merged.txt'
# Replace with the desired output file path
merge_files(file1_path, file2_path, output_file_path)
```

72. /*Create First GUI Application using Python-Tkinter*/

```
import tkinter as tk
def on_button_click():
label.config(text="Hello, Tkinter!")
# Create the main window
window = tk.Tk()
window.title("My First GUI App")
# Set geometry (widthxheight)
window.geometry('350x200')
# Create a label
label = tk.Label(window, text="Welcome to Tkinter!")
label.pack(pady=10)
# Create a button
button = tk.Button(window, text="Click Me", command=on_button_click)
button.pack(pady=10)
# Start the Tkinter event loop
window.mainloop()
```

73. /*Age Calculator using Tkinter*/

```
import tkinter as tk
from datetime import datetime
def calculate_age():
birthdate_str = entry_birthdate.get()
    try:
# Convert the input birthdate string to a datetime object
    birthdate = datetime.strptime(birthdate_str, "%Y-%m-%d")
# Get the current date
current_date = datetime.now()
# Calculate the age
    age = current_date.year - birthdate.year - ((current_date.month, current_date.day) < (birthdate.month, birthdate.day))
# Display the result
result_label.config(text=f"Your age is: {age} years")
    except ValueError:
result_label.config(text="Invalid date format. Please use YYYY-MM-DD.")
# Create the main window
window = tk.Tk()
window.title("Age Calculator")
# Set geometry (widthxheight)
window.geometry('350x200')
# Create and place widgets in the window
```

```
label_birthdate = tk.Label(window, text="Enter your birthdate (YYYY-MM-DD):")
label_birthdate.pack(pady=10)
entry_birthdate = tk.Entry(window)
entry_birthdate.pack(pady=10)
button_calculate = tk.Button(window,
 text="Calculate Age", command=calculate_age)
 button_calculate.pack(pady=10)
result_label = tk.Label(window, text="")
result_label.pack(pady=10)
# Start the Tkinter event loop
window.mainloop()
```

74. /*Create a digital clock using Tkinter*/

```
import tkinter as tk
from time import strftime
def update_time():
    current_time = strftime('%H:%M:%S %p')
    label.config(text=current_time) label.after(1000, update_time) # Update every 1000 milliseconds (1 second)
# Create the main window
window = tk.Tk()
window.title("Digital Clock")
# Create a label to display the time
label = tk.Label(window, font=('calibri', 40, 'bold'), background='black',
 foreground='white')
label.pack(anchor='center')

# Call the update_time function to initialize the label
update_time()
# Start the Tkinter event loop
window.mainloop()
```

75./*Create Simple registration form using python Tkinter*/

```
import tkinter as tk
from tkinter import messagebox
def register_user():
# Retrieve values from the form
    username = entry_username.get()
    password = entry_password.get()
    gender = var_gender.get()
    hobbies = [var_hobby1.get(), var_hobby2.get(), var_hobby3.get()]
    country = listbox_country.get(listbox_country.curselection())
    address = text_address.get("1.0", tk.END)
# Display the registered user information
    message = f"Registered User:\nUsername: {username}\nPassword: {password}\nGender: {gender}\nHobbies: {hobbies}\nCountry: {country}\nAddress: {address}"
```

```
messagebox.showinfo("Registration Successful", message)
# Optionally, you can clear the form after registration
clear_form()
def clear_form():
    entry_username.delete(0, tk.END)

    entry_password.delete(0, tk.END)
    var_gender.set("")
# Clear radio button selection
    var_hobby1.set(0)
# Clear checkbox selection
    var_hobby2.set(0)
    var_hobby3.set(0)
listbox_country.selection_clear(0, tk.END)

# Clear listbox selection
text_address.delete("1.0", tk.END)

# Create the main window
window = tk.Tk()
window.title("Registration Form")

# Create and place widgets in the window
label_username = tk.Label(window,
 text="Username:")

label_username.grid(row=0, column=0, padx=10, pady=5, sticky="e")

entry_username = tk.Entry(window)

entry_username.grid(row=0, column=1, padx=10, pady=5)

label_password = tk.Label(window,
text="Password:")

label_password.grid(row=1, column=0, padx=10, pady=5, sticky="e")

entry_password = tk.Entry(window, show="*")

entry_password.grid(row=1, column=1, padx=10, pady=5)

label_gender = tk.Label(window,
text="Gender:")

label_gender.grid(row=2, column=0, padx=10, pady=5, sticky="e")

var_gender = tk.StringVar(value="Male")

radio_male = tk.Radiobutton(window,
text="Male",
variable=var_gender,
value="Male")
```

```
radio_male.grid(row=2,
column=1, padx=10, pady=5,
sticky="w")

radio_female = tk.Radiobutton(window,
text="Female",
variable=var_gender,
value="Female")

radio_female.grid(row=2,
column=2, padx=10, pady=5,
sticky="w")

label_hobbies = tk.Label(window,
text="Hobbies:")

label_hobbies.grid(row=3,
column=0, padx=10, pady=5,
sticky="e")

var_hobby1 = tk.StringVar(value="Reading")

check_hobby1 = tk.Checkbutton(window,
text="Reading",
variable=var_hobby1)

check_hobby1.grid(row=3,
column=1, padx=10, pady=5,
sticky="w")

var_hobby2 = tk.StringVar(value="Sports")

check_hobby2 = tk.Checkbutton(window,
text="Sports",
variable=var_hobby2)

check_hobby2.grid(row=3,
column=2, padx=10, pady=5,
sticky="w")

var_hobby3 = tk.StringVar(value="Music")

check_hobby3 = tk.Checkbutton(window,
text="Music",
variable=var_hobby3)

check_hobby3.grid(row=3,
column=3, padx=10, pady=5,
sticky="w")

label_country = tk.Label(window,
text="Country:")

label_country.grid(row=4,
column=0, padx=10, pady=5,
sticky="e")

countries = ["USA", "Canada",
"UK", "India", "Australia"]

listbox_country = tk.Listbox(window,
selectmode=tk.SINGLE,
height=len(countries))

for country in countries:

listbox_country.insert(tk.END,
country)
```

```
listbox_country.grid(row=4,
column=1, padx=10, pady=5)

label_address =
tk.Label(window,
 text="Address:")

label_address.grid(row=5,
column=0, padx=10, pady=5,
sticky="e")

text_address = tk.Text(window,
height=4, width=30)

text_address.grid(row=5,
column=1, columnspan=3,
padx=10, pady=5)

button_register =
tk.Button(window,
text="Register",
command=register_user)

button_register.grid(row=6,
columnspan=4, pady=10)

# Start the Tkinter event loop

 window.mainloop()
```

76./*Create a Voice Recorder using Python*/

Creating a voice recorder using Python involves using the **sounddevicelibrary** for capturing **audio** and the **waviolibrary** for saving the recorded audio as a WAV file

```
#pip install sounddevice
#pip install wavio
#at first install above liberary
import sounddevice as sd
import wavio
def record_voice(duration,
samplerate=44100,
filename="output.wav"):
    print("Recording...")
# Record audio
    recording =
sd.rec(int(samplerate *
duration),

samplerate=samplerate,
channels=2, dtype='int16')
sd.wait()
    print("Recording complete.")
    # Save as WAV file
wavio.write(filename,
recording, samplerate,
sampwidth=3)
    print(f"Audio saved as
{filename}")
if __name__ == "__main__":
    # Set the duration of the
recording in seconds
recording_duration = 5
# Specify the filename for the
output WAV file
output_filename =
"output.wav"
record_voice(recording_duratio
n, filename=output_filename)
```

77./*Create a Screen recorder using Python*/

Creating a screen recorder in Python can be achieved using the **pyautoguilibrary** for capturing screenshots and the **imageio library** for creating a video from the captured frames

```
#pip install pyautogui
#pip install imageio
#at first install above liberary
import pyautogui
import imageio
import os
import time

def
record_screen(output_filename
="output.mp4", duration=10,
fps=30):
    print("Recording...")
    # Specify the screen
resolution
screen_size = pyautogui.size()
    # Set up the output file
output_path =
os.path.join(os.getcwd(),
output_filename)
    writer =
imageio.get_writer(output_path
, fps=fps)
start_time = time.time()
    try:
        while time.time() -
start_time< duration:
# Capture screenshot
        screenshot =
pyautogui.screenshot()
    # Convert the screenshot
to a NumPy array
    frame =
imageio.core.util.Array(screens
hot)
    # Append the frame to the
video
writer.append_data(frame)
    except KeyboardInterrupt:
        pass
    finally:
writer.close()
    print(f"Recording complete.
Video saved as
{output_filename}")
if __name__ == "__main__":
# Set the duration of the
recording in seconds
recording_duration = 10

    # Specify the filename for
the output video file
output_filename =
"output.mp4"
record_screen(output_filename
, duration=recording_duration)
```

78./*Draw a Tic Tac Toe Board using Python-Turtle*/

```
import turtle
def draw_board():
turtle.speed(2) # Set turtle
speed (1=slow, 10=fastest)
    # Draw horizontal lines
turtle.penup()
turtle.goto(-150, 50)
turtle.pendown()
turtle.forward(300)
```

```
turtle.penup()
turtle.goto(-150, -50)
turtle.pendown()
turtle.forward(300)
    # Draw vertical lines
turtle.penup()
turtle.goto(-50, 150)
turtle.right(90)
turtle.pendown()
turtle.forward(300)
turtle.penup()
turtle.goto(50, 150)
turtle.pendown()
turtle.forward(300)
turtle.hideturtle()
# Hide turtle after drawing
if __name__ == "__main__":
draw_board()
turtle.done()
```

79./*Create pong game using Python – Turtle*/

```
import turtle
# Set up the screen
screen = turtle.Screen()
screen.title("Pong Game")
screen.bgcolor("black")
screen.setup(width=600,
 height=400)
# Paddle A
paddle_a = turtle.Turtle()
paddle_a.speed(0)
paddle_a.shape("square")
paddle_a.color("white")
paddle_a.shapesize(stretch_wi
d=1, stretch_len=5)
paddle_a.penup()
paddle_a.goto(-250, 0)
# Paddle B
paddle_b = turtle.Turtle()
paddle_b.speed(0)
paddle_b.shape("square")
paddle_b.color("white")
paddle_b.shapesize(stretch_wi
d=1, stretch_len=5)
paddle_b.penup()
paddle_b.goto(240, 0)
# Ball
ball = turtle.Turtle()
ball.speed(40)
ball.shape("square")
 ball.color("white")
ball.penup()
ball.goto(0, 0)
ball.dx = 2 # Ball movement
speed in the x-axis
ball.dy = -2 # Ball movement
speed in the y-axis
# Paddle movement functions
def paddle_a_up():
    y = paddle_a.ycor()
    if y < 190:
    y += 20
paddle_a.sety(y)
def paddle_a_down():
    y = paddle_a.ycor()
    if y > -190:
        y -= 20
paddle_a.sety(y)
def paddle_b_up():
    y = paddle_b.ycor()
    if y < 190:
        y += 20
```

```
paddle_b.sety(y)
def paddle_b_down():
    y = paddle_b.ycor()
    if y > -190:
        y -= 20
paddle_b.sety(y)
# Keyboard bindings
screen.listen()
screen.onkey(paddle_a_up, "w")
screen.onkey(paddle_a_down, "s")
screen.onkey(paddle_b_up, "Up")
screen.onkey(paddle_b_down, "Down")
# Main game loop
while True:
screen.update()
    # Move the ball
ball.setx(ball.xcor() + ball.dx)
ball.sety(ball.ycor() + ball.dy)
    # Border checking
    if ball.ycor() > 190 or ball.ycor() < -190:
ball.dy *= -1 # Reverse the direction when hitting the top or bottom border
    # Paddle collisions
    if (ball.xcor() > 235 and ball.xcor() < 240) and (ball.ycor() <paddle_b.ycor() + 50 and ball.ycor() >paddle_b.ycor() - 50):
ball.color("blue")
ball.setx(235)
ball.dx *= -1
elif (ball.xcor() < -240 and ball.xcor() > -245) and (ball.ycor() <paddle_a.ycor() + 50 and ball.ycor() >paddle_a.ycor() - 50):
ball.color("red")
ball.setx(-240)
ball.dx *= -1
```

80./*Python program to read CSV file using Pandas*/

```
import pandas as pd
# Specify the path to your CSV file
csv_file_path = 'your_file.csv'
# Read the CSV file into a DataFrame
df = pd.read_csv(csv_file_path)
# Display the DataFrame
print(df)
```

81./*Python program to create data frame using Pandas*/

```
import pandas as pd
# Sample data
data = {'Name': [lakshay, 'palak', 'priyansh'],
    'Age': [14, 11, 9],
    'City': ['nainital', 'ramnagar', 'haldwani']}
# Create a DataFrame
df = pd.DataFrame(data)
# Display the DataFrame
print(df)
```

82./*Python program to remove column using Pandas*/ i

```
mport pandas as pd
```

```
# Create a DataFrame
data = {'Name': [lakshay, 'palak', 'priyansh'],
    'Age': [14, 11, 9],
    'City': ['nainital', 'ramnagar', 'haldwani']}
df = pd.DataFrame(data)
# Display the original DataFrame
print("Original DataFrame:")
print(df)
# Remove the 'Age' column
df = df.drop('Age', axis=1)
# Display the DataFrame after removing the 'Age' column
print("\nDataFrame after removing 'Age' column:")
print(df)
```

83./*Python program to search column using Pandas*/

```
import pandas as pd
# Create a DataFrame
data = {'Name': [lakshay, 'palak', 'priyansh'],
    'Age': [14, 11, 9],
    'City': ['nainital', 'ramnagar', 'haldwani']}
df = pd.DataFrame(data)

# Display the original DataFrame
print("Original DataFrame:")
print(df)
# Search for rows where the 'City' column is 'San Francisco'
search_result = df[df['City'] == 'San Francisco']
# Display the search result
print("\nSearch result for 'City' column containing 'San Francisco':")
print(search_result)
```

84./*Python program to use matplotliberary*/

```
import pandas as pd
import matplotlib.pyplot as plt
# Create a DataFrame
data = {'Name': [lakshay, 'palak', 'priyansh'],
    'Age': [14, 11, 9],
    'City': ['nainital', 'ramnagar', 'haldwani']}
df = pd.DataFrame(data)

# Display the DataFrame
print("Original DataFrame:")
print(df)

# Plot the 'Age' column
plt.plot(df['Age'], marker='o', linestyle='-')
plt.title('Age Distribution')

plt.xlabel('Index')
plt.ylabel('Age')
plt.grid(True)
plt.show()
```

85./*Python program to create scatterplot using matplotliberary*/

```
import matplotlib.pyplot as plt
# Sample data
x = [1, 2, 3, 4, 5]
y = [2, 3, 5, 7, 11]
# Create scatter plot
plt.scatter(x, y)
#Title and labels
plt.title('Scatter Plot Example')
plt.xlabel('X values')
plt.ylabel('Y values')
# Show the plot
plt.show()
```

86./*Python program to create bar plot using matplotliberary*/

```
import matplotlib.pyplot as plt
# Sample data
labels = ['A', 'B', 'C', 'D', 'E']
values = [3, 7, 2, 5, 8]
# Create bar plot
plt.bar(labels, values)
# Title and labels
plt.title('Bar Plot Example')
plt.xlabel('Labels')
plt.ylabel('Values')
# Show the plot
plt.show()
```

87./* Python program to show stack implementation*/

```
class Stack:
    def __init__(self):
        self.items = []
    def is_empty(self):
        return len(self.items)== 0
    def push(self, item):
        self.items.append(item)
    def pop(self):
        if not self.is_empty():
            return self.items.pop()
        else:
            raise IndexError("pop
            from an empty stack")
    def peek(self):
        if not self.is_empty():
            return self.items[-1]
        else:
            raise IndexError
            ("peek from an empty
            stack")
    def size(self):
        return len(self.items)
# Example usage:
if __name__ == "__main__":
    stack = Stack()
    stack.push(1)
    stack.push(2)
    stack.push(3)
    print("Current stack:",
    stack.items)
    print("Stack size:",
    stack.size())
    print("Peek:", stack.peek())
    print("Pop:", stack.pop())
    print("Current stack:",
    stack.items)
```

88./* Python program to show queue implementation*/

```
class Queue:
    def __init__(self):
        self.items = []
```

```
def is_empty(self):
        return len(self.items)==0
    def enqueue(self, item):
        self.items.append(item)
    def dequeue(self):
        if not self.is_empty():
            return
            self.items.pop(0)
        else:
        raise IndexError
        ("dequeue from an empty
        queue")
    def peek(self):
        if not self.is_empty():
            return self.items[0]
        else:
        raise IndexError("peek
        from an empty queue")
    def size(self):
        return len(self.items)
```

 # Example usage:

```
if __name__ == "__main__":
    queue = Queue()
    queue.enqueue(1)
    queue.enqueue(2)
    queue.enqueue(3)
    print("Current queue:",
    queue.items)
    print("Queue size:",
    queue.size())
    print("Peek:", queue.peek())
    print("Dequeue:",
    queue.dequeue())
    print("Current queue:",
    queue.items)
```

89./* Python program to show linked list implementation*/

```
class Node:
    def __init__(self, data):
    self.data = data
    self.next = None
class LinkedList:
    def __init__(self):
    self.head = None

    def append(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
        return
            last_node = self.head
        while last_node.next:
            last_node =
            last_node.next
            last_node.next =
            new_node
    def prepend(self, data):
        new_node = Node(data)
        new_node.next =
        self.head
        self.head = new_node
    def delete_node(self, key):
        current_node = self.head
        if current_node and
            current_node.data==
            key:
            self.head =
            current_node.next
            current_node = None
            return prev = None
        while current_node and
        current_node.data !=
        key:
            prev = current_node
            current_node =
            current_node.next
            if current_node is
```

```
            None:
                return prev.next =
                current_node.next
                current_node =
                None
    def print_list(self):
        current_node = self.head
        while current_node:
    print(current_node.data)
    current_node =
    current_node.next
# Example usage:
if __name__ == "__main__":
    ll = LinkedList()
    ll.append(1)
    ll.append(2)
    ll.append(3)
    ll.append(4)
    ll.prepend(0)
    ll.print_list()
    ll.delete_node(3)
    print("After deleting 3:")
    ll.print_list()
```

90./* Python program to show calculates the mean, median, mode, variance, and standard deviation*/

```
import numpy as np
from scipy import stats
def
calculate_statistics(numbers):
    mean=np.mean(numbers)
    median=np.median(number
    s)
    mode=stats.mode(numbers
    )[0][0]
    variance = np.var(numbers)
    std_dev = np.std(numbers)
    return mean, median, mode,
variance, std_dev
if __name__ == "__main__":
numbers = [2, 4, 4, 4, 5, 5, 7, 9]
mean, median, mode, variance,
std_dev =
calculate_statistics(numbers)
    print("Mean:", mean)
    print("Median:", median)
    print("Mode:", mode)
    print("Variance:", variance)
    print("Standard  Deviation:",
std_dev)
```

91./* Python program to include API */

```
import requests
def fetch_data_from_api(url):
    try:
        response =
        requests.get(url)
        response.raise_
        for_status()
# Raise an exception for HTTP
errors (4xx or 5xx)
        data = response.json()
#Convert the response to
JSON #format
        return data
    except requests.exceptions.
        RequestException as e:
            print("Error fetching
            data:", e)
            return None
# Example usage:
if __name__ == "__main__":
    url ="https://jsonplaceholder
    .typicode.com/posts/1"
```

```
    api_data =
    fetch_data_from_api(url)
        if api_data:
            print("API
            Response:", api_data)
```

92./* Python program to include GOOGLE MAP API */

```
import requests
def
    geocode_adrs (address):
    api_key = 'YOUR_API_KEY'
# Replace 'YOUR_API_KEY'
with your actual Google Maps
API key
    url = f'https://maps.google
    apis.com/maps/api/
    geocode/json?address
    ={address} &key={api_key}'
    try:
        response =
        requests.get(url)
        response.raise_for
        _ status()
        data = response.json()
        if data['status'] == 'OK':
            location =
            data['results'][0]
            ['geometry']['location']
            latitude = location['lat']
            longitude=
            location['lng']
            return latitude,
            longitude
        else:
            print("Geocoding
            failed:", data['status'])
            return None, None
    except
    requests.exceptions.
    RequestException as e:
        print("Error geocoding
        address:", e)
        return None, None
# Example usage:
if __name__ == "__main__":
    address = "1600
Amphitheatre Parkway,
Mountain View, CA"
    latitude, longitude =
    geocode_adrs(address)
    if latitude is not None and
        longitude is not None:
        print("Latitude:",
        latitude)
        print("Longitude:",
        longitude)
```