Mastering Python Programming

From Beginner to Advanced with AI,ML & Data Science

-Suryanshsk

About the Author

Avanish Singh is a passionate software developer, full-stack web developer, and AI enthusiast with a deep expertise in Python, Java, Flutter, AI, and ML. With a career rooted in solving complex problems through innovative coding solutions, Avanish has contributed to numerous projects across various domains, from web development to artificial intelligence.

A dedicated programmer and lifelong learner, Avanish has mastered the art of making complex concepts accessible and understandable. He is an experienced educator and mentor, guiding aspiring developers through the intricacies of coding and helping them achieve their goals.

In addition to his professional pursuits, Avanish is an active participant in hackathons and coding competitions, where he has honed his skills and contributed to groundbreaking projects. His approach to Python programming is both practical and insightful, making him a trusted voice in the tech community.

When not coding, Avanish enjoys exploring new technologies, contributing to open-source projects, and sharing his knowledge through workshops and online platforms.

Table of Contents

Chapter 1: Introduction to Python Programming

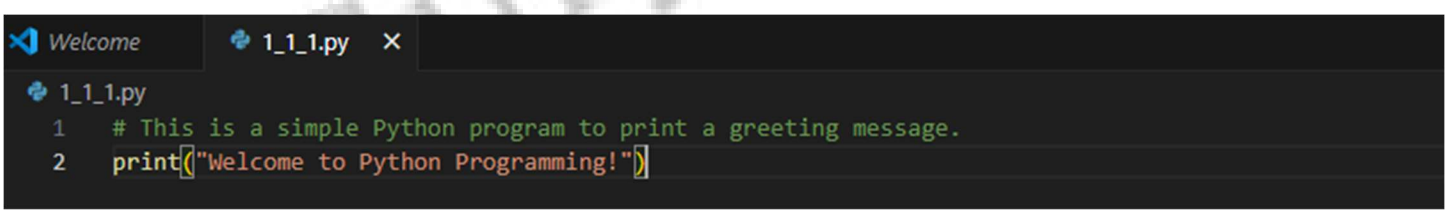
1.1 What is Python?

Python is a high-level, interpreted programming language that was created by Guido van Rossum and first released in 1991. It is designed to be easy to read and write, with a syntax that emphasizes readability and simplicity. Python supports multiple programming paradigms, including procedural, object-oriented, and functional programming.

Key Features of Python:

- **Interpreted Language:** Python code is executed line by line, which makes it easier to test and debug.
- **High-Level Language:** Python abstracts away many of the complexities of computer operations, allowing you to focus on solving problems rather than managing memory or dealing with low-level system details.
- **Dynamically Typed:** In Python, you don't need to declare the type of a variable when you create one; Python automatically assigns the type based on the value you provide.
- **Extensive Standard Library:** Python comes with a large standard library that includes modules and packages for various tasks, such as handling files, working with data, and performing network operations

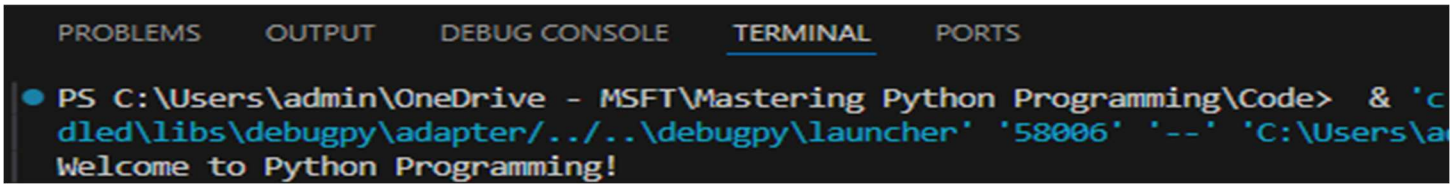
Example Code:



Explanation:

- The `print()` function displays the text `"Welcome to Python Programming!"` on the screen.

Output:



1.2 Why Learn Python?

Python is one of the most popular programming languages in the world, and there are several reasons why learning Python is beneficial:

1.2.1 Ease of Learning

Python's syntax is clear, intuitive, and close to human language. This makes it easier to learn, especially for beginners who are new to programming.

1.2.2 Versatility

Python can be used for various applications, such as:

- **Web Development:** Frameworks like Django and Flask make it easy to build web applications.
- **Data Science and Machine Learning:** Libraries like NumPy, Pandas, and TensorFlow are extensively used in these fields.
- **Automation:** Python can automate repetitive tasks, making it a valuable tool for productivity.
- **Software Development:** Python is used in developing desktop applications, games, and complex algorithms.

1.2.3 Large Community and Resources

Python has a vast community of developers, which means there are many resources available for learning and problem-solving. Whether it's tutorials, forums, or documentation, you'll find plenty of help when learning Python.

1.2.4 Career Opportunities

Python is widely used in industries, including technology, finance, healthcare, and more. Proficiency in Python opens doors to various career opportunities, such as software development, data science, and AI/ML engineering.

1.3 Installing Python

To start coding in Python, you first need to install Python on your system. Here's a step-by-step guide:

1.3.1 Downloading Python

1. **Visit the Python Official Website:** Go to python.org and navigate to the Downloads section.
2. **Choose Your Operating System:** Select your operating system (Windows, macOS, or Linux) and download the latest version of Python.

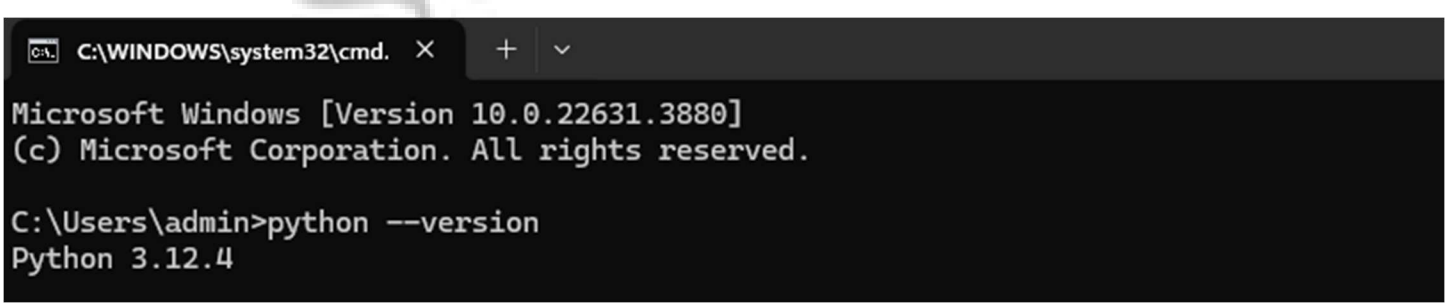
1.3.2 Installing Python

1. **Run the Installer:** Open the downloaded installer.
2. **Add Python to PATH:** Check the box that says "Add Python to PATH" before proceeding with the installation. This ensures that you can run Python from the command line.
3. **Choose Installation Options:** You can proceed with the default installation options or customize them as per your needs.
4. **Complete the Installation:** Click "Install Now" and wait for the installation to complete.

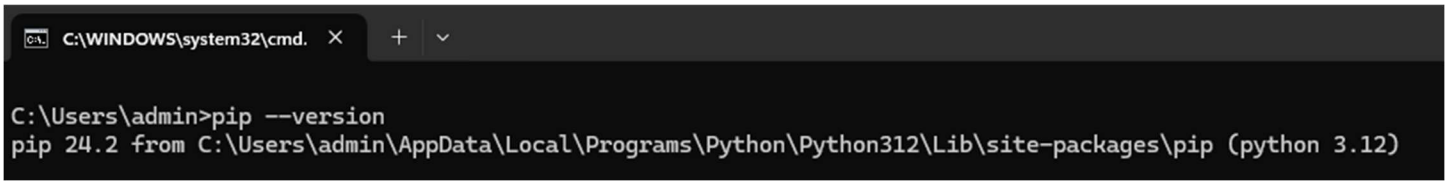
1.3.3 Verifying the Installation

After installation, verify that Python is installed correctly:

1. **Open a Terminal or Command Prompt:** Depending on your OS, open a terminal (macOS/Linux) or command prompt (Windows)
2. **Check Python Version:** Type the following command and press Enter:



3. **Check PIP Installation:** PIP (Python Package Installer) should also be installed with Python. Verify it using:



If both commands return a version number, your installation is successful.

1.4 Setting Up Your Development Environment

1.4.1 Choosing a Text Editor or IDE

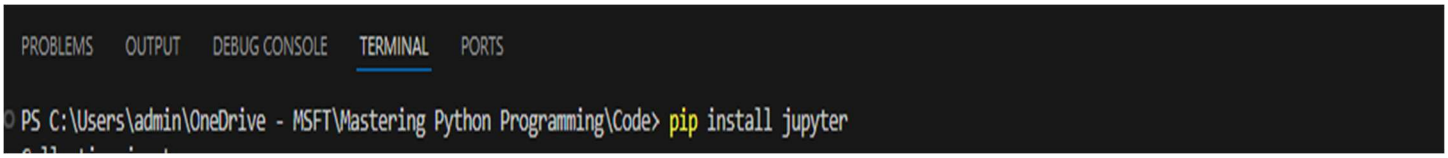
A text editor or Integrated Development Environment (IDE) is where you'll write and run your Python code. Here are some popular options:

- **Text Editors:**
  - **Sublime Text:** A lightweight, versatile text editor that supports multiple programming languages.
  - **Notepad++:** A simple yet powerful text editor with support for various file types and plugins.
- **IDEs:**
  - **PyCharm:** A powerful IDE specifically for Python, with features like intelligent code completion, debugging, and more.
  - **Visual Studio Code:** A popular open-source code editor with excellent support for Python through extensions.

1.4.2 Installing Additional Tools

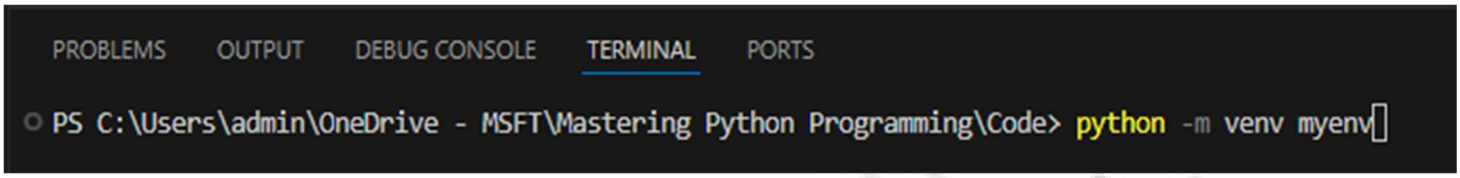
You may also want to install the following tools to enhance your Python development experience:

- **Jupyter Notebook:** A web-based application that allows you to create and share documents containing live code, equations, visualizations, and explanatory text. It's especially useful for data science and AI projects.
  - Install via pip:



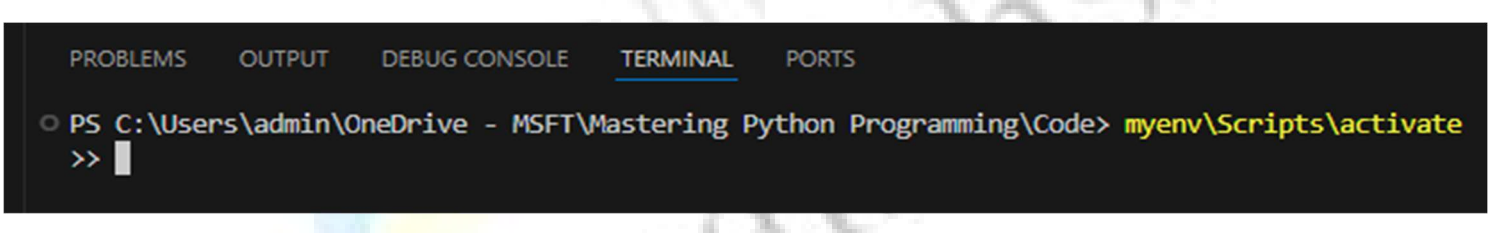
Virtual Environment: It's a good practice to use virtual environments to manage dependencies for different projects. This keeps your projects isolated and ensures that you have the right packages installed for each project.

- Create a virtual environment



Activate the virtual environment:

- On Windows:

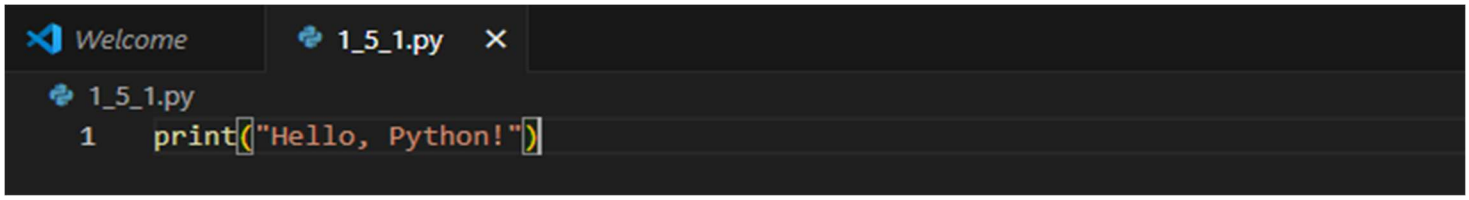


1.5 Running Your First Python Program

Now that Python is installed, let's write and run your first Python program.

1.5.1 Using the Command Line

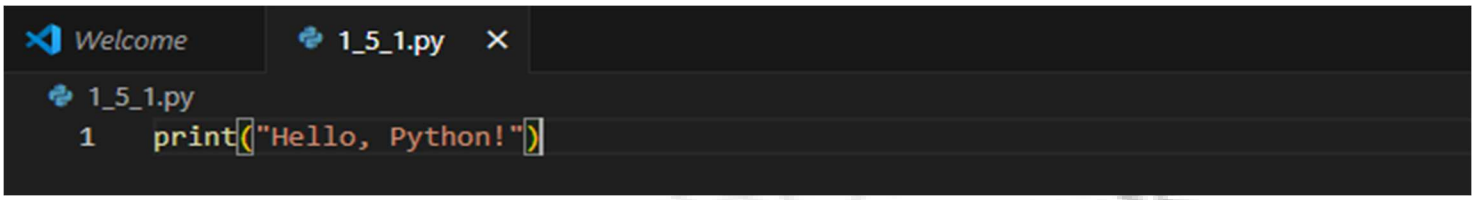
1. **Open Your Terminal or Command Prompt:** On Windows, you can search for "cmd" in the start menu. On macOS/Linux, open a terminal window.
2. **Start the Python Interpreter:** Type `python` and press Enter.
3. **Write Your Program:** At the >>> prompt, type the following code and press Enter:



4. **Exit the Interpreter:** Type `exit()` or press `Ctrl + Z` (on Windows) or `Ctrl + D` (on macOS/Linux) to exit.

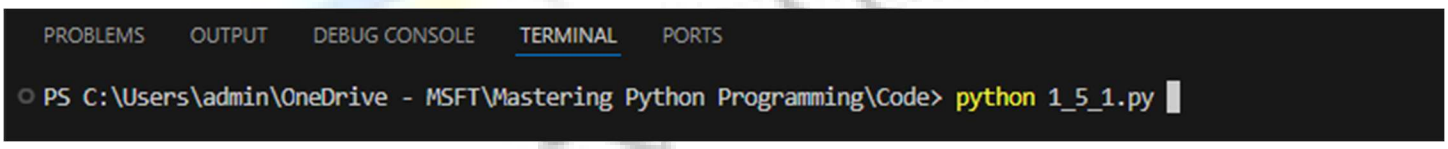
1.5.2 Running a Python Script

1. **Create a Python Script:**
   - Open your text editor or IDE and create a new file.
   - Save the file with a `.py` extension, e.g., `hello.py`.
   - In the file, write the following code:

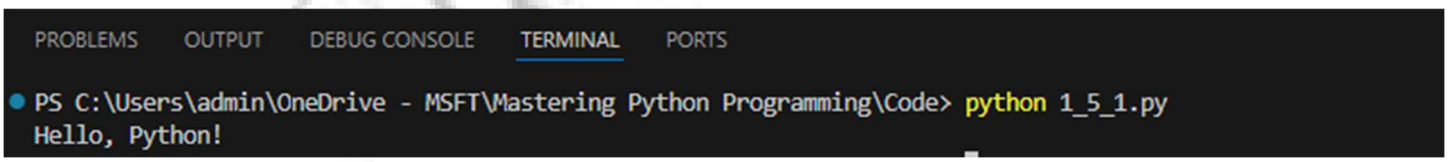


Run the Script from the Command Line:

- Navigate to the directory where your script is saved using the `cd` command.
- Run the script by typing:



Output:



Chapter 2: Python Basics

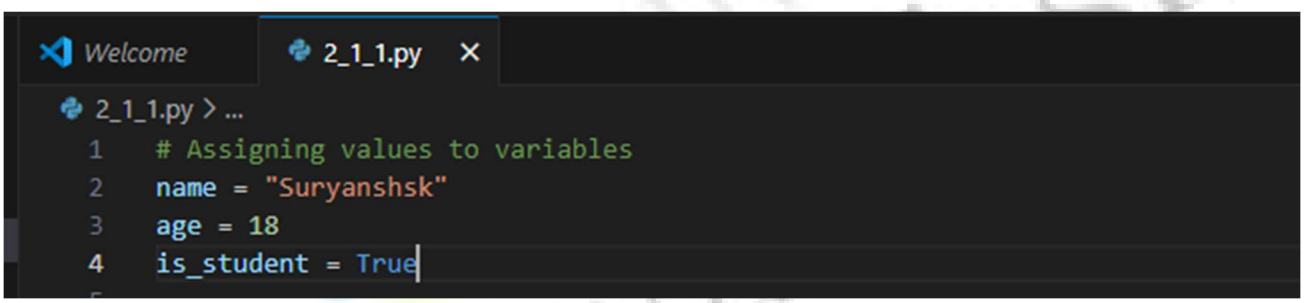
2.1 Variables and Data Types

In Python, variables are used to store data. A variable is essentially a name that refers to a value. Python supports various data types to store different kinds of data.

2.1.1 Variables

A variable in Python is created when you assign a value to it. You don't need to declare the variable type explicitly.

Example Code:



Explanation:

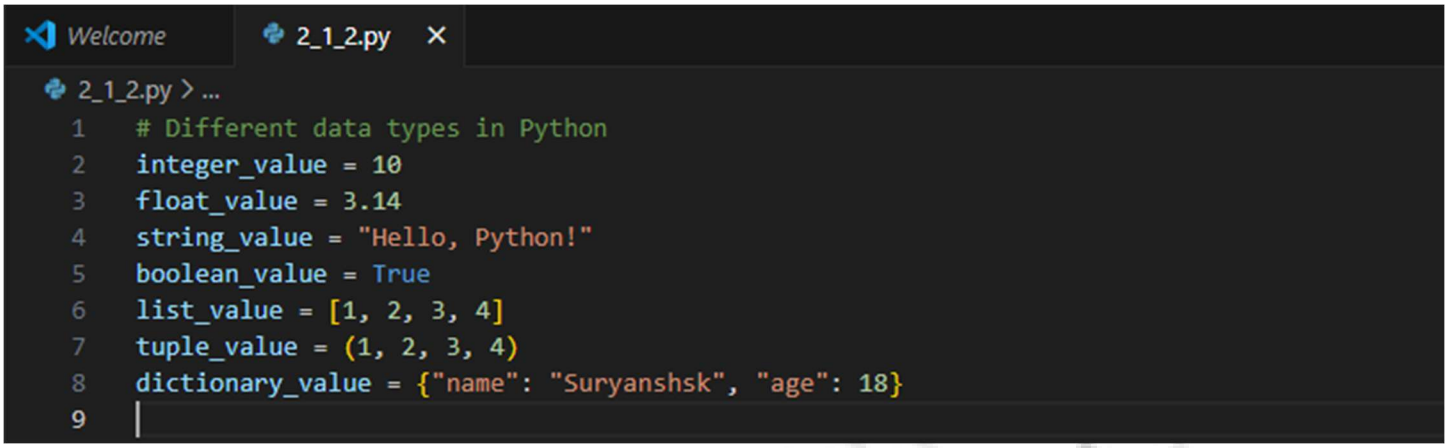
- `name` is a string variable that stores text.
- `age` is an integer variable that stores whole numbers.
- `is_student` is a boolean variable that stores `True` or `False`.

2.1.2 Data Types

Python has several built-in data types:

1. **Integers:** Whole numbers, e.g., `10`, `-5`, `0`.
2. **Floats:** Decimal numbers, e.g., `3.14`, `-2.5`.
3. **Strings:** Sequence of characters, e.g., `"Hello, World!"`.
4. **Booleans:** Represents `True` or `False`.
5. **Lists:** Ordered collection of items, e.g., `[1, 2, 3, 4]`.
6. **Tuples:** Ordered collection of items (immutable), e.g., `(1, 2, 3, 4)`.
7. **Dictionaries:** Collection of key-value pairs, e.g., `{"name": "Suryanshsk", "age": 18}`.

Example Code:



Explanation:

- Each variable is assigned a value of a specific data type.

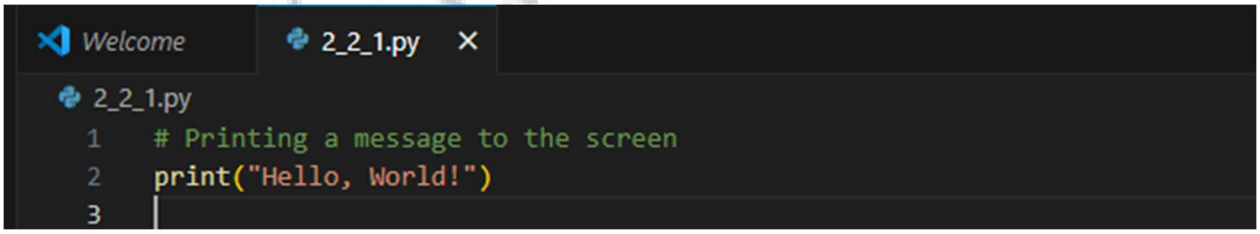
2.2 Basic Input and Output

Python provides simple functions to interact with users by taking input and displaying output.

2.2.1 Output using `print()` Function

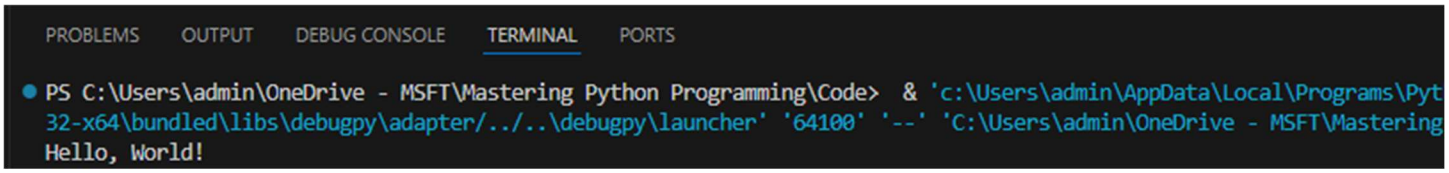
The `print()` function is used to display output on the screen.

Example Code:



Explanation:

- The `print()` function displays the text `"Hello, World!"` on the screen.



2.2.2 Input using `input()` Function

The `input()` function is used to take input from the user.

Example Code:

Explanation:

- The `input()` function prompts the user to enter a value. The entered value is stored in the `name` variable.
- The `print()` function then displays a greeting message using the entered name.

Output:

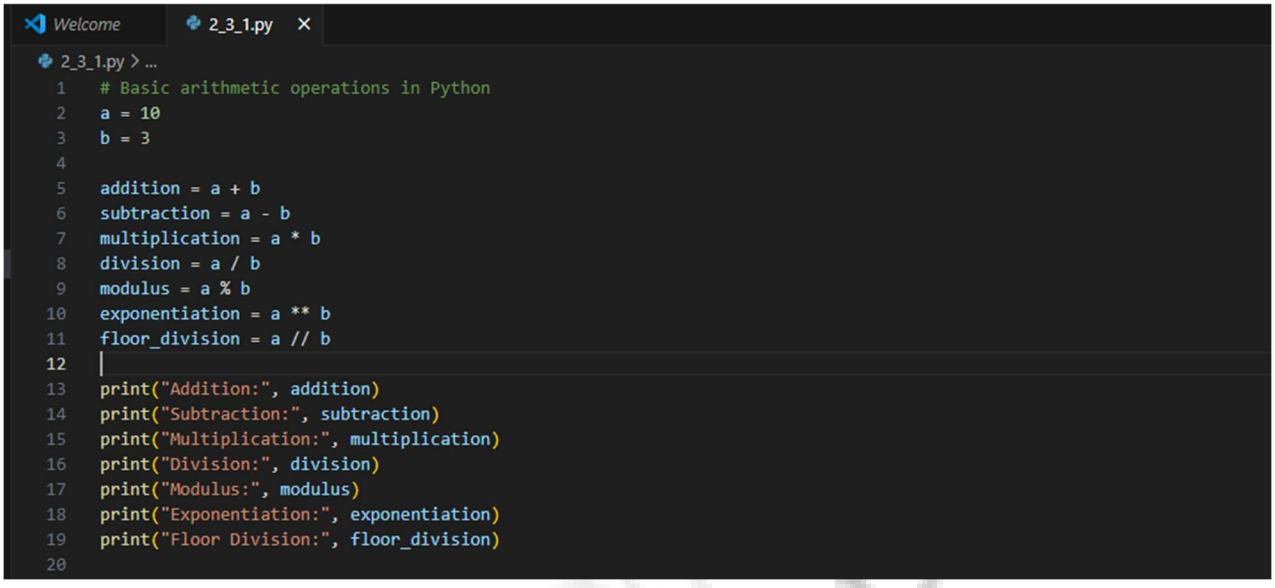
2.3 Arithmetic Operations

Python can perform various arithmetic operations using basic operators like `+`, `-`, `*`, `/`, etc.

2.3.1 Basic Arithmetic Operators

1. **Addition (+)**: Adds two numbers.
2. **Subtraction (-)**: Subtracts the second number from the first.
3. **Multiplication (*)**: Multiplies two numbers.
4. **Division (/)**: Divides the first number by the second.
5. **Modulus (%)**: Returns the remainder of the division.
6. **Exponentiation (**)**: Raises the first number to the power of the second.
7. **Floor Division (//)**: Divides and returns the largest integer less than or equal to the result.

Example Code:



Explanation:

- Each arithmetic operation is performed using the corresponding operator, and the results are stored in variables.

Output:

```
Addition: 13
Subtraction: 7
Multiplication: 30
Division: 3.3333333333333335
Modulus: 1
Exponentiation: 1000
Floor Division: 3
```

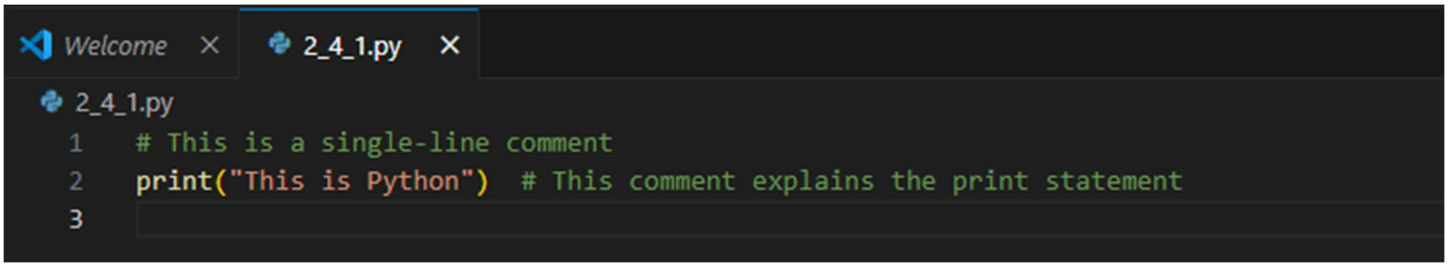
2.4 Comments in Python

Comments are used to explain code and make it more readable. Python supports single-line and multi-line comments.

2.4.1 Single-Line Comments

Single-line comments start with a # symbol.

Example Code:



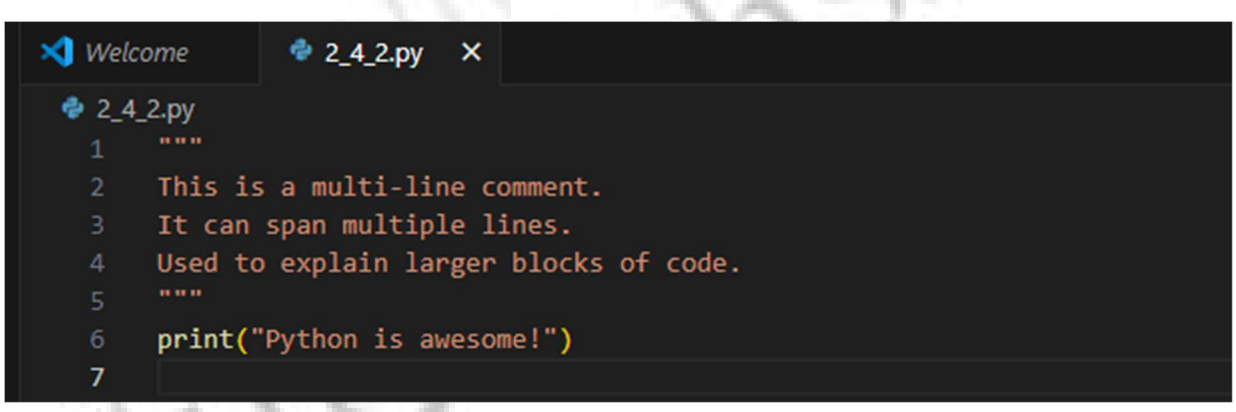
Explanation:

- Comments are ignored by the Python interpreter and are only meant for the developer's reference.

2.4.2 Multi-Line Comments

Multi-line comments can be created using triple quotes (`'''` or `"""`).

Example Code:



Explanation:

- Multi-line comments are typically used to describe sections of code or provide detailed explanations.

Output

2.5 Python Syntax and Indentation

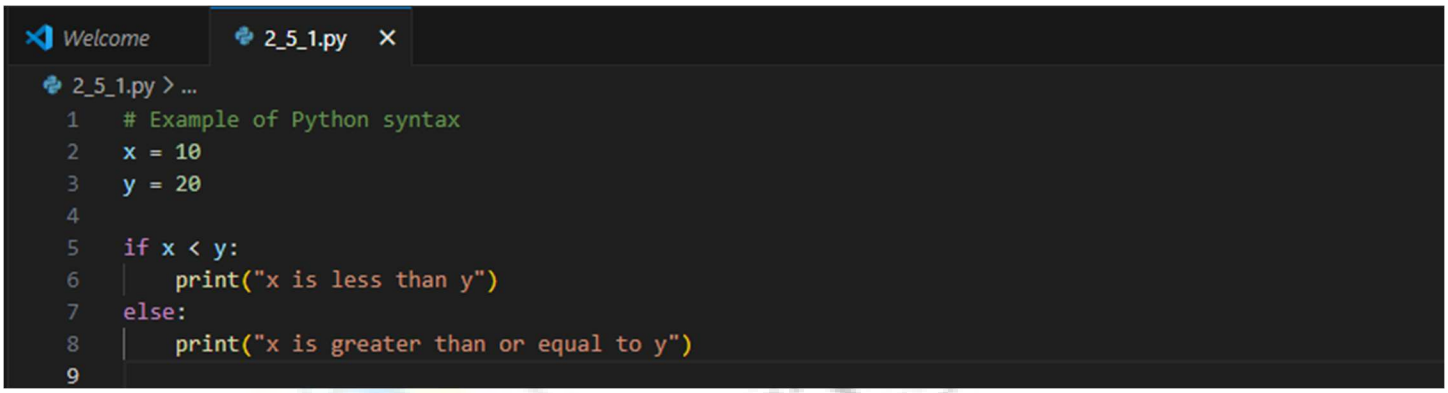
Python syntax refers to the rules that define how a Python program is written. One of Python's most unique features is its use of indentation to define blocks of code.

2.5.1 Python Syntax

Python syntax is designed to be readable and concise. Here are some basic rules:

1. **Case Sensitivity:** Python is case-sensitive, meaning `Variable` and `variable` are considered different.
2. **Statements:** Python statements typically end with a newline. You can use a semicolon (`;`) to separate multiple statements on the same line.
3. **Code Blocks:** Code blocks in Python are defined by indentation, not by braces `{}`.

Example Code:



Explanation:

- The `if` statement checks if `x` is less than `y`. If true, it prints a message; otherwise, it executes the `else` block.

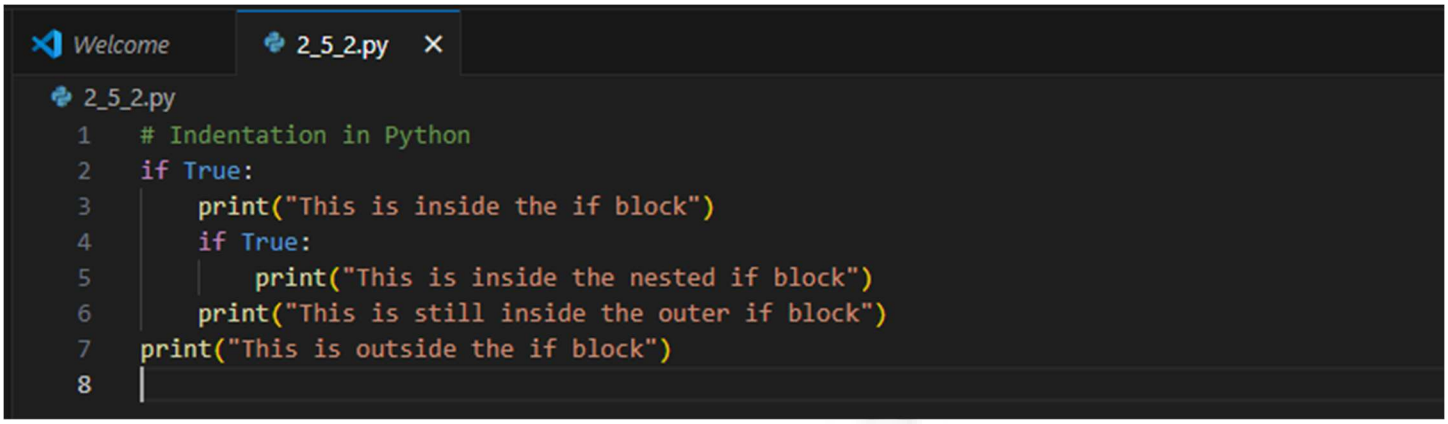
Output:

2.5.2 Indentation in Python

Indentation is crucial in Python as it defines the scope of loops, functions, classes, and other control structures.

- **Standard Indentation:** Python typically uses 4 spaces for indentation.
- **Consistency:** Always maintain consistent indentation throughout your code.

Example Code:



Explanation:

- The code inside the `if` block is indented, indicating that it belongs to the block. The final `print` statement is outside the block, so it's not indented.

Output:

```
This is inside the if block
This is inside the nested if block
This is still inside the outer if block
This is outside the if block
```

Chapter 3: Control Structures

Control structures are fundamental to any programming language, allowing developers to control the flow of the program's execution based on conditions, loops, and exception handling. This chapter covers the essential control structures in Python, including conditional statements, loops, and exception handling.

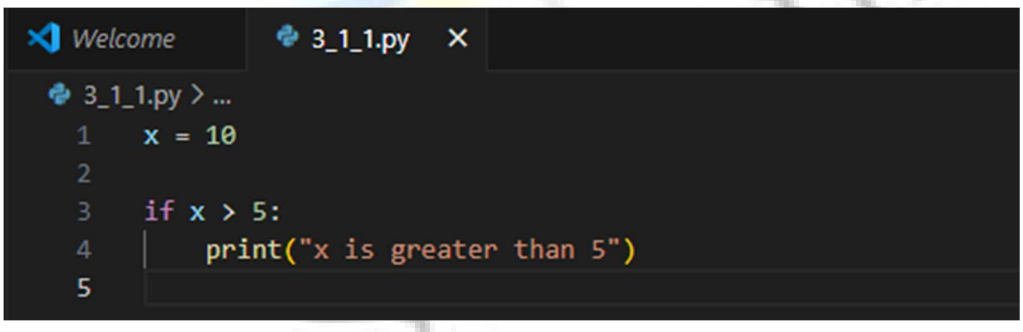
3.1 Conditional Statements: `if`, `elif`, `else`

Conditional statements allow you to execute certain pieces of code based on specific conditions. Python provides `if`, `elif`, and `else` statements to implement decision-making in your code.

3.1.1 The `if` Statement

The `if` statement is used to test a condition. If the condition is `True`, the code block under the `if` statement is executed.

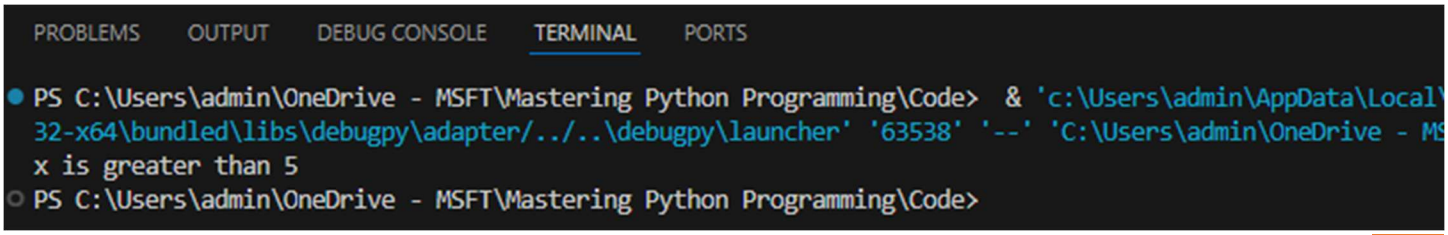
Example Code:



Explanation:

- The `if` statement checks whether `x` is greater than 5. Since the condition is `True`, the message is printed.

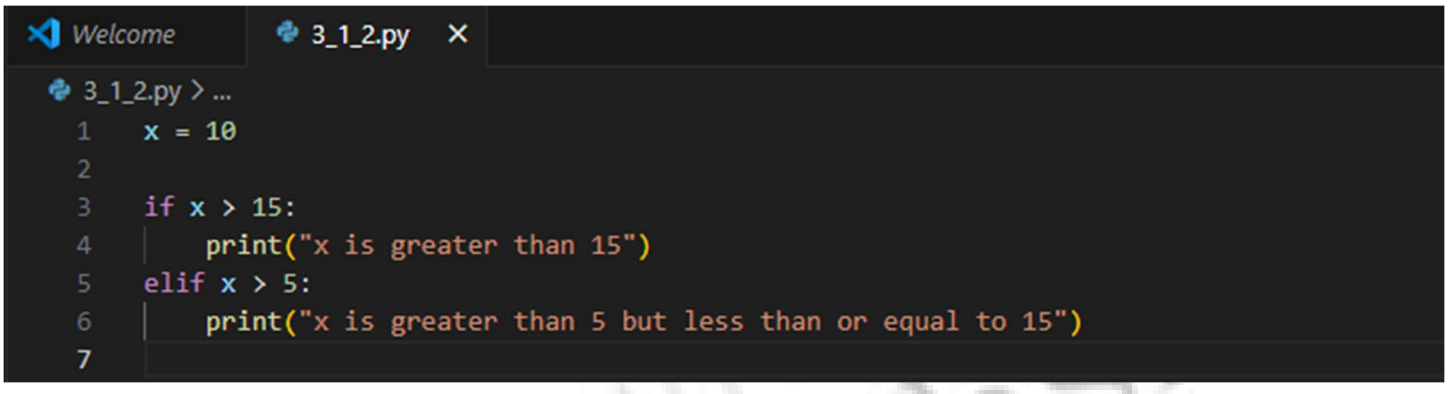
Output:



3.1.2 The `elif` Statement

The `elif` (short for "else if") statement allows you to check multiple conditions. If the first `if` condition is `False`, the `elif` condition is checked.

Example Code:



Explanation:

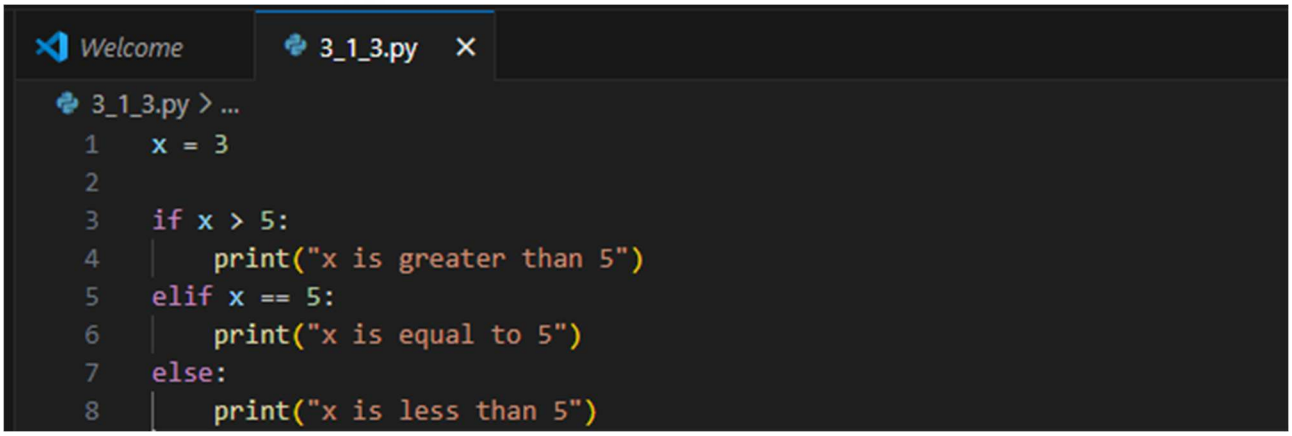
- The first condition `x > 15` is `False`, so Python moves to the `elif` statement, which is `True`. Therefore, the message is printed.

Output:

3.1.3 The `else` Statement

The `else` statement is used to define a block of code that will run if none of the previous conditions are `True`.

Example Code:



Explanation:

- Since both the `if` and `elif` conditions are `False`, the `else` block is executed.

Output:

```
x is less than 5
```

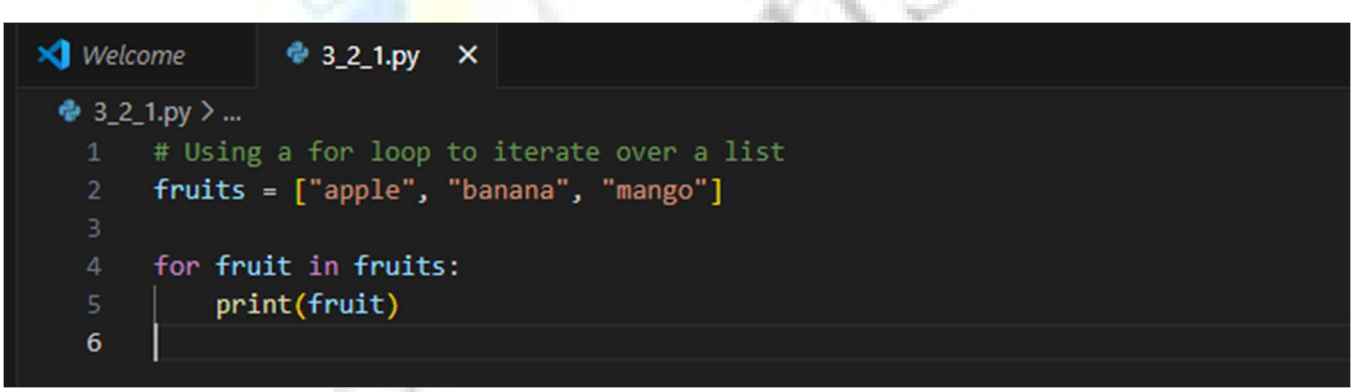
3.2 Loops: `for` Loop, `while` Loop

Loops are used to execute a block of code repeatedly. Python provides two types of loops: `for` loops and `while` loops.

3.2.1 `for` Loop

A `for` loop is used to iterate over a sequence (like a list, tuple, string, or range) and execute a block of code for each item in the sequence.

Example Code:



Explanation:

- The `for` loop iterates over each item in the `fruits` list and prints it.

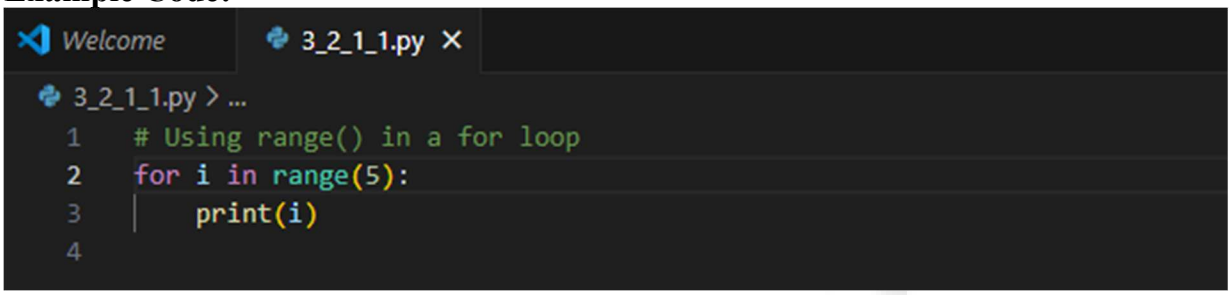
Output:

```
apple
banana
mango
```

3.2.1.1 `range()` Function

The `range()` function generates a sequence of numbers, which is commonly used in `for` loops.

Example Code:



Explanation:

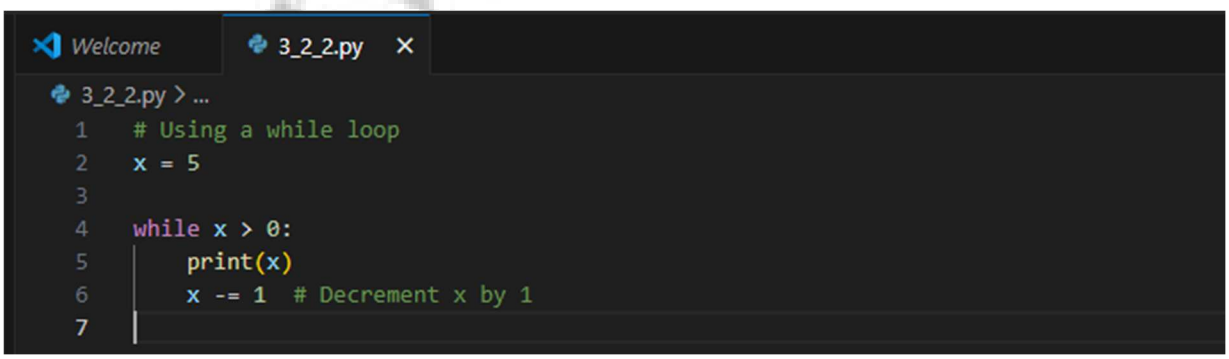
- The `range(5)` function generates numbers from 0 to 4, and the `for` loop iterates over these numbers.

Output:

3.2.2 `while` Loop

A `while` loop continues to execute a block of code as long as a specified condition remains `True`.

Example Code:



Explanation:

- The `while` loop checks whether `x` is greater than 0. As long as the condition is `True`, it prints the value of `x` and then decrements it by 1.

Output:

```
5
4
3
2
1
```

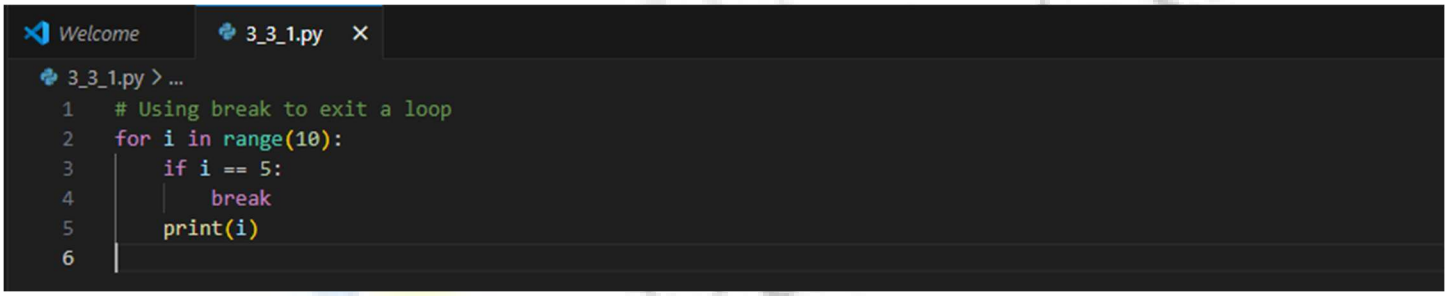
3.3 Break and Continue Statements

The `break` and `continue` statements are used to control the flow of loops.

3.3.1 The `break` Statement

The `break` statement is used to exit a loop prematurely, regardless of the loop's condition.

Example Code:



Explanation:

- The loop will print numbers from 0 to 4. When `i` equals 5, the `break` statement exits the loop.

Output:

```
0
1
2
3
4
```

3.3.2 The `continue` Statement

The `continue` statement skips the rest of the code inside the loop for the current iteration and jumps to the next iteration of the loop.

Example Code:

```python
# Using continue to skip an iteration
for i in range(10):
    if i % 2 == 0:
        continue
    print(i)

```

Explanation:

- The loop will print only odd numbers because the `continue` statement skips the even numbers.

Output:

```
1
3
5
7
9
```

3.4 Exception Handling: `try`, `except`, `finally`

Exception handling in Python is a way to handle errors that occur during the execution of a program. This prevents the program from crashing and allows it to handle the error gracefully.

3.4.1 The `try` and `except` Blocks

The `try` block contains the code that might throw an exception, and the `except` block contains the code that handles the exception.

Example Code:

```python
# Handling division by zero error
try:
    result = 10 / 0
except ZeroDivisionError:
    print("You can't divide by zero!")

```

Explanation:

- The `try` block attempts to divide 10 by 0, which raises a `ZeroDivisionError`. The `except` block catches the error and prints an error message.

Output:

```
You can't divide by zero!
```

3.4.2 The `finally` Block

The `finally` block contains code that will run no matter what, even if an exception occurs. It's typically used for cleanup actions like closing files or releasing resources.

Example Code:

Welcome | 3_4_2.py

3_4_2.py > ...

```python
# Using finally to clean up
try:
    file = open("sample.txt", "r")
    content = file.read()
except FileNotFoundError:
    print("File not found!")
finally:
    print("Closing the file.")
    file.close()
```

Explanation:

- The `finally` block ensures that the file is closed, regardless of whether an exception occurs.

Output:

```
File not found!
Closing the file.
```

Chapter 4: Functions and Modules

Functions and modules are fundamental components of Python programming. They help organize code into reusable blocks, making it more manageable and modular. This chapter covers how to define functions, use function arguments, work with lambda functions, and import and create modules. It also explores how to use PIP to install external modules.

4.1 Defining Functions

A function is a block of reusable code that performs a specific task. Functions help break down complex problems into smaller, manageable pieces.

4.1.1 Syntax of a Function

A function is defined using the `def` keyword, followed by the function name, parentheses, and a colon. The code block within the function is indented.

Example Code:

Welcome | 4_1_1.py

4_1_1.py > ...

```
1  def greet(name):
2      print(f"Hello, {name}!")
3
4  # Call the function
5  greet("Suryanshsk")
6
```

Explanation:

- `greet` is the function name.
- `name` is a parameter that the function accepts.
- The function prints a greeting message using the provided name.

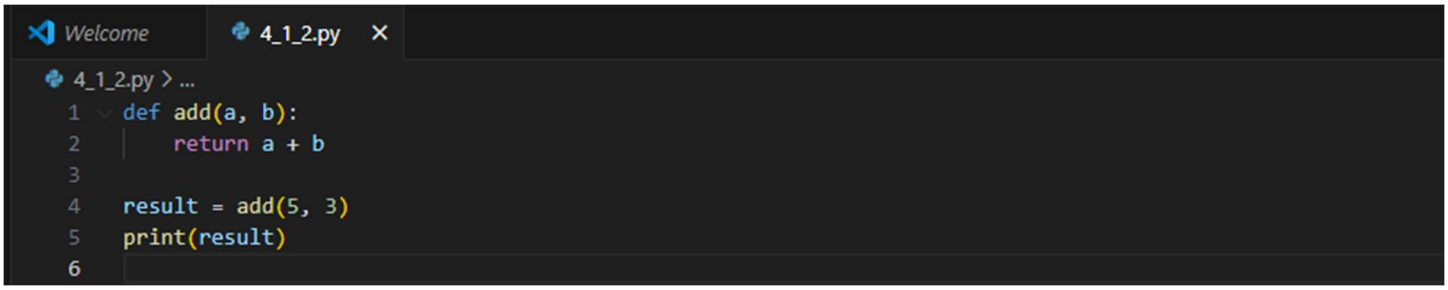
Output:

```
Hello, Suryanshsk!
```

4.1.2 Calling a Function

To use a function, you call it by its name followed by parentheses, passing any required arguments.

Example Code:



Explanation:

- The `add` function takes two arguments `a` and `b`, adds them, and returns the result.
- The result is then printed.

Output:

```
8
```

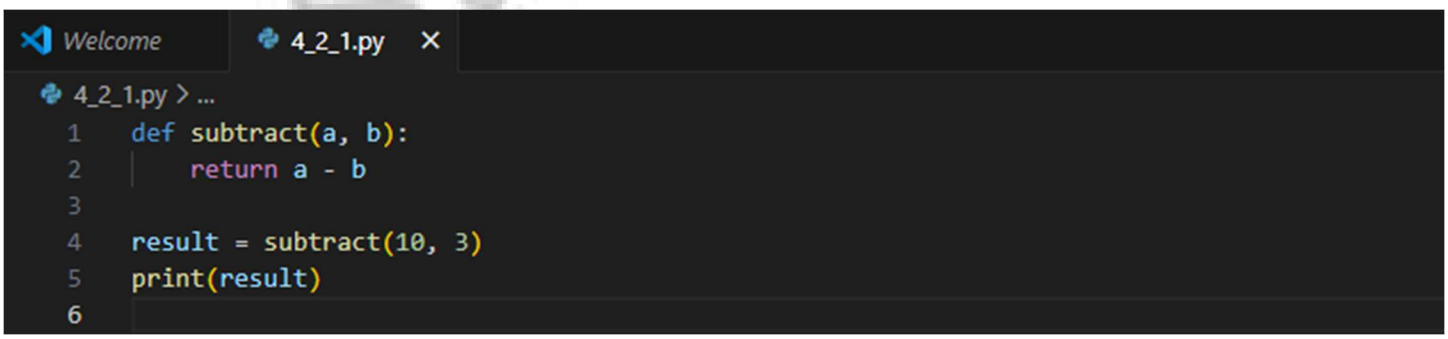
4.2 Function Arguments and Return Values

Functions can take arguments and return values, which allow them to work with different inputs and produce outputs.

4.2.1 Positional Arguments

Positional arguments are the most straightforward way to pass values to a function. The values are assigned to the parameters in the order they are passed.

Example Code:



Explanation:

- The function subtracts `b` from `a` and returns the result.

Output:

```
7
```

4.2.2 Keyword Arguments

Keyword arguments allow you to specify the argument values by name, which can make the function call more readable.

Example Code:

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

result = multiply(a=4, b=6)
print(result)
```

Explanation:

- The values of `a` and `b` are specified using their parameter names.

Output:

```
24
```

4.2.3 Default Arguments

You can provide default values for arguments, making them optional when the function is called.

Example Code:

```python
def power(base, exponent=2):
    return base ** exponent

print(power(5))
print(power(5, 3))
```

Explanation:

- The `exponent` parameter has a default value of 2.
- If `exponent` is not provided, the function uses the default value.

Output:

```
25
125
```

4.2.4 Variable-Length Arguments

You can use `*args` and `**kwargs` to pass a variable number of arguments to a function.

Example Code:

```python
def summarize(*args):
    return sum(args)

result = summarize(1, 2, 3, 4)
print(result)
```

Explanation:

- The `*args` syntax allows the function to accept any number of positional arguments, which are then summed.

Output:

```
10
```

4.3 Lambda Functions

Lambda functions, also known as anonymous functions, are small, one-line functions defined using the `lambda` keyword. They are often used for short, simple operations.

4.3.1 Syntax of Lambda Functions

The syntax for a lambda function is:

```python
lambda arguments: expression
```

Example Code:

```python
lambda arguments: expression
square = lambda x: x ** 2
print(square(5))
```

Explanation:

- The lambda function takes one argument `x` and returns `x` squared.

Output:
```
25
```

4.3.2 Using Lambda Functions

Lambda functions are commonly used in situations where a simple function is needed, such as in sorting or filtering data.

Example Code:

4_3_2.py
```python
numbers = [2, 3, 1, 4]
numbers.sort(key=lambda x: x)
print(numbers)

```

Explanation:

- The `sort` method uses a lambda function to sort the list.

Output:
```
[1, 2, 3, 4]
```

4.4 Importing Modules

Modules are files containing Python code (variables, functions, classes) that can be imported and used in other Python programs. Python has a rich standard library of modules for various tasks.

4.4.1 Importing a Module

You can import a module using the `import` keyword.

Example Code:

4_4_1.py
```python
import math

print(math.sqrt(16))
```

Explanation:

- The `math` module is imported, and its `sqrt` function is used to calculate the square root of 16.

Output:

```
4.0
```

4.4.2 Importing Specific Functions or Variables

You can also import specific functions or variables from a module using the `from` keyword.

Example Code:

4_4_2.py

```python
from math import pi, sin

print(pi)
print(sin(pi / 2))

```

Explanation:

- The `pi` constant and `sin` function are imported directly from the `math` module.

Output:

```
3.141592653589793
1.0
```

4.4.3 Aliasing Modules

You can give a module or function an alias using the `as` keyword, which can make your code more concise.

Example Code:

4_4_3.py

```python
import numpy as np

array = np.array([1, 2, 3])
print(array)
```

Explanation:

- The `numpy` module is imported with the alias `np`.

Output:

```
[1 2 3]
```

4.5 Creating Your Own Modules

You can create your own modules by writing functions, variables, or classes in a Python file and importing them into other files.

4.5.1 Writing a Module

To create a module, simply save a Python script with a `.py` extension.

Example Code (`mymodule.py`):

mymodule.py

```python
def greet(name):
    return f"Hello, {name}!"

```

4.5.2 Importing Your Module

You can import your custom module just like any other Python module.

Example Code (in another file):

4_5_2.py

```python
import mymodule

print(mymodule.greet("Suryanshsk"))

```

Explanation:

- The `greet` function from `mymodule.py` is used in another Python script.

Output:

```
Hello, Suryanshsk!
```

4.6 PIP and Installing External Modules

PIP is Python's package installer, used to install external modules not included in the standard library.

4.6.1 Installing a Module with PIP

You can install a module using the `pip install` command.

Example Command:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\Users\admin\OneDrive - MSFT\Mastering Python Programming\Code> pip install requests
```

Explanation:

- This command installs the `requests` module, which is used for making HTTP requests.

4.6.2 Using an Installed Module

Once installed, you can import and use the module in your code.

Example Code:

4_6_2.py

```python
import requests

response = requests.get("https://api.github.com")
print(response.status_code)
```

Explanation:

- The `requests` module is used to make a GET request to the GitHub API.

Output:

```
200
```

Chapter 5: Data Structures

Data structures are a way of organizing and storing data so that it can be accessed and modified efficiently. This chapter covers Python's built-in data structures: lists, tuples, sets, and dictionaries. It also introduces list comprehensions for creating new lists in a concise way.

5.1 Lists

Lists are ordered collections of items, which can be of any data type. Lists are mutable, meaning their contents can be changed after creation.

5.1.1 Creating Lists

Lists are created by placing items inside square brackets `[]`, separated by commas.

Example Code:

5_1_1.py

```python
fruits = ["apple", "banana", "cherry"]
print(fruits)
```

Explanation:

- A list of fruits is created and printed.

Output:

```
['apple', 'banana', 'cherry']
```

5.1.2 Accessing List Elements

You can access individual elements in a list by their index, starting at 0.

Example Code:

5_1_2.py

```python
fruits = ["apple", "banana", "Mango"]
print(fruits[0])  # First element
print(fruits[-1])  # Last element
```

Explanation:

- The first element (`"apple"`) and the last element (`"cherry"`) are accessed using their respective indices.

Output:

```
apple
Mango
```

5.1.3 Modifying Lists

Lists are mutable, so you can change their elements after creation.

Example Code:

5_1_3.py

```python
fruits = ["apple", "banana", "Mango"]
fruits[1] = "blueberry"
print(fruits)
```

Explanation:

- The second element of the list is changed from `"banana"` to `"blueberry"`.

Output:

```
['apple', 'blueberry', 'Mango']
```

5.1.4 List Methods

Python provides several methods to work with lists, such as `append`, `remove`, `sort`, and more.

Example Code:

5_1_4.py

```python
fruits = ["apple", "banana", "Mango"]
fruits.append("orange")
fruits.remove("apple")
fruits.sort()
print(fruits)
```

Explanation:

- `"orange"` is added to the list, `"apple"` is removed, and the list is sorted alphabetically.

Output:
```
['Mango', 'banana', 'orange']
```

5.2 Tuples

Tuples are similar to lists but are immutable, meaning their contents cannot be changed after creation. Tuples are often used to group related data.

5.2.1 Creating Tuples

Tuples are created by placing items inside parentheses `()`, separated by commas.

Example Code:

5_2_1.py
```
point = (10, 20)
print(point)
```

Explanation:

- A tuple representing a point with coordinates (10, 20) is created.

Output:
```
(10, 20)
```

5.2.2 Accessing Tuple Elements

You can access individual elements in a tuple by their index, just like lists.

Example Code:

5_2_2.py
```
point = (10, 20)
print(point[0])  # First element
```

Explanation:

- The first element (`10`) of the tuple is accessed.

Output:
```
10
```

5.2.3 Tuple Unpacking

You can unpack the values of a tuple into separate variables.

Example Code:

5_2_3.py
```python
point = (10, 20)
x, y = point
print(f"x: {x}, y: {y}")

```

Explanation:

- The values of the tuple `point` are unpacked into the variables `x` and `y`.

Output:
```
x: 10, y: 20
```

5.3 Sets

Sets are unordered collections of unique items. They are useful when you need to ensure that an element is present only once in a collection.

5.3.1 Creating Sets

Sets are created by placing items inside curly braces `{}`, separated by commas, or by using the `set()` function.

Example Code:

5_3_1.py
```python
numbers = {1, 2, 3, 4, 4, 5}
print(numbers)
```

Explanation:

- A set is created with unique elements. The duplicate `4` is automatically removed.

Output:

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

5.3.2 Set Operations

Sets support mathematical operations like union, intersection, difference, and symmetric difference.

Example Code:

5_3_2.py

```python
a = {1, 2, 3}
b = {3, 4, 5}

print(a | b)  # Union
print(a & b)  # Intersection
print(a - b)  # Difference
print(a ^ b)  # Symmetric Difference
```

Explanation:

- The operations demonstrate the various ways sets can be combined and compared.

Output:

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

5.4 Dictionaries

Dictionaries are collections of key-value pairs, where each key is unique, and is associated with a specific value. Dictionaries are mutable, so their contents can be changed after creation.

5.4.1 Creating Dictionaries

Dictionaries are created by placing key-value pairs inside curly braces `{}`, separated by commas. The key and value are separated by a colon `:`.

Example Code:

```python
student = {"name": "Suryanshsk", "age": 18, "grade": "A"}
print(student)
```

Explanation:

- A dictionary representing a student is created, with keys `"name"`, `"age"`, and `"grade"`.

Output:

```
{'name': 'Suryanshsk', 'age': 18, 'grade': 'A'}
```

5.4.2 Accessing Dictionary Elements

You can access the value associated with a specific key by using square brackets `[]`.

Example Code:

```python
student = {"name": "Suryanshsk", "age": 18, "grade": "A"}
print(student["name"])
```

Explanation:

- The value associated with the key `"name"` is accessed.

Output:

```
Suryanshsk
```

5.4.3 Modifying Dictionaries

You can add, modify, or remove key-value pairs in a dictionary.

Example Code:

```python
student = {"name": "Suryanshsk", "age": 18, "grade": "A"}
student["age"] = 21  # Modify
student["major"] = "Computer Science"  # Add
del student["grade"]  # Remove
print(student)
```

Explanation:

- The `"age"` value is updated, a new key `"major"` is added, and the `"grade"` key is removed.

Output:
```
{'name': 'Suryanshsk', 'age': 21, 'major': 'Computer Science'}
```

5.5 List Comprehensions

List comprehensions provide a concise way to create lists. They consist of brackets containing an expression followed by a `for` clause and optionally `if` clauses.

5.5.1 Basic List Comprehension

You can create a list of squares using a list comprehension.

Example Code:

5_5_1.py
```python
squares = [x ** 2 for x in range(1, 6)]
print(squares)
```

Explanation:

- A list of squares of numbers from 1 to 5 is created using list comprehension.

Output:
```
[1, 4, 9, 16, 25]
```

5.5.2 List Comprehension with Conditionals

You can include an `if` statement to filter elements.

Example Code:

5_5_2.py
```python
evens = [x for x in range(10) if x % 2 == 0]
print(evens)
```

Explanation:

- A list of even numbers from 0 to 9 is created.

Output:

```
[0, 2, 4, 6, 8]
```

6. Object-Oriented Programming (OOP)

Object-Oriented Programming (OOP) is a programming paradigm that uses objects and classes to design and structure code. It provides a clear modular structure for programs, making it easier to manage complexity.

6.1 Classes and Objects

6.1.1 Defining a Class

A class is a blueprint for creating objects. It encapsulates data for the object and methods to manipulate that data.

Example Code:

6_1_1.py

```python
class Dog:
    def __init__(self, name, breed):
        self.name = name
        self.breed = breed

    def bark(self):
        return f"{self.name} says Woof!"
```

Explanation:

- The `Dog` class defines two attributes (`name`, `breed`) and one method (`bark`).

6.1.2 Creating an Object

An object is an instance of a class. You can create multiple objects from a single class.

Example Code:

6_1_2.py

```python
class Dog:
    def __init__(self, name, breed):
        self.name = name
        self.breed = breed

    def bark(self):
        return f"{self.name} says Woof!"

my_dog = Dog("Rex", "German Shepherd")
print(my_dog.bark())
```

Explanation:

- `my_dog` is an object of the `Dog` class, with the name `"Rex"` and breed `"German Shepherd"`.

Output:

```
Rex says Woof!
```

6.2 Constructors and Destructors

6.2.1 Constructors

A constructor is a special method that is automatically called when an object is created. In Python, the `__init__` method is used as a constructor.

Example Code:

6_2_1.py

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

    def area(self):
        return 3.14 * self.radius ** 2
```

Explanation:

- The `Circle` class uses the `__init__` constructor to initialize the radius of the circle.

6.2.2 Destructors

A destructor is a method that is automatically called when an object is deleted or goes out of scope. In Python, the `__del__` method is used as a destructor.

Example Code:

6_2_2.py

```python
class Example:
    def __init__(self, value):
        self.value = value
        print(f"Object created with value {self.value}")

    def __del__(self):
        print(f"Object with value {self.value} is being deleted")
```

Explanation:

- The `Example` class uses the `__del__` destructor to print a message when the object is deleted.

6.3 Inheritance

Inheritance allows a class to inherit attributes and methods from another class, promoting code reusability.

6.3.1 Single Inheritance

In single inheritance, a class (child) inherits from a single parent class.

Example Code:

6_3_1.py

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

    def speak(self):
        return f"{self.name} makes a sound"

class Dog(Animal):
    def speak(self):
        return f"{self.name} barks"
```

Explanation:

- The `Dog` class inherits from the `Animal` class and overrides the `speak` method.

6.3.2 Multiple Inheritance

In multiple inheritance, a class can inherit from more than one parent class.

Example Code:

6_3_2.py

```python
class Walker:
    def walk(self):
        return "Walking"

class Swimmer:
    def swim(self):
        return "Swimming"

class Amphibian(Walker, Swimmer):
    pass
```

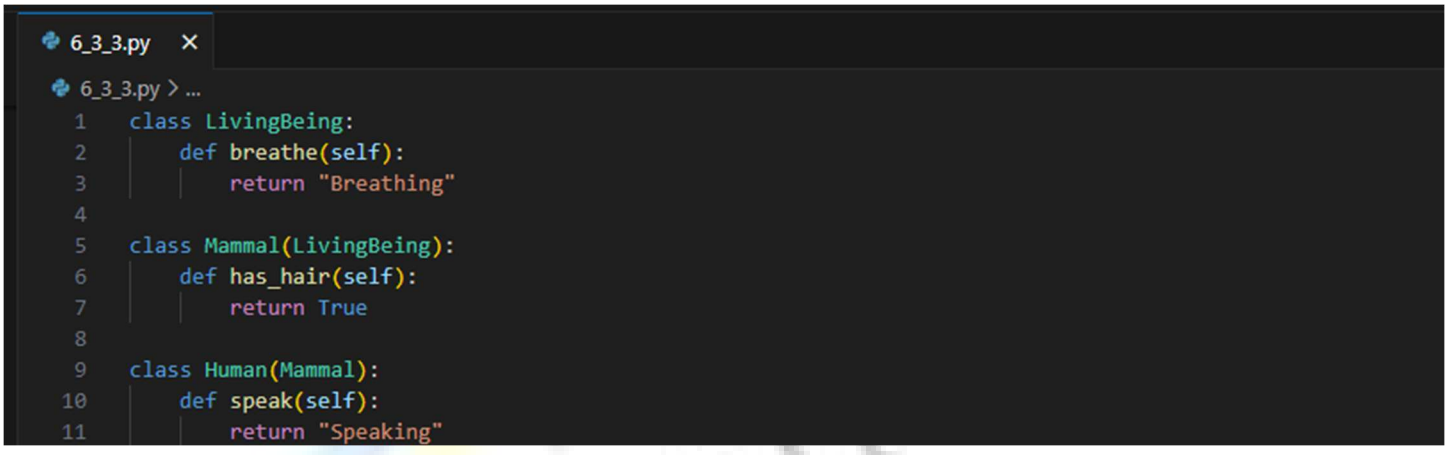
Explanation:

- The `Amphibian` class inherits from both `Walker` and `Swimmer` classes.

6.3.3 Multilevel Inheritance

In multilevel inheritance, a class is derived from another derived class.

Example Code:



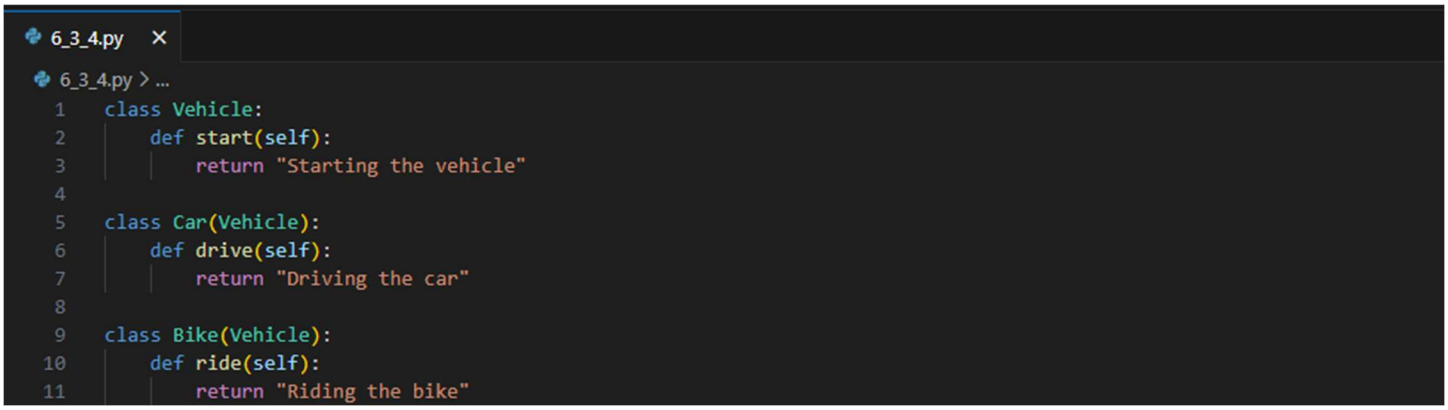
Explanation:

- The `Human` class inherits from `Mammal`, which in turn inherits from `LivingBeing`.

6.3.4 Hierarchical Inheritance

In hierarchical inheritance, multiple classes inherit from a single parent class.

Example Code:



Explanation:

- Both `Car` and `Bike` inherit from the `Vehicle` class.

6.4 Polymorphism

Polymorphism allows different classes to be treated as instances of the same class through a common interface. It promotes flexibility and integration.

6.4.1 Method Overriding

Method overriding allows a child class to provide a specific implementation of a method already defined in its parent class.

Example Code:

6_4_1.py

```python
class Bird:
    def fly(self):
        return "Bird is flying"

class Penguin(Bird):
    def fly(self):
        return "Penguins can't fly"
```

Explanation:

- The `Penguin` class overrides the `fly` method of the `Bird` class.

6.4.2 Method Overloading

Python does not support method overloading by default, but you can achieve it using default arguments.

Example Code:

6_4_2.py

```python
class Calculator:
    def add(self, a, b=0):
        return a + b
calc = Calculator()
print(calc.add(10))  # 10
print(calc.add(10, 5))  # 15
```

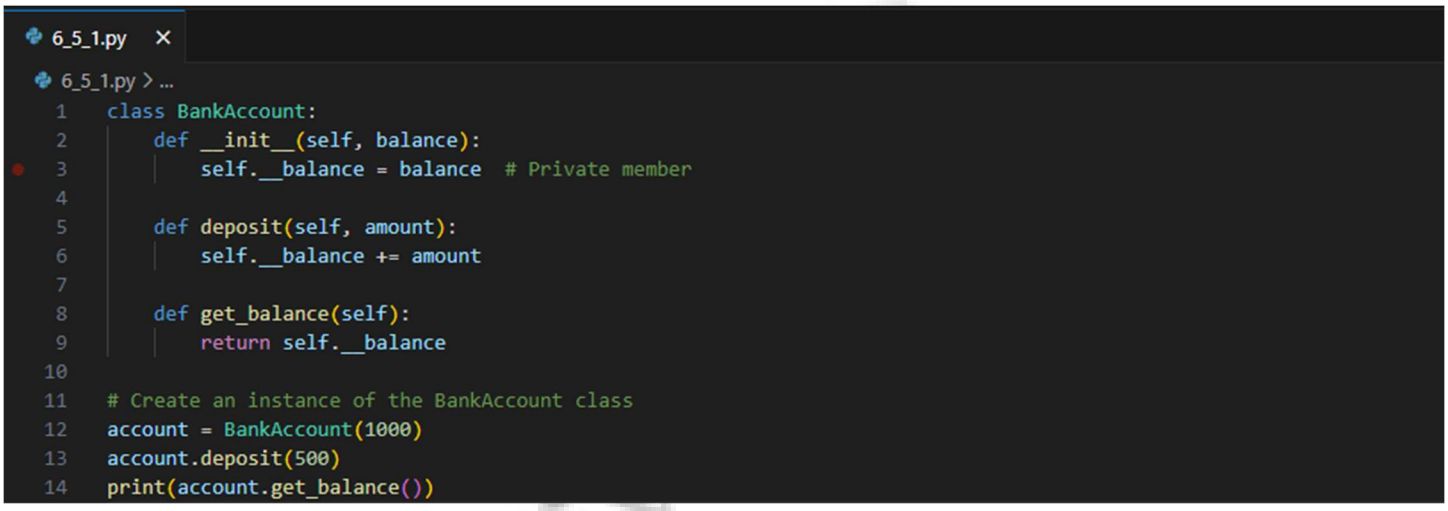
6.5 Encapsulation

Encapsulation is the mechanism of restricting access to certain components of an object and only exposing a limited interface to the user.

6.5.1 Private and Protected Members

In Python, private members are denoted by a double underscore prefix (__), while protected members use a single underscore (_).

Example Code:



Explanation:

- The `__balance` attribute is private and can only be accessed through the `deposit` and `get_balance` methods.

Output:

```
1500
```

6.6 Abstract Classes and Interfaces

Abstract classes cannot be instantiated and are meant to be subclassed. They can contain abstract methods, which must be implemented by subclasses.

6.6.1 Creating Abstract Classes

In Python, abstract classes are created using the `ABC` (Abstract Base Class) module.

Example Code:

6_6_1.py

```python
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def make_sound(self):
        pass

class Dog(Animal):
    def make_sound(self):
        return "Woof!"
```

Explanation:

- The `Animal` class is an abstract class with an abstract method `make_sound`. The `Dog` class implements this method.

6.6.2 Interfaces

Python does not have a separate interface keyword, but abstract classes with only abstract methods can serve as interfaces.

Example Code:

6_6_2.py

```python
from abc import ABC, abstractmethod
class Drawable(ABC):
    @abstractmethod
    def draw(self):
        pass

class Circle(Drawable):
    def draw(self):
        return "Drawing a circle"
```

Explanation:

- The `Drawable` class acts as an interface with the `draw` method, which is implemented by the `Circle` class.

7. File Handling

File handling is an essential part of any application that needs to store and retrieve data. Python provides a variety of functions and modules to work with files.

7.1 Reading from and Writing to Files

7.1.1 Opening a File

In Python, the `open()` function is used to open files.

Example Code:

```python
file = open("example.txt", "r")  # Open a file in read mode
# At the Place of example.txt you have to write te path of our file
```

Explanation:

- The file `example.txt` is opened in read mode (`"r"`).

7.1.2 Reading a File

You can read the contents of a file using methods like `read()`, `readline()`, or `readlines()`.

Example Code:

```python
file = open("example.txt", "r")
content = file.read()
print(content)
file.close()
```

Explanation:

- The entire content of the file is read and printed, and the file is closed.

7.1.3 Writing to a File

To write data to a file, open it in write mode (`"w"`), append mode (`"a"`), or write binary mode (`"wb"`).

Example Code:

```python
file = open("example.txt", "w") # w means write
file.write("Hello, World!")
file.close()
```

Explanation:

- `"Hello, World!"` is written to the file `example.txt`.

7.2 Working with CSV Files

CSV (Comma Separated Values) files are commonly used for storing tabular data.

7.2.1 Reading CSV Files

Python's `csv` module makes it easy to read and write CSV files.

CSV File:

	A	B
1	Name	Age
2	Suryanshsk	18

Example Code:

```python
import csv  #pip install csv

with open("data.csv", "r") as file: # At the Place Of data.csv You Have to write your path
    reader = csv.reader(file)
    for row in reader:
        print(row)
```

Output:

```
['Name', 'Age ']
['Suryanshsk', '18']
```

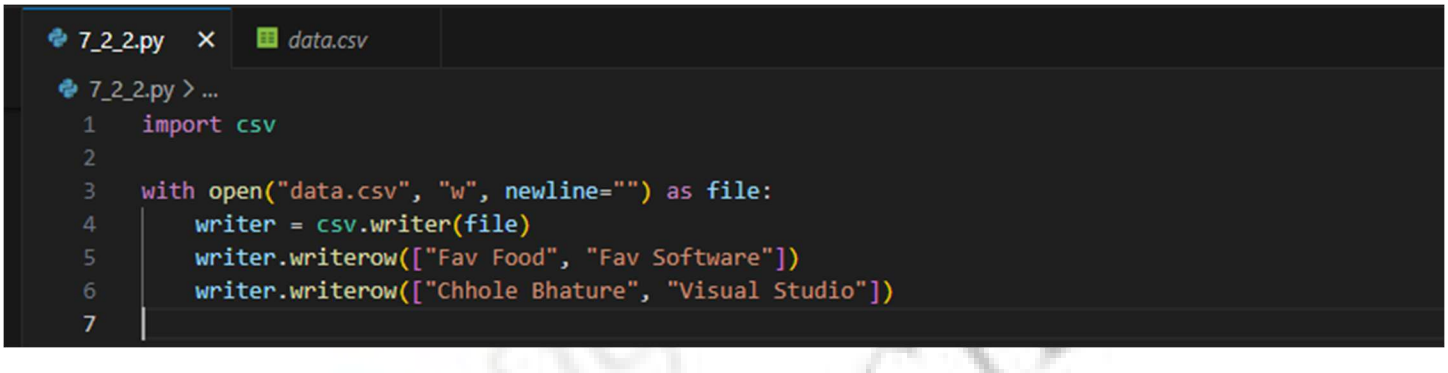
Explanation:

- This code reads each row of the `data.csv` file and prints it.

7.2.2 Writing to CSV Files

You can also write to CSV files using the `csv` module.

Example Code:



Output:

	A	B
1	Fav Food	Fav Software
2	Chhole Bhature	Visual Studio

Explanation:

- This code writes two rows to the `data.csv` file.

7.3 File Methods and Operations

7.3.1 Common File Methods

Python provides various methods to manipulate files:

- `read()`: Reads the entire file.
- `readline()`: Reads a single line from the file.
- `write()`: Writes a string to the file.
- `close()`: Closes the file.

7.3.2 File Operations

You can perform operations like renaming, deleting, and copying files using the `os` and `shutil` modules.

Example Code:

```python
import os

os.rename("old_name.txt", "new_name.txt")  # Rename a file
os.remove("example.txt")  # Delete a file
```

Explanation:

- The `os.rename()` method renames a file, and `os.remove()` deletes it.

7.4 Exception Handling in File Operations

Exception handling ensures that your program can handle errors that may occur during file operations.

7.4.1 Handling File Exceptions

You can handle file-related exceptions using the `try-except` block.

Example Code:

```python
try:
    file = open("example.txt", "r")
    content = file.read()
except FileNotFoundError:
    print("File not found!")
finally:
    file.close()
```

Explanation:

- The code attempts to open and read a file. If the file does not exist, it catches a `FileNotFoundError`.

8. Advanced Python Concepts

Python provides advanced features that enhance its power and flexibility. These concepts include generators, decorators, context managers, and more.

8.1 Generators and Iterators

8.1.1 Generators

Generators are functions that return an iterable set of items, one at a time, in a lazy manner.

Example Code:

8_1_1.py

```python
def count_up_to(max):
    count = 1
    while count <= max:
        yield count
        count += 1

for number in count_up_to(5):
    print(number)
```

Explanation:

- The `count_up_to()` function is a generator that yields numbers from 1 to `max`.

8.1.2 Iterators

Iterators are objects that can be iterated upon. They implement the `__iter__()` and `__next__()` methods.

Example Code:

8_1_2.py

```python
class Counter:
    def __init__(self, max):
        self.max = max
        self.current = 1

    def __iter__(self):
        return self

    def __next__(self):
        if self.current <= self.max:
            self.current += 1
            return self.current - 1
        else:
            raise StopIteration

counter = Counter(5)
for num in counter:
    print(num)
```

Explanation:

- The `Counter` class is an iterator that produces numbers from 1 to `max`.

8.2 Decorators

Decorators are a powerful tool in Python that allow you to modify the behavior of a function or class method.

8.2.1 Function Decorators

Function decorators allow you to extend or alter the behavior of functions.

Example Code:

8_2_1.py

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

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

say_hello()
```

Explanation:

- The `my_decorator` function is applied to `say_hello()` using the `@` syntax.

Output:

```
Something is happening before the function is called.
Hello!
Something is happening after the function is called.
```

8.2.2 Class Method Decorators

Decorators can also be applied to class methods to modify their behavior.

Example Code:

8_2_2.py

```python
def method_decorator(func):
    def wrapper(self):
        print(f"Before {func.__name__} is called")
        result = func(self)
        print(f"After {func.__name__} is called")
        return result
    return wrapper

class MyClass:
    @method_decorator
    def greet(self):
        print("Greetings from MyClass!")

obj = MyClass()
obj.greet()
```

Explanation:

- The `method_decorator` modifies the `greet` method of `MyClass`.

Output:

```
Before greet is called
Greetings from MyClass!
After greet is called
```

8.3 Context Managers

Context managers allow you to allocate and release resources precisely when you want to.

8.3.1 Using `with` Statements

The `with` statement is used with context managers to ensure that resources are properly managed.

Example Code:

8_3_1.py

```python
with open("example.txt", "w") as file:
    file.write("Hello, World!")
```

Explanation:

- The `with` statement automatically closes the file when the block inside it is exited.

8.3.2 Creating Custom Context Managers

You can create custom context managers by defining the `__enter__` and `__exit__` methods.

Example Code:

8_3_2.py

```python
class MyContextManager:
    def __enter__(self):
        print("Entering the context")
        return self

    def __exit__(self, exc_type, exc_value, traceback):
        print("Exiting the context")

with MyContextManager() as manager:
    print("Inside the context")
```

Explanation:

- The `MyContextManager` class defines a custom context manager.

Output:

```
Entering the context
Inside the context
Exiting the context
```

8.4 Working with Dates and Times

Python's `datetime` module provides classes for manipulating dates and times.

8.4.1 Date and Time Basics

You can create and manipulate date and time objects using the `datetime` module.

Example Code:

8_4_1.py

```python
from datetime import datetime

now = datetime.now()
print("Current date and time:", now)

today = datetime.today()
print("Today's date:", today)
```

Explanation:

- The `datetime.now()` function returns the current date and time.

8.4.2 Formatting Dates and Times

You can format date and time objects using the `strftime` method.

Example Code:

8_4_2.py

```python
from datetime import datetime

now = datetime.now()
print("Current date and time:", now)

today = datetime.today()
print("Today's date:", today)

formatted_date = now.strftime("%Y-%m-%d %H:%M:%S")
print("Formatted date and time:", formatted_date)
```

Explanation:

- The `strftime` method formats the date and time according to the specified format.

8.4.3 Parsing Dates and Times

You can parse a string into a date and time object using the `strptime` method.

Example Code:

8_4_3.py

```python
from datetime import datetime
date_string = "2023-08-12 10:30:00"
parsed_date = datetime.strptime(date_string, "%Y-%m-%d %H:%M:%S")
print("Parsed date and time:", parsed_date)
```

Explanation:

- The `strptime` method parses a date string into a `datetime` object.

8.5 Regular Expressions

Regular expressions (regex) are patterns used to match character combinations in strings. Python's `re` module provides support for regex operations.

8.5.1 Basic Regex Operations

You can use the `re` module to perform basic regex operations like searching, matching, and replacing.

Example Code:

8_5_1.py

```python
import re

text = "The quick brown fox jumps over the lazy dog"
pattern = r"quick"
match = re.search(pattern, text)
if match:
    print("Pattern found:", match.group())
```

Explanation:

- The `re.search()` function searches for the pattern `"quick"` in the text and returns a match object if found.

8.5.2 Matching and Extracting Data

You can use regex to match and extract specific patterns from text.

Example Code:

8_5_2.py

```python
import re
text = "Contact me at suryanshsk@hotmail.com"
pattern = r"\b[\w.%+-]+@[\w.-]+\.[a-zA-Z]{2,}\b"
email = re.findall(pattern, text)
print("Extracted email:", email)
```

Explanation:

- The regex pattern matches an email address in the text and returns it.

8.5.3 Replacing and Splitting Strings

You can use regex to replace and split strings based on specific patterns.

Example Code:

8_5_3.py

```python
import re
text = "The rain in Spain"
new_text = re.sub(r"ain", "XYZ", text)
print("Replaced text:", new_text)
```

Explanation:

- The `re.sub()` function replaces all occurrences of `"ain"` with `"XYZ"` in the text.

9. Python for Web Development

Python is a versatile language that's popular in web development, both for backend and full-stack development.

9.1 Introduction to Web Development with Python

Web development with Python involves creating websites or web applications using Python's extensive libraries and frameworks.

9.1.1 Frontend vs. Backend

- **Frontend:** The part of a web application that users interact with, usually built using HTML, CSS, and JavaScript.
- **Backend:** The server-side part of a web application, handling business logic, database interactions, and more.

9.1.2 Why Use Python for Web Development?

- **Ease of Use:** Python's syntax is simple and easy to learn.
- **Strong Frameworks:** Python has powerful web frameworks like Flask and Django.
- **Community Support:** A large, active community provides extensive resources.

9.2 Flask: A Micro Web Framework

Flask is a lightweight and flexible web framework for building small to medium-sized web applications.

9.2.1 Installing Flask

To install Flask, use the following command:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
PS C:\Users\admin\OneDrive - MSFT\Mastering Python Programming\Code> pip install flask
```

9.2.2 Creating a Basic Flask Application

Example Code:

```python
from flask import Flask

app = Flask(__name__)

@app.route('/')
def home():
    return "Hello, World!"

if __name__ == '__main__':
    app.run(debug=True)
```

Output:

Hello, World!

Explanation:

- This code creates a basic Flask application that displays "Hello, World!" on the homepage.

9.2.3 Routing in Flask

Routing maps URLs to functions.

Example Code:

```python
from flask import Flask

app = Flask(__name__)

@app.route('/')
def home():
    return "Hello, World!"

@app.route('/about')
def about():
    return "This is the about page."

if __name__ == '__main__':
    app.run(debug=True)
```

Explanation:

- The `/about` route maps to the `about` function, which returns a string.

9.3 Django: A Full-Stack Web Framework

Django is a high-level web framework that encourages rapid development and clean, pragmatic design.

9.3.1 Installing Django

To install Django, use the following command:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\Users\admin\OneDrive - MSFT\Mastering Python Programming\Code> pip install django
```

9.3.2 Creating a Django Project

Command Line:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\Users\admin\OneDrive - MSFT\Mastering Python Programming\Code> django-admin startproject myproject
>> cd myproject
>> python manage.py runserver
>>
```

Explanation:

- This sequence of commands creates a new Django project and starts the development server.

9.3.3 Django Models and Views

Django's models define the structure of your database, while views handle the logic of what data is presented to the user.

Example Code:

9_3_3.py

```
9_3_3.py > Article
1  from django.db import models
2
3  class Article(models.Model):
4      title = models.CharField(max_length=100)
5      content = models.TextField()
```

Explanation:

- This code defines an `Article` model with `title` and `content` fields.

9.3.4 Templating in Django

Django uses templates to separate HTML design from Python code.

Example Template:

```html
<h1>{{ article.title }}</h1>
<p>{{ article.content }}</p>
```

Explanation:

- This template displays the title and content of an article using Django's template language.

9.4 Building a Simple Web Application

Combining Flask or Django, you can build a simple web application from scratch.

9.4.1 Defining the Application Structure

Organize your files and folders to keep your project maintainable.

Structure Example:

```
myproject/
    app/
        templates/
        static/
        __init__.py
        views.py
        models.py
```

Explanation:

- The `templates/` folder contains HTML files, and `static/` holds CSS and JavaScript files.

9.4.2 Creating the Application

Example Code:

- Define routes, templates, and models in your Flask or Django application to create a fully functional web application.

We Will Develop An Simple Website With Flask, Django And Python in Our Project Section

9.5 Connecting to a Database

Web applications often need to store data persistently, which is done through databases.

9.5.1 Setting Up a Database in Flask

Use SQLAlchemy for database interactions in Flask.

Example Code:

9_5_1.py

```python
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
app = Flask(__name__)

app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///data.db'
db = SQLAlchemy(app)
```

Explanation:

- This code configures Flask to use a SQLite database.

9.5.2 Setting Up a Database in Django

Django comes with built-in support for several databases, including SQLite.

Example Code:

- Configure your database in the `settings.py` file:

9_5_2.py

```python
import os

BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': BASE_DIR / "db.sqlite3",
    }
}
#Explanation:
#This code sets up SQLite as the database for your Django project.

```

10. Introduction to Data Science

Data science is the process of extracting valuable insights from data using various techniques and tools.

10.1 What is Data Science?

Data science combines statistics, computer science, and domain knowledge to analyze data and derive insights.

10.1.1 The Data Science Process

- **Data Collection:** Gathering raw data from various sources.
- **Data Cleaning:** Removing inconsistencies and preparing the data for analysis.
- **Data Analysis:** Applying statistical techniques to extract insights.
- **Data Visualization:** Representing data graphically to make it easier to understand.

10.2 Python Libraries for Data Science: NumPy, Pandas, Matplotlib

Python offers several powerful libraries for data science.

10.2.1 NumPy

NumPy is the foundational library for numerical computing in Python.

Example Code:

10_2_1.py

```python
import numpy as np

arr = np.array([1, 2, 3])
print(arr.mean())
```

Explanation:

- This code creates a NumPy array and calculates its mean.

10.2.2 Pandas

Pandas is used for data manipulation and analysis.

Example Code:

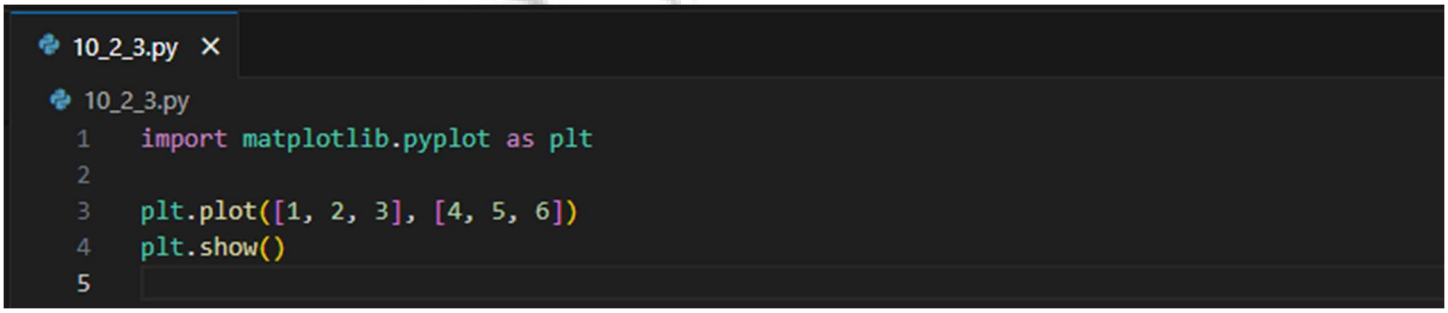
Explanation:

- This code creates a DataFrame and displays the first few rows.

10.2.3 Matplotlib

Matplotlib is a plotting library for creating static, animated, and interactive visualizations.

Example Code:



Explanation:

- This code creates a simple line plot and displays it.

10.3 Data Cleaning and Preprocessing

Before analyzing data, you must clean and preprocess it.

10.3.1 Handling Missing Data

Missing data can be handled by removing or imputing values.

Example Code:

10_3_1.py

```python
import pandas as pd
df = pd.DataFrame({
    "Name": ["Suryanhsk", "Avanish"],
    "Age": [18, 17]
})
df.dropna(inplace=True)  # Removes missing data
df.fillna(df.mean(), inplace=True)  # Imputes missing data with the mean
```

Explanation:

- This code shows how to handle missing data in a DataFrame.

10.3.2 Data Transformation

Transforming data is essential for making it suitable for analysis.

Example Code:

10_3_2.py

```python
import pandas as pd
df = pd.DataFrame({
    "Name": ["Suryanhsk", "Avanish"],
    "Age": [18, 17]
})
df['Age'] = df['Age'] * 2  # Example of scaling data
```

Explanation:

- This code doubles the values in the `Age` column.

10.4 Data Visualization

Data visualization helps communicate insights effectively.

10.4.1 Creating Basic Plots

You can create basic plots like histograms, bar charts, and scatter plots using Matplotlib.

Example Code:

10_4_1.py

```python
import pandas as pd
import matplotlib.pyplot as plt

df = pd.DataFrame({
    "Name": ["Suryanhsk", "Avanish"],
    "Age": [18, 17]
})
df['Age'].hist()
plt.show()
```

Explanation:

- This code creates a histogram of the `Age` column.

10.4.2 Advanced Visualization Techniques

Use Seaborn for more complex visualizations.

Example Code:

10_4_2.py

```python
import seaborn as sns
import pandas as pd
import matplotlib.pyplot as plt

df = pd.DataFrame({
    "Name": ["Suryanhsk", "Avanish"],
    "Age": [18, 17]
})

sns.boxplot(x="Age", data=df)
plt.show()
```

Explanation:

- This code creates a boxplot to visualize the distribution of ages.

10.5 Basic Statistical Analysis

Statistics are essential for understanding data and making inferences.

10.5.1 Descriptive Statistics

Calculate basic statistics like mean, median, and mode.

Example Code:

10_5_1.py

```python
import pandas as pd
df = pd.DataFrame({
    "Name": ["Suryanhsk", "Avanish"],
    "Age": [18, 17]
})
print(df['Age'].mean())  # Mean
print(df['Age'].median())  # Median
print(df['Age'].mode())  # Mode
```

Explanation:

- This code calculates the mean, median, and mode of the `Age` column.

10.5.2 Correlation Analysis

Correlation measures the relationship between two variables.

Example Code:

Explanation:

- This code calculates the correlation matrix for the DataFrame.

10_5_2.py

```python
import pandas as pd

df = pd.DataFrame({
    "Age": [18, 17]
})

print(df.corr())
```

11. Machine Learning with Python

Machine learning (ML) is a subset of artificial intelligence (AI) that enables systems to learn from data and make predictions.

11.1 Introduction to Machine Learning

11.1.1 What is Machine Learning?

Machine learning involves training models on data to make predictions or decisions without being explicitly programmed.

11.1.2 Types of Machine Learning

- **Supervised Learning:** The model is trained on labeled data.
- **Unsupervised Learning:** The model identifies patterns in unlabeled data.
- **Reinforcement Learning:** The model learns through rewards and punishments.

11.2 Supervised vs. Unsupervised Learning

11.2.1 Supervised Learning

In supervised learning, the model learns from labeled data.

Example:

- Predicting house prices based on historical data.

11.2.2 Unsupervised Learning

In unsupervised learning, the model finds patterns in data without explicit labels.

Example:

- Clustering customers based on purchasing behavior.

11.3 Scikit-Learn Library

Scikit-Learn is a powerful library for machine learning in Python.

11.3.1 Installing Scikit-Learn

Install Scikit-Learn using pip:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\Users\admin\OneDrive - MSFT\Mastering Python Programming\Code> pip install scikit-learn
```

11.3.2 Basic Usage of Scikit-Learn

Scikit-Learn provides tools for data preprocessing, model training, and evaluation.

#pip install scikit-learn

Example Code:

11_3_2.py

```python
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score

# Load data
iris = load_iris()
X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target, test_size=0.2)

# Train model
model = DecisionTreeClassifier()
model.fit(X_train, y_train)

# Make predictions
y_pred = model.predict(X_test)

# Evaluate model
print(accuracy_score(y_test, y_pred))
```

Output:

```
0.9666666666666667
```

Explanation:

- This code demonstrates the basic workflow of loading data, splitting it into training and test sets, training a model, and evaluating its accuracy.

11.4 Building Your First Machine Learning Model

11.4.1 The Machine Learning Workflow

1. **Data Collection:** Gather data from various sources.
2. **Data Preprocessing:** Clean and transform the data.
3. **Model Training:** Train a machine learning model on the processed data.
4. **Model Evaluation:** Test the model's performance on unseen data.
5. **Model Deployment:** Use the model in a real-world application.

11.5 Evaluating Model Performance

11.5.1 Confusion Matrix

A confusion matrix is a table used to describe the performance of a classification model.

Example Code:

11_5_1.py

```python
from sklearn.metrics import confusion_matrix
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score

# Load data
iris = load_iris()
X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target, test_size=0.2)

# Train model
model = DecisionTreeClassifier()
model.fit(X_train, y_train)

# Make predictions
y_pred = model.predict(X_test)

print(confusion_matrix(y_test, y_pred))
```

Output:

```
[[12  0  0]
 [ 0  8  2]
 [ 0  0  8]]
```

Explanation:

- This code generates a confusion matrix to evaluate model performance.

11.5.2 Accuracy, Precision, Recall, and F1-Score

Evaluate the model using various metrics.

Example Code:

11_5_2.py

```python
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score

# Load data
iris = load_iris()
X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target, test_size=0.2)

# Train model
model = DecisionTreeClassifier()
model.fit(X_train, y_train)

# Make predictions
y_pred = model.predict(X_test)


print(f"Accuracy: {accuracy_score(y_test, y_pred)}")
print(f"Precision: {precision_score(y_test, y_pred, average='macro')}")
print(f"Recall: {recall_score(y_test, y_pred, average='macro')}")
print(f"F1-Score: {f1_score(y_test, y_pred, average='macro')}")
```

Output:

```
Accuracy: 0.9666666666666667
Precision: 0.9666666666666667
Recall: 0.9666666666666667
F1-Score: 0.9649122807017543
```

Explanation:

- This code calculates accuracy, precision, recall, and F1-score for the model.

Chapter 12: Artificial Intelligence with Python

12.1 Introduction to Artificial Intelligence

Artificial Intelligence (AI) is the simulation of human intelligence in machines that are programmed to think, learn, and adapt. Python has become a leading language in AI due to its simplicity, a vast number of libraries, and active community support.

12.1.1 Key Areas of AI:

- **Machine Learning:** Enabling computers to learn from data.
- **Natural Language Processing (NLP):** Allowing machines to understand and respond to human language.
- **Computer Vision:** Enabling machines to interpret and understand visual data.
- **Robotics:** Creating intelligent robots that can interact with their environment.

12.2 Natural Language Processing (NLP)

NLP is a field of AI that focuses on the interaction between computers and human languages. Python offers powerful libraries for NLP, including NLTK and spaCy.

12.2.1 Installing NLTK:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\Users\admin\OneDrive - MSFT\Mastering Python Programming\Code> pip install nltk
```

12.2.2 Basic NLP Tasks:

- **Tokenization:** Breaking text into words or sentences.
- **Stop Words Removal:** Removing common words that don't contribute much meaning.
- **Stemming and Lemmatization:** Reducing words to their base form.

Example Code:

12_2_2.py

```python
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize

nltk.download('punkt')
nltk.download('stopwords')
nltk.download('punkt_tab')  # Add this line

text = "Natural language processing is a fascinating field of AI."
tokens = word_tokenize(text)
tokens = [word for word in tokens if word.lower() not in stopwords.words('english')]

print(tokens)
```

Output:

```
[nltk_data] Downloading package punkt to
[nltk_data]     C:\Users\admin\AppData\Roaming\nltk_data...
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package stopwords to
[nltk_data]     C:\Users\admin\AppData\Roaming\nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
[nltk_data] Downloading package punkt_tab to
[nltk_data]     C:\Users\admin\AppData\Roaming\nltk_data...
[nltk_data]   Unzipping tokenizers\punkt_tab.zip.
['Natural', 'language', 'processing', 'fascinating', 'field', 'AI', '.']
```

Explanation:

- This code demonstrates how to tokenize a sentence and remove stopwords.

12.3 Deep Learning with TensorFlow and Keras

Deep Learning is a subset of AI that mimics the workings of the human brain in processing data and creating patterns for decision making.

12.3.1 Installing TensorFlow and Keras:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
PS C:\Users\admin\OneDrive - MSFT\Mastering Python Programming\Code> pip install tensorflow
```

12.3.2 Building a Neural Network with Keras:

Keras, integrated with TensorFlow, provides a high-level API to build and train deep learning models.

Example Code:

12_3_2.py 2

12_3_2.py > ...

```python
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# Building a simple neural network
model = Sequential()
model.add(Dense(64, activation='relu', input_shape=(10,)))
model.add(Dense(64, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

print(model.summary())
```

Output:

```
Model: "sequential"
```

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 64)	704
dense_1 (Dense)	(None, 64)	4,160
dense_2 (Dense)	(None, 1)	65

```
Total params: 4,929 (19.25 KB)
Trainable params: 4,929 (19.25 KB)
Non-trainable params: 0 (0.00 B)
None
```

What is the purpose of each layer in this neural network?

```
First Dense Layer:
Purpose: This layer has 64 neurons and uses the ReLU activation function.
Role: It serves as the first hidden layer, transforming the input data (with 10 features)
into a higher-dimensional space. The ReLU activation helps introduce non-linearity, allowing
the network to learn more complex patterns.
```

```
Second Dense Layer:
Purpose: This layer also has 64 neurons and uses the ReLU activation function.
Role: It acts as an additional hidden layer, further transforming the data. Adding more
layers allows the network to learn more abstract features and improve its ability to
generalize from the training data.


Output Layer:
Purpose: This layer has 1 neuron and uses the sigmoid activation function.
Role: It serves as the output layer, producing a single value between 0 and 1. The sigmoid
activation is suitable for binary classification tasks, as it outputs a probability-like
value indicating the likelihood of the input belonging to a particular class.
Each layer in the network contributes to transforming the input data step-by-step, enabling
the model to learn and make predictions.
```

Explanation:

- This code sets up a simple feedforward neural network with Keras, designed for binary classification.

12.4 Creating AI Models

12.4.1 Building a Sentiment Analysis Model:

Sentiment analysis is a popular AI application used to determine the sentiment behind a text (e.g., positive, negative, neutral).

1. **Imports necessary modules** from TensorFlow and Keras.
2. **Example data**: Defines some example sentences and their corresponding labels.
3. **Tokenization**:
   - Initializes a `Tokenizer` with a vocabulary size of 1000 words.
   - Fits the tokenizer on the example sentences.
   - Converts the sentences into sequences of integers.
   - Pads the sequences to ensure they all have the same length (5 in this case).
4. **Building the model**:
   - Adds an `Embedding` layer to convert word indices into dense vectors of fixed size (64).
   - Adds an `LSTM` layer with 64 units to capture sequential dependencies.
   - Adds a `Dense` layer with a sigmoid activation function for binary classification.
5. **Compiles the model** with the Adam optimizer, binary cross-entropy loss, and accuracy as a metric.
6. **Prints the model summary** to show the architecture.

Example Code:

12_4_1.py

```
.vscode > 12_4_1.py > ...
from tensorflow.keras.preprocessing.text import Tokenizer # type: ignore
from tensorflow.keras.preprocessing.sequence import pad_sequences # type: ignore
from tensorflow.keras.models import Sequential # type: ignore
from tensorflow.keras.layers import Embedding, LSTM, Dense # type: ignore

# Example data
sentences = ["I love this!", "I hate this!", "This is amazing!", "This is terrible!"]
labels = [1, 0, 1, 0]

# Tokenization
tokenizer = Tokenizer(num_words=1000)
tokenizer.fit_on_texts(sentences)
sequences = tokenizer.texts_to_sequences(sentences)
data = pad_sequences(sequences, maxlen=5)

# Building the model
model = Sequential()
model.add(Embedding(input_dim=1000, output_dim=64, input_length=5))
model.add(LSTM(64))
model.add(Dense(1, activation='sigmoid'))

model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

print(model.summary())
```

Output:

```
Model: "sequential"
```

Layer (type)	Output Shape	Param #
embedding (Embedding)	?	0 (unbuilt)
lstm (LSTM)	?	0 (unbuilt)
dense (Dense)	?	0 (unbuilt)

```
Total params: 0 (0.00 B)
Trainable params: 0 (0.00 B)
Non-trainable params: 0 (0.00 B)
None
```

Explanation:

- This code demonstrates how to build a simple sentiment analysis model using LSTM (Long Short-Term Memory) networks.

12.5 Implementing AI in Python Projects

Integrating AI into real-world applications involves using AI models to solve specific problems, such as image recognition, language translation, or recommendation systems.

12.5.1 Real-World Example: Chatbot

A chatbot can be created using NLP techniques and integrated with an AI model to provide meaningful responses.

Example Code:

12_5_1.py

```python
from chatterbot import ChatBot # type: ignore
from chatterbot.trainers import ChatterBotCorpusTrainer # type: ignore

# Creating a chatbot instance
chatbot = ChatBot('AI Assistant')
trainer = ChatterBotCorpusTrainer(chatbot)

# Training with the English corpus
trainer.train("chatterbot.corpus.english")

# Get a response to an input statement
response = chatbot.get_response("Hello, how are you today?")
print(response)
```

Explanation:

- This code shows how to create a simple chatbot using the `ChatterBot` library.

Chapter 13: Python for Automation

13.1 Automating Tasks with Python

Python can automate repetitive tasks, freeing up time for more complex work. Automation can range from simple tasks like renaming files to complex ones like managing entire workflows.

13.1.1 Renaming Files in a Directory:

Example Code:

13_1_1.py

```python
import os

def rename_files(directory, prefix):
    for count, filename in enumerate(os.listdir(directory)):
        dst = f"{prefix}_{str(count)}.txt"
        src = f"{directory}/{filename}"
        dst = f"{directory}/{dst}"

        os.rename(src, dst)

rename_files('/path/to/directory', 'file')
```

Explanation:

- This script renames all files in a directory by adding a prefix and a sequential number.

13.2 Working with APIs

APIs (Application Programming Interfaces) allow Python to interact with external services, enabling data exchange between different software systems.

13.2.1 Making API Requests:

Example Code:

```python
import requests

# Making a GET request to a public API
response = requests.get('https://api.example.com/data') #Useyour own Api Data For This
data = response.json()

print(data)
```

Explanation:

- This code makes a GET request to a public API and prints the JSON response.

13.3 Web Scraping with BeautifulSoup

Web scraping involves extracting data from websites. Python's BeautifulSoup library makes it easy to scrape and parse HTML and XML content.

13.3.1 Installing BeautifulSoup:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\Users\admin\OneDrive - MSFT\Mastering Python Programming\Code> pip install beautifulsoup4
```

13.3.2 Scraping a Web Page:

Example Code:

```python
import requests
from bs4 import BeautifulSoup

url = 'https://suryanshsk.netlify.app/'
response = requests.get(url)
soup = BeautifulSoup(response.content, 'html.parser')

# Extracting all headings from the page
headings = soup.find_all('h1')
for heading in headings:
    print(heading.text)
```

Output:

```
Suryanshsk.
```

Explanation:

- This script extracts all `<h1>` headings from a web page.

13.4 Automating Excel with OpenPyXL

Python can automate Excel tasks like creating, reading, and modifying spreadsheets using the OpenPyXL library.

13.4.1 Installing OpenPyXL:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\Users\admin\OneDrive - MSFT\Mastering Python Programming\Code> pip install openpyxl
```

13.4.2 Creating and Writing to an Excel File:

Example Code:

13_4_2.py

```python
import openpyxl

# Creating a new Excel file
workbook = openpyxl.Workbook()
sheet = workbook.active

# Writing data to the file
sheet['A1'] = 'Hello'
sheet['B1'] = 'World'

# Saving the file
workbook.save('example.xlsx')
```

Output:

	A	B	C
1	Hello	World	
2			

Explanation:

- This code creates a new Excel file and writes data into it.

13.5 Automating Emails and Social Media Posts

Python can automate the sending of emails and posting on social media platforms, making it a valuable tool for digital marketing and communication.

13.5.1 Sending an Email with SMTP:

Example Code:

13_5_1.py

```python
import smtplib
from email.mime.text import MIMEText

def send_email(subject, body, to_email):
    from_email = "your_email@example.com"
    password = "your_password"

    # Email content
    msg = MIMEText(body)
    msg['Subject'] = subject
    msg['From'] = from_email
    msg['To'] = to_email

    # Sending email
    server = smtplib.SMTP('smtp-mail.outlook.com', 587)
    server.starttls()
    server.login(from_email, password)
    server.sendmail(from_email, to_email, msg.as_string())
    server.quit()

send_email('Hello', 'This is a test email.', 'recipient@example.com')

```

Explanation:

- This script sends an email using Python's `smtplib` library.

13.5.2 Posting on Twitter with Tweepy:

#For Consumer_key Consumer_secret,Access_token,Access_token_secret

#visit https://developer.x.com/en/docs/authentication/oauth-1-0a/api-key-and-secret

Example Code:

13_5_2.py

```python
import tweepy

# Authenticate to Twitter
auth = tweepy.OAuthHandler('consumer_key', 'consumer_secret')
auth.set_access_token('access_token', 'access_token_secret')

api = tweepy.API(auth)

# Post a tweet
api.update_status('Hello, world! This is an automated tweet.')
```

Explanation:

- This script posts a tweet using the `Tweepy` library, which interacts with the Twitter API.

Chapter 14 : Building Real-World Projects

Project 1: Personal Voice Assistant

Note: Create Each Project In Separate Folder And For Each Project Save every Code Of project In Separate Folder

Example : For Project 1- Every File of Project 1 Save In Separate Folder Give Name Project 1 And for Another Project Create Another Separate Folder

1. Project Overview

- **Objective:** Create a voice assistant capable of performing tasks such as answering questions, opening applications, and retrieving information from the web.
- **Tools & Libraries:** Python, SpeechRecognition, pyttsx3 (text-to-speech), Wikipedia, webbrowser, datetime, os, smtplib.

2. Setting Up the Environment

- Install necessary libraries

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
PS C:\Users\admin\OneDrive - MSFT\Mastering Python Programming\Code> pip install SpeechRecognition pyttsx3 wikipedia
```

3. Building the Voice Recognition Module

- **Code**

speech_recognition.py

Project_1 > speech_recognition.py > recognize_speech

```python
import speech_recognition as sr

def recognize_speech():
    recognizer = sr.Recognizer()
    with sr.Microphone() as source:
        print("Listening...")
        audio = recognizer.listen(source)

    try:
        print("Recognizing...")
        query = recognizer.recognize_google(audio, language='en-in')
        print(f"User said: {query}\n")
    except Exception as e:
        print("Sorry, I could not understand. Could you please say that again?")
        return "None"
    return query.lower()
```

4. Adding Text-to-Speech Capability

- **Code:**

speech_recognition.py | Text-to-Speech.py

Project_1 > Text-to-Speech.py > ...

```python
import pyttsx3

engine = pyttsx3.init()

def speak(text):
    engine.say(text)
    engine.runAndWait()

speak("Hello! How can I assist you today?")

```

5. Implementing Core Functionalities

- **Tasks:** Greeting, fetching information from Wikipedia, opening applications, telling time.
- **Code:**

speech_recognition.py | text_to_speech.py | main_application.py

Project_1 > main_application.py > ...

```python
import datetime
import wikipedia
import webbrowser
import os
from speech_recognition import recognize_speech
from text_to_speech import speak

def greet_user():
    hour = int(datetime.datetime.now().hour)
    if hour >= 0 and hour < 12:
        speak("Good Morning!")
    elif hour >= 12 and hour < 18:
        speak("Good Afternoon!")
    else:
        speak("Good Evening!")
    speak("I am your voice assistant. How can I help you today?")

def fetch_wikipedia(query):
    speak('Searching Wikipedia...')
    results = wikipedia.summary(query, sentences=2)
    speak("According to Wikipedia")
    speak(results)

def open_app(app_name):
    if "notepad" in app_name:
        os.system("notepad")
    elif "browser" in app_name:
```

```
            webbrowser.open("https://google.com")
        else:
            speak("Sorry, I can't open that application.")

    def tell_time():
        strTime = datetime.datetime.now().strftime("%H:%M:%S")
        speak(f"The time is {strTime}")

    # Example usage
    greet_user()
    command = recognize_speech()
    if 'wikipedia' in command:
        command = command.replace("wikipedia", "")
        fetch_wikipedia(command)
    elif 'time' in command:
        tell_time()
    elif 'open' in command:
        open_app(command)
```

Note: Be Careful About File name

6. Running the Complete Voice Assistant

- **Code Integration:** Combine all functionalities into a continuous loop to run the assistant.
- **Output:** The assistant listens for commands, processes them, and provides responses or performs actions.

For Advance Version Of this Project Visit My GitHub Repositories:

https://github.com/suryanshsk/Python-Voice-Assistant-Suryanshsk

Project 2: E-Commerce Recommendation System

Step 1: Set Up the Project Directory

Create a project folder called `ecommerce_recommendation_system`. Inside this folder, create the following files:

- `preprocessing.py`: For data preprocessing.
- `train_model.py`: For model training.
- `recommend.py`: For making recommendations.

Step 2: Install Required Libraries

#pip install pandas, numpy, scikit-learn

Step 3: Data Collection

You can either use a public dataset or create a synthetic dataset. Save the dataset as `ratings.csv` inside the `data/` folder.

Example of `ratings.csv`:

	A	B	C
1	user_id	item_id	rating
2	1	101	5
3	2	102	3
4	5	101	3
5	3	103	4
6	3	104	2
7	1	104	5
8	4	105	2

Step 4: Data Preprocessing (`preprocessing.py`)

Create a script to load and preprocess the data.

ratings.csv | preprocessing.py

Project_2 > preprocessing.py > ...

```python
import pandas as pd

def load_data(file_path):
    """Load the ratings dataset."""
    return pd.read_csv(file_path)

def preprocess_data(ratings):
    """Prepare the data for the recommendation system."""
    # Create a pivot table with users as rows and items as columns
    ratings_matrix = ratings.pivot(index='user_id', columns='item_id', values='rating').fillna(0)
    return ratings_matrix

if __name__ == "__main__":
    ratings = load_data("ratings.csv")
    ratings_matrix = preprocess_data(ratings)
    print(ratings_matrix)

```

Step 5: Model Training (`train_model.py`)

Use collaborative filtering to create the recommendation model.

ratings.csv | preprocessing.py | train_model.py

Project_2 > train_model.py > ...

```python
from sklearn.metrics.pairwise import cosine_similarity
import numpy as np
import pandas as pd
from preprocessing import load_data, preprocess_data

def train_model(ratings_matrix):
    """Train the recommendation model using cosine similarity."""
    user_similarity = cosine_similarity(ratings_matrix)
    np.fill_diagonal(user_similarity, 0)
    return user_similarity

if __name__ == "__main__":
    ratings = load_data("ratings.csv")
    ratings_matrix = preprocess_data(ratings)
    user_similarity = train_model(ratings_matrix)

    # Save the user similarity matrix
    np.save("user_similarity.npy", user_similarity)
    print("Model trained and user similarity matrix saved.")
```

Step 6: Making Recommendations (`recommend.py`)

Use the trained model to make recommendations.

ratings.csv | preprocessing.py | train_model.py | recommend.py

Project_2 > recommend.py > ...

```python
import numpy as np
import pandas as pd
from preprocessing import load_data, preprocess_data

def recommend_items(user_id, ratings_matrix, user_similarity, top_n=5):
    """Recommend items to a user based on similarity to other users."""
    user_idx = user_id - 1
    similar_users = np.argsort(-user_similarity[user_idx])[:top_n]

    similar_users_ratings = ratings_matrix.iloc[similar_users]
    recommended_items = similar_users_ratings.mean(axis=0).sort_values(ascending=False).index

    return recommended_items[:top_n]

if __name__ == "__main__":
    ratings = load_data("ratings.csv")
    ratings_matrix = preprocess_data(ratings)

    # Load the trained user similarity matrix
    user_similarity = np.load("user_similarity.npy")

    user_id = int(input("Enter user ID for recommendations: "))
    recommendations = recommend_items(user_id, ratings_matrix, user_similarity, top_n=5)

    print(f"Recommended items for user {user_id}: {list(recommendations)}")

```

Step 7: Run the System

1. First, preprocess the data:

   ```
   python preprocessing.py
   ```

2. Train the model:

   ```
   python train_model.py
   ```

3. Make recommendations:

   ```
   python recommend.py
   ```

Project 3: Automated Stock Trading Bot

1. Project Overview

- **Objective:** Create a bot that automates stock trading based on predefined strategies.
- **Tools & Libraries:** Python, Alpaca API, Pandas, NumPy.

2. Setting Up the Environment

- Install necessary libraries:

 # pip install alpaca-trade-api pandas numpy

3. Connecting to the Alpaca API

- **Code:**

alpca_api.py | main_application.py

project_3 > alpca_api.py > ...

```python
import alpaca_trade_api as tradeapi # type: ignore

API_KEY = 'your_api_key'
API_SECRET = 'your_api_secret'
BASE_URL = 'https://paper-api.alpaca.markets'

api = tradeapi.REST(API_KEY, API_SECRET, BASE_URL, api_version='v2')
account = api.get_account()
print(account)
```

4. Implementing a Simple Trading Strategy

- **Code:**

alpca_api.py | main_application.py

project_3 > main_application.py > moving_average_strategy

```python
from flask import app


def moving_average_strategy(symbol, short_window=40, long_window=100):
    barset = app.get_barset(symbol, 'day', limit=long_window)
    bars = barset[symbol]

    short_ma = sum([bar.c for bar in bars[-short_window:]]) / short_window
    long_ma = sum([bar.c for bar in bars]) / long_window

    if short_ma > long_ma:
        print(f"Buy signal for {symbol}")
        app.submit_order(
            symbol=symbol,
            qty=1,
            side='buy',
            type='market',
            time_in_force='gtc'
        )
    elif short_ma < long_ma:
        print(f"Sell signal for {symbol}")
        app.submit_order(
            symbol=symbol,
            qty=1,
            side='sell',
            type='market',
            time_in_force='gtc'
        )

# Example usage
moving_average_strategy('AAPL')
```

5. Running the Stock Trading Bot

- **Output:** The bot executes trades based on the strategy and logs the transactions.

For All Others Mega Projects Visit My GitHub Repositories:

https://github.com/suryanshsk

15.Preparing for Placement Interviews

Preparing for placement interviews can be a daunting task, especially when it comes to technical interviews that require strong coding skills. This section of your book will guide readers through the essential topics they need to master to excel in Python coding interviews. From understanding common coding questions to mastering data structures and algorithms, this chapter will equip your readers with the knowledge and confidence needed to crack coding interviews successfully.

1. Python Coding Questions for Interviews

Overview: This subsection covers typical Python coding questions that candidates are likely to encounter during technical interviews. These questions range from basic syntax and operations to more complex problems that test a candidate's understanding of Python's core concepts.

Topics Covered:

- **Basic Syntax Questions:**
  - Examples of simple print statements, variable assignments, and basic operations.
  - **Example:**

1.py

placement > 1.py > ...

```python
# Question: What will be the output of the following code?
a = 5
b = 10
print(a + b)
# Answer: 15
```

- **String Manipulation:**
  - Operations such as reversing a string, checking for palindromes, and finding substrings.
  - **Example:**

2.py

placement > 2.py > ...

```python
# Question: Write a function to check if a string is a palindrome.
def is_palindrome(s):
    return s == s[::-1]

```

- **List and Array Operations:**
  - Common list manipulations, including sorting, searching, and removing duplicates.
  - **Example:**

3.py

```
placement > 3.py > ...
1   # Question: Write a function to remove duplicates from a list.
2   def remove_duplicates(lst):
3       return list(set(lst))
4
```

- **Dictionary and Set Operations:**
  - Working with dictionaries and sets, including common use cases like counting occurrences and filtering data.
  - s**Example:**

4.py

```
placement > 4.py > ...
1   # Question: Write a function to count the occurrences of each word in a sentence.
2   def word_count(sentence):
3       words = sentence.split()
4       return {word: words.count(word) for word in set(words)}
5
```

- **Basic Algorithms:**
  - Simple algorithmic problems, such as finding the maximum or minimum in a list, or basic search algorithms.
  - **Example:**

5.py

```
placement > 5.py > ...
1   # Question: Write a function to find the maximum value in a list.
2   def find_max(lst):
3       return max(lst)
4
```

2. Solving Problems with Python: A Step-by-Step Guide

Overview: This section provides a step-by-step approach to solving coding problems using Python. It emphasizes the importance of understanding the problem, planning the solution, writing clean and efficient code, and testing thoroughly.

Topics Covered:

- **Understanding the Problem:**
  - Techniques to carefully read and understand what the problem is asking.
  - Breaking down the problem into smaller, manageable parts.
  - **Example:**

    ```
    Problem: Given a list of integers, return the indices of the two
    numbers that add up to a specific target.
    ```

- **Planning the Solution:**
  - How to brainstorm possible approaches and select the most efficient one.
  - Writing pseudocode before diving into the actual coding.
  - **Example:**

    ```
    Pseudocode:
    - Create a dictionary to store the difference between the target and
    each element.
    - Loop through the list to check if the current element exists in the
    dictionary.
    - If it exists, return the index.
    ```

- **Writing the Code:**
  - Implementing the solution in Python using clear, readable code.
  - Best practices for naming variables, using functions, and structuring the code.
  - **Example:**

6.py

placement > 6.py > ...

```python
def two_sum(nums, target):
    seen = {}
    for i, num in enumerate(nums):
        remaining = target - num
        if remaining in seen:
            return [seen[remaining], i]
        seen[num] = i

```

- **Testing the Solution:**
  - Importance of testing with various cases, including edge cases.
  - How to write test cases and use Python's `unittest` or `pytest` frameworks.

  - **Example:**

7.py

placement > 7.py > ...

```python
import unittest

def two_sum(nums, target):
    for i in range(len(nums)):
        for j in range(i + 1, len(nums)):
            if nums[i] + nums[j] == target:
                return [i, j]

class TestTwoSum(unittest.TestCase):
    def test_two_sum(self):
        self.assertEqual(two_sum([2, 7, 11, 15], 9), [0, 1])
        self.assertEqual(two_sum([3, 2, 4], 6), [1, 2])
        self.assertEqual(two_sum([3, 3], 6), [0, 1])

if __name__ == '__main__':
    unittest.main()
```

3. Data Structures and Algorithms in Python

Overview: This section dives into the most important data structures and algorithms that every Python developer should know. Understanding these concepts is crucial for solving complex problems efficiently.

Topics Covered:

- **Data Structures:**
  - **Arrays and Lists:**
    - How to use Python's list data structure for array-like operations.
    - **Example:**

8.py

placement > 8.py > ...

```python
# Inserting an element into a list
my_list = [1, 2, 3]
my_list.append(4)
```

- **Stacks and Queues:**
  - Implementing stack and queue operations using lists and `collections.deque`.
  - **Example:**

9.py

placement > 9.py > ...

```python
from collections import deque
# Stack
stack = []
stack.append(1)
stack.pop()

# Queue
queue = deque([1, 2, 3])
queue.append(4)
queue.popleft()
```

- **Dictionaries and Hashmaps:**
  - Efficiently storing and retrieving key-value pairs.
  - **Example:**

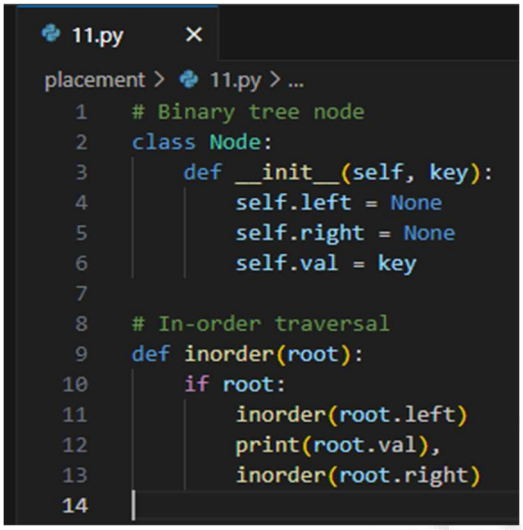
10.py

placement > 10.py > ...

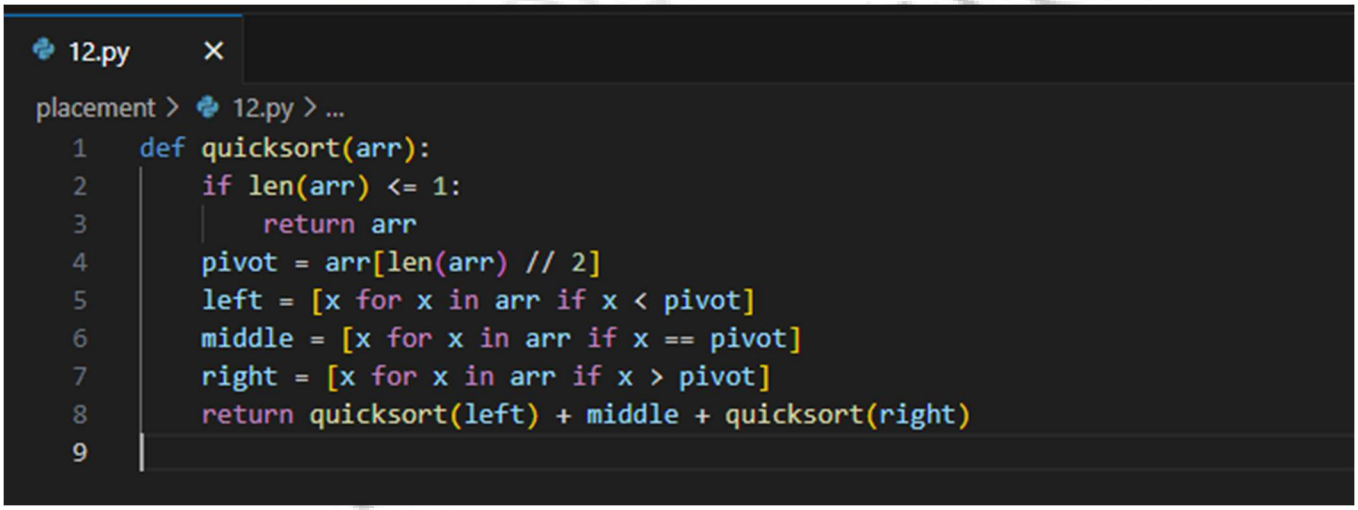
```python
# Using a dictionary to store word frequencies
word_freq = {}
for word in ["apple", "banana", "apple"]:
    word_freq[word] = word_freq.get(word, 0) + 1

```

- **Trees and Graphs:**
  - Basic concepts and Python implementations of binary trees, binary search trees, and graph traversal algorithms.

- **Example:**



- **Algorithms:**
  - **Sorting Algorithms:**
    - Implementations of bubble sort, merge sort, and quicksort.
    - **Example:**



- **Search Algorithms:**
  - Linear search and binary search implementations.
  - **Example:**

13.py

placement > 13.py > ...

```python
def binary_search(arr, target):
    low, high = 0, len(arr) - 1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
    return -1

```

- **Dynamic Programming:**
  - Solving problems using dynamic programming with examples like Fibonacci sequence, knapsack problem, etc.

  - **Example:**

14.py

placement > 14.py > ...

```python
def fibonacci(n, memo={}):
    if n in memo:
        return memo[n]
    if n <= 2:
        return 1
    memo[n] = fibonacci(n - 1, memo) + fibonacci(n - 2, memo)
    return memo[n]

```

4. Tips for Cracking Coding Interviews

Overview: This section provides practical advice and strategies for approaching coding interviews. It includes tips on problem-solving techniques, time management, communication, and handling difficult questions.

Topics Covered:

- **Problem-Solving Techniques:**
  - Approaching problems systematically using methods like divide and conquer, and brute force vs. optimized solutions.
  - **Example:**

```
Tip: Always start with the brute force solution, then gradually optimize
it. Interviewers often appreciate seeing your thought process.
```

- **Time Management:**
  - Strategies for managing time effectively during coding interviews, such as prioritizing problems and knowing when to move on from a stuck problem.
  - **Example:**

```
Tip: If you are stuck on a problem for more than 10 minutes, it's often
better to move on to the next problem and return later.
```

- **Communication Skills:**
  - The importance of clearly explaining your thought process, asking clarifying questions, and discussing your approach with the interviewer.
  - **Example:**

```
Tip: Always talk through your thought process while coding. It shows the
interviewer how you think and approach problems.
```

- **Handling Difficult Questions:**
  - Techniques for dealing with questions that are particularly challenging or beyond your current knowledge.
  - **Example:**

```
Tip: If you don't know the answer, it's okay to admit it, but then try to
reason through the problem based on your existing knowledge.
```

5. Practice Interview Questions

Overview: This final section offers a collection of practice problems and interview questions that readers can use to test their knowledge and prepare for actual coding interviews. Each question is accompanied by hints and solutions to help readers learn from their mistakes.

Topics Covered:

- **Warm-Up Problems:**
  - Simple problems to get started with Python coding.
  - **Example:**

15.py

placement > 15.py > ...

```python
# Question: Write a function to find the sum of all even numbers in a list.
def sum_even_numbers(lst):
    return sum(x for x in lst if x % 2 == 0)

```

- **Intermediate Problems:**
  - Problems that require a deeper understanding of data structures and algorithms.
  - **Example:**

16.py

placement > 16.py > ...

```python
# Question: Implement a function to check if two strings are anagrams.
def are_anagrams(str1, str2):
    return sorted(str1) == sorted(str2)

```

- **Advanced Problems:**
  - Challenging problems that test algorithmic thinking and problem-solving abilities.

Example:

17.py

placement > 17.py > ...

```python
# Question: Write a function to find the longest palindromic substring in a given string.
def longest_palindrome(s):
    if len(s) == 0:
        return ""
    longest = s[0]
    for i in range(len(s)):
        for j in range(i + 1, len(s) + 1):
            if s[i:j] == s[i:j][::-1] and len(s[i:j]) > len(longest):
                longest = s[i:j]
    return longest

```

- **Mock Interviews:**
  - Tips for conducting mock interviews with peers or mentors to simulate the real interview experience.
  - **Example:**

    ```
    Tip: Schedule regular mock interviews with friends or mentors to get
    comfortable with the interview format and receive constructive
    feedback.
    ```

I Suggest you to give HackerRank Python Skills Accelerate Certification Test For More Deeper:

https://www.hackerrank.com/skills-verification/python_basic

16.Conclusion

1. Recap of Key Concepts

Overview: In this section, you'll briefly revisit the main topics and concepts discussed throughout the book. This recap will serve as a quick reference guide, helping readers consolidate their knowledge and ensuring they've grasped the essential points.

Key Concepts Recap:

- **Python Fundamentals:**
  - The basics of Python programming, including syntax, data types, and control structures.
  - **Importance:** These fundamentals form the foundation for all Python projects.
- **Project Development Process:**
  - The step-by-step approach to building Python projects, from ideation to deployment.
  - **Importance:** Understanding this process is crucial for developing scalable and maintainable software.
  - **Data Science ,AI ,ML ,Automation**
- **Real-World Applications:**
  - How Python is used in real-world scenarios, including voice assistants, recommendation systems, image recognition, automated trading bots, and AI chatbots.
  - **Importance:** These examples demonstrate the versatility and power of Python in solving practical problems.
- **Data Structures and Algorithms:**
  - A deep dive into essential data structures like lists, dictionaries, trees, and graphs, as well as algorithms for searching, sorting, and optimization.
  - **Importance:** Mastery of these topics is key to writing efficient and effective Python code, especially in a technical interview setting.
- **Interview Preparation:**
  - Tips and strategies for excelling in Python coding interviews, including problem-solving techniques, communication skills, and handling difficult questions.
  - **Importance:** This knowledge equips readers with the confidence and skills needed to succeed in their job search and career advancement.

2. Further Learning Resources

Overview: This section provides readers with a curated list of resources to continue their learning journey. Whether they're looking to deepen their Python knowledge, explore advanced topics, or stay updated with the latest trends, these resources will be invaluable.

Learning Resources:

- **Documentation and Tutorials:**
  - **Python's Official Documentation (python.org):**
    - The go-to resource for Python's syntax, libraries, and updates.
  - **Real Python (realpython.com):**
    - Offers tutorials, quizzes, and articles on a wide range of Python topics.
- **Communities and Forums:**
  - **Stack Overflow:**
    - A Q&A platform where developers can ask questions and share knowledge.
  - **Reddit's r/learnpython:**
    - A supportive community for Python learners of all levels.
- **Open Source Contributions:**
  - **GitHub:**
    - Contribute to Python projects, collaborate with other developers, and learn from open-source code.
    - You can also visit my Repositories
  - **Python's Developer Community:**
    - Participate in Python Enhancement Proposals (PEPs) and stay involved with Python's development.
  - **Suryanshsk**:
    - Follow Me on Social to media to learn More About Programming

---

3. Final Words of Encouragement

Overview: As the book comes to a close, this section aims to inspire and motivate readers to apply what they've learned, continue exploring, and keep pushing their boundaries.

Encouragement Points:

- **Embrace the Journey:**
  - Learning Python and building projects is a continuous journey. Every line of code written, every error encountered, and every project completed is a step forward. Embrace the learning process, and don't be afraid to make mistakes—they are an essential part of growth.
- **Keep Practicing:**
  - The best way to master Python is through consistent practice. Keep coding, keep experimenting, and keep challenging yourself with new projects. The more you code, the more confident and skilled you'll become.
- **Stay Curious:**
  - The world of programming is vast and constantly evolving. Stay curious and open to learning new things. Whether it's a new Python library, a different programming language, or a novel problem-solving technique, there's always something new to discover.

- **Believe in Yourself:**
  - The skills and knowledge you've gained from this book have equipped you to tackle real-world challenges. Believe in your abilities, and don't shy away from opportunities to apply what you've learned. Your hard work and dedication will pay off.
- **Build and Share:**
  - Use the knowledge you've acquired to build meaningful projects. Share your creations with the world, contribute to open-source projects, or even start your own. By doing so, you'll not only reinforce your learning but also inspire others in the community.
- **Never Stop Learning:**
  - Technology is always advancing, and there's always something new to learn. Make lifelong learning a part of your career. Whether it's through books, courses, or hands-on projects, continue to expand your horizons.

Scan This For Contact Me And for all other my resource

It's a Just Starting Never Stop here Always Try to learn Something New ,Something More Deeper .It's Your Beginning Always Be happy ,Be Positive , Be Calm And Be Funny

“Innovation Starts with a single line of code and a boundless Vision “

-Suryanshsk

Thank You So Much For Your Valuable Time