CODTECH IT SOLUTIONS PVT.LTD

IT SERVICES & IT CONSULTING

8-7-7/2, Plot NO.51, Opp: Naveena School, Hasthinapuram Central, Hyderabad , 500 079. Telangana

Python Programming Material

OUR PARTNERS & CERTIFICATIONS

Google Partner

aws PARTNER

CERTIFIED ISO 9001 CERTIFIED

MSME MICRO, SMALL & MEDIUM ENTERPRISES

सत्यमेव जयते

AICTE All India Council for Technical Education

सत्यमेव जयते

MINISTRY OF CORPORATE AFFAIRS

GOVERNMENT OF INDIA

CODTECH IT SOLUTIONS PVT.LTD
IT SERVICES & IT CONSULTING
8-7-7/2, Plot NO.51, Opp: Naveena School, Hasthinapuram Central, Hyderabad , 500 079. Telangana

Zero to Mastery: Python Programming

Chapter 1: Introduction to Python
- What is Python?
- Why Learn Python?
- Installing Python and Setting Up the Environment
- Running Python Scripts and the Interactive Shell
- Writing Your First Python Program

Chapter 2: Python Basics - Syntax and Variables
- Understanding Python Syntax
- Variables and Data Types (int, float, str, bool)
- Comments and Code Readability
- C
- Basic Input and Output

Chapter 3: Operators and Expressions
- Arithmetic Operators
- Comparison Operators
- Logical Operators
- Assignment Operators
- Identity and Membership Operators

Chapter 4: Control Flow - Decision Making
- if, elif, and else Statements
- Nested Conditionals
- Short Circuit Evaluation

Chapter 5: Loops and Iterations
- for Loop
- while Loop
- break, continue, and pass Statements
- Looping with range()

Chapter 6: Functions and Modules

- Defining Functions (def)
- Function Parameters and Return Values
- *args and **kwargs
- Lambda Functions
- Importing and Using Modules

Chapter 7: Data Structures - Lists and Tuples

- Lists: Creation, Indexing, Slicing
- List Methods (append(), remove(), sort(), etc.)
- List Comprehensions
- Tuples: Immutable Sequences

Chapter 8: Dictionaries and Sets

- Dictionary Basics: Keys and Values
- Dictionary Methods (get(), update(), pop(), etc.)
- Set Operations (union, intersection, difference)
- When to Use Dictionaries vs. Sets

Chapter 9: Working with Strings

- String Methods (strip(), split(), join(), etc.)
- String Formatting (f-strings, format())
- String Manipulation and Common Operations

Chapter 10: File Handling

- Reading and Writing Files (open(), read(), write())
- Working with Different File Modes (r, w, a, r+)
- Handling File Exceptions

Chapter 11: Exception Handling

- try, except, finally, and else
- Handling Multiple Exceptions
- Raising Custom Exceptions

Chapter 12: Object-Oriented Programming (OOP)

- Classes and Objects
- Constructor (__init__) and Instance Variables
- Methods and self
- Inheritance and Polymorphism
- Encapsulation and Abstraction

Chapter 13: Working with Modules and Packages

- Built-in Modules (math, random, os, etc.)
- Creating and Importing Custom Modules
- Installing and Using External Libraries (pip)

Chapter 14: Working with Databases (SQLite & MySQL)

- Introduction to Databases
- Connecting to SQLite/MySQL
- CRUD Operations (Create, Read, Update, Delete)
- Using sqlite3 and mysql-connector-python

Chapter 15: Web Scraping with Python

- Introduction to Web Scraping
- Using requests to Fetch Web Pages
- Parsing HTML with BeautifulSoup
- Extracting Data and Saving Results

Chapter 16: Data Visualization with Matplotlib and Seaborn

- Introduction to matplotlib
- Plotting Line, Bar, and Scatter Plots
- Styling Graphs and Adding Labels
- Using seaborn for Advanced Visualization

Chapter 17: Machine Learning Basics

- Introduction to Machine Learning
- Using scikit-learn for ML Models
- Supervised vs. Unsupervised Learning
- Basic Model Training and Evaluation

Chapter 18: Introduction to Web Development with Flask/Django

- Understanding Web Frameworks
- Setting Up a Simple Flask/Django App
- Handling Routes and Templates
- Working with Forms and Databases

Chapter 19: Testing and Debugging in Python

- Writing Unit Tests with unittest
- Debugging with pdb
- Common Python Errors and Fixes

Chapter 20: Advanced Python Concepts

- Decorators and Generators
- Multi-threading and Parallel Processing
- Working with Virtual Environments
- Understanding Python's Memory Management

Chapter 21: Python Project Ideas and Best Practices

- Best Coding Practices and Style Guide (PEP 8)
- Project Ideas for Beginners, Intermediate, and Advanced Levels
- Contributing to Open Source and GitHub
- Python Career Path: What's Next?

This roadmap ensures a step-by-step learning path from absolute beginner to advanced proficiency in Python. Would you like a more detailed breakdown for any specific chapter? 🚀

CHAPTER -1
Introduction to Python

What is Python?

Python is a high-level, interpreted programming language that is widely used for various applications, including web development, data science, artificial intelligence, automation, and more. Created by Guido van Rossum in the late 1980s and officially released in 1991, Python was designed with an emphasis on simplicity and readability, making it an excellent choice for both beginners and experienced developers.

Key Features of Python

1. Simple and Readable Syntax

Python's syntax is clean and easy to understand, resembling natural language. This makes coding more intuitive and reduces the learning curve for beginners. Unlike other programming languages that require complex syntax, Python emphasizes readability, allowing developers to write fewer lines of code to achieve the same functionality.

2. Interpreted Language

Python is an interpreted language, meaning that the code is executed line by line rather than being compiled beforehand. This allows for quick testing and debugging, making development faster and more efficient.

3. Dynamically Typed

Unlike statically typed languages such as C or Java, Python does not require explicit declaration of variable types. The interpreter automatically determines the data type at runtime, making the development process more flexible.

4. Object-Oriented and Procedural

Python supports both object-oriented programming (OOP) and procedural programming. OOP allows developers to use concepts like classes and objects, making the code more reusable and modular. Meanwhile, procedural programming is suitable for writing simple scripts.

5. Cross-Platform Compatibility

Python is platform-independent, meaning that code written on one operating system (Windows, macOS, or Linux) can run on another without modification, as long as Python is installed.

6. Extensive Libraries and Frameworks

Python has a rich ecosystem of libraries and frameworks that extend its functionality. Some popular ones include:

- NumPy, Pandas, and Matplotlib – for data analysis and visualization

- **TensorFlow and PyTorch – for machine learning and deep learning**
- **Django and Flask – for web development**
- **Selenium – for automation and web scraping**
- **Requests – for working with APIs**

7. Large Community and Support

Python has one of the largest programming communities in the world, meaning there is extensive documentation, forums, and online tutorials available. This ensures that developers can easily find solutions to problems and collaborate on open-source projects.

Applications of Python

Python is a versatile language used in many fields:

1. **Web Development – Frameworks like Django and Flask allow developers to build powerful web applications quickly.**
2. **Data Science & Machine Learning – Python is widely used in AI, ML, and data analysis due to its powerful libraries like Scikit-learn and TensorFlow.**
3. **Automation & Scripting – Python can automate repetitive tasks, such as file handling, data entry, and web scraping.**
4. **Cybersecurity – Python is commonly used in penetration testing and ethical hacking.**

- **Game Development – Libraries like Pygame enable the creation of simple games.**
- **Conclusion**
- **Python's simplicity, versatility, and extensive support make it one of the most popular programming languages in the world. Whether you are a beginner learning to code or a professional developing complex applications, Python provides the tools and flexibility to bring ideas to life efficiently.** 🚀

CODTECH IT SOLUTIONS PVT.LTD
IT SERVICES & IT CONSULTING
8-7-7/2, Plot NO.51, Opp: Naveena School, Hasthinapuram Central, Hyderabad , 500 079. Telangana

Why Learn Python?

1. **Easy to Learn & Use – Python has a simple syntax similar to English, making it beginner-friendly and easy to read.**
2. **Versatile & Powerful – It is used in web development, data science, AI, automation, and more, making it highly adaptable.**
3. **Huge Community Support – A large global community ensures extensive documentation, tutorials, and troubleshooting help.**
4. **Rich Libraries & Frameworks – Python offers vast libraries like NumPy, TensorFlow, and Django, speeding up development.**
5. **High Demand & Career Growth – Python developers are in high demand, with lucrative job opportunities in various industries.** 🚀

Installing Python and Setting Up the Environment?

- **Download Python – Visit the official Python website (python.org) and download the latest version for your OS (Windows, macOS, or Linux).**
- **Install Python – Run the installer and check the "Add Python to PATH" option before proceeding with the installation.**

- Verify Installation – Open a terminal or command prompt and type python --version to check if Python is installed correctly.
- Install a Code Editor – Use editors like VS Code, PyCharm, or Jupyter Notebook for writing and running Python code efficiently.
- Use the Python Shell – Open the terminal and type python to enter the interactive shell for quick testing of Python commands.
- Install pip (Package Manager) – Ensure pip is installed by running pip --version, which helps in installing external libraries.
- Set Up a Virtual Environment – Use python -m venv myenv to create a virtual environment and activate it (myenv\Scripts\activate on Windows, source myenv/bin/activate on macOS/Linux).
- Install Required Packages – Use pip install package_name to install necessary Python libraries like NumPy, Pandas, Flask, etc.
- Write Your First Program – Create a Python script (hello.py) with print("Hello, Python!") and run it using python hello.py.
- Explore IDE Features – Learn debugging, syntax highlighting, and extensions in your chosen IDE for a better coding experience. 🚀

Running Python Scripts and the Interactive Shell

1. **Using the Interactive Shell – Open a terminal or command prompt and type python (or python3 on some systems) to enter the Python interactive shell.**
2. **Executing Commands – In the interactive shell, you can type Python commands and see immediate output, making it useful for quick testing.**
3. **Exiting the Shell – Type exit() or press Ctrl + Z (Windows) / Ctrl + D (macOS/Linux) to exit the interactive shell.**
4. **Creating a Python Script – Write Python code in a text file with a .py extension, e.g., script.py.**
5. **Running a Script from Terminal – Use python script.py to execute the script in the command line.**
6. **Running a Script in an IDE – Most IDEs like VS Code, PyCharm, and Jupyter Notebook allow you to run scripts with a single click.**
7. **Using Shebang (Linux/macOS) – Add #!/usr/bin/env python3 at the top of a script to run it directly (chmod +x script.py and ./script.py).**
8. **Passing Command-Line Arguments – Use sys.argv to accept arguments, e.g., python script.py arg1 arg2.**
9. **Running Python in Jupyter Notebook – Install Jupyter using pip install notebook and run jupyter notebook to write and execute Python in an interactive environment.**

- **Debugging Python Code – Use print() for simple debugging or run scripts with python -m pdb script.py for step-by-step debugging.** 🚀

Writing Your First Python Program ?

- **Open a Text Editor or IDE – Use a simple text editor like Notepad or an IDE like VS Code, PyCharm, or Jupyter Notebook.**
- **Create a New Python File – Save a new file with a .py extension, e.g., hello.py.**
- **Write Your First Python Code – Type the following simple**

- **program:**

```
python
print("Hello, Python!")
```

- **Save the File – Ensure the file is saved in a known directory.**
- **Open a Terminal or Command Prompt – Navigate to the directory where your file is saved using the cd command.**
- **Run the Python Script – Execute the program by typing python hello.py and pressing Enter.**

- See the Output – The terminal should display: Hello, Python!.
- Modify the Program – Try adding another line, like:

python :

```
name = input("Enter your name: ")
print("Hello, " + name + "!")
```

CHAPTER -2
Python Basics - Syntax and Variables

Understanding Python Syntax

✓ Indentation-Based Structure – Uses indentation instead of {} to define code blocks.
✓ Dynamic Typing – No need to declare variable types; Python assigns them automatically.
✓ Simple & Readable – Clear syntax, similar to English, with minimal keywords.
✓ No Semicolons Required – Statements don't require ; at the end.
✓ Case-Sensitive – variable and Variable are treated as different identifiers.
✓ Comments for Clarity – Use # for single-line comments and ''' ''' or """ """ for multi-line comments.
✓ Extensive Standard Library – Comes with built-in modules and functions for efficient programming.

Example:

```
# Print a message
print("Hello, Python!")  # Output: Hello, Python!
```

CODTECH IT SOLUTIONS PVT.LTD
IT SERVICES & IT CONSULTING
8-7-7/2, Plot NO.51, Opp: Naveena School, Hasthinapuram Central, Hyderabad , 500 079. Telangana

Variables and Data Types (int, float, str, bool)

1. Variables

- Variables store data and are dynamically typed (no need for explicit declaration).
- Assigned using = (e.g., x = 10).

2. Data Types

Integer (int) – Whole numbers (e.g., x = 10).
Float (float) – Decimal numbers (e.g., y = 3.14).
String (str) – Text enclosed in quotes (e.g., name = "Santhosh").
✔Boolean (bool) – True/False values (e.g., is active = True).

Example Code:

```
x = 10         # Integer
y = 3.14       # Float
name = "Python"  # String
is_active = True # Boolean

print(type(x), type(y), type(name), type(is_active))
```

Comments and Code Readability

Comments and Code Readability in Python

1. Comments in Python

✓ Single-Line Comments – Use # to add a comment.

✓ Multi-Line Comments – Use ''' ''' or """ """ for block comments.

2. Importance of Code Readability

✓ Indentation-Based Structure – Python enforces indentation for clean, readable code.

✓ Descriptive Variable Names – Use meaningful names (total_price instead of tp).

✓ Consistent Formatting – Follow PEP 8 guidelines for better readability.

✓ Limit Line Length – Keep lines under 79 characters for clarity.

Example Code:

```
# This is a single-line comment
'''
This is a multi-line comment.
It explains the purpose of the code below.
'''
def greet(name):
    """Returns a greeting message"""
    return f"Hello, {name}!"  # Using an f-string for readability

print(greet("Santhosh"))
```

CODTECH IT SOLUTIONS PVT.LTD
IT SERVICES & IT CONSULTING
8-7-7/2, Plot NO.51, Opp: Naveena School, Hasthinapuram Central, Hyderabad , 500 079. Telangana

C Programming - Basics

C is a powerful, general-purpose programming language widely used for system programming, embedded systems, and application development.

1. Features of C

1. Fast & Efficient – Low-level memory access and minimal runtime overhead.
2.Portable & Cross-Platform – Can run on various operating systems.
3. Structured Programming – Supports functions and modular code.
4.Rich Library Support – Standard libraries for I/O, math, and more.
5.Foundation for Other Languages – Influenced C++, Java, and Python.

2. Basic Syntax in C

```
#include <stdio.h>  // Include standard input-output library

int main() {
   printf("Hello, C!");  // Print statement
   return 0;  // Exit the program
}
```

3. Variables & Data Types in C

1 Integer (int) – Whole numbers (e.g., int x = 10;)
2 Float (float) – Decimal numbers (e.g., float y = 3.14;)
3 Character (char) – Single character (e.g., char c = 'A';)
4Double (double) – High-precision floating point (e.g., double d = 3.14159;)

4. Type Casting in C

Explicit conversion using type casting:

```
int a = 10;
float b = (float)a;  // Converts int to float
printf("%f", b);  // Output: 10.000000
```

Basic Input and Output

Basic Input and Output in C

In C, input and output operations are performed using the printf() and scanf() functions from the stdio.h library.

1. Output Using printf()

The printf() function displays output on the screen.

Example:

```
#include <stdio.h>

int main() {
   printf("Hello, C Programming!\n");  // \n is used for a
new line
   return 0;
}
```

Output:

```
Hello, C Programming!
```

Chapter 3
Operators and Expressions

Arithmetic Operators

Arithmetic Operators in C
Arithmetic operators in C are used to perform mathematical operations like addition, subtraction, multiplication, etc.

1. List of Arithmetic Operators

Symbol	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Remainder / Modulus
**	Exponent(raise to power)
//	Floor division

Example in Python

```
a = 10
b = 3

print("Addition:", a + b)       # 13
print("Subtraction:", a - b)    # 7
print("Multiplication:", a * b)  # 30
print("Division:", a / b)       # 3.3333
print("Modulus:", a % b)        # 1
print("Exponentiation:", a ** b)  # 1000
print("Floor Division:", a // b)  # 3
```

Comparison Operators

Comparison operators are used to compare two values and return a boolean result (True or False).

Operator	Meaning
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Examples in Python

Checking Equality (==)

```
a = 10
b = 10
print(a == b)  # True
```

2.Checking Inequality (!=)

```
x = 5
y = 3
print(x != y) # True
```

3.Using in Conditional Statements

```
age = 18
if age >= 18:
   print("You can vote.")
else:
   print("You cannot vote.")
```

4.Using in Loops

```
num = 1
while num <= 5:
   print(num)
   num += 1
```

Logical Operators

Logical operators are used to combine multiple conditions and return a Boolean value (True or False)

logical operator	Description	Syntax
and	True if both statements are true	a and b
or	True if one of the statements is true	a or b
not	Reverse the result. If the result is true it will return false and viceversa	not a not b

1.Using and (Both Conditions Must Be True)

```
age = 20
has_voter_id = True

if age >= 18 and has_voter_id:
   print("You are eligible to vote.")  # Output: You are eligible to vote.
else:
   print("You are not eligible to vote.")
```

2.Using or (At Least One Condition Must Be True)

```
rainy = False
umbrella = True

if rainy or umbrella:
    print("You are prepared for rain.")  # Output: You are prepared for rain.
else:
    print("You might get wet!")
```

3.Using not (Reverses the Condition)

```
is_sunny = False

if not is_sunny:
    print("It's not a sunny day.")  # Output: It's not a sunny day.
```

Usage in Loops

```
num = 5
while not (num == 0):  # Loop runs until num becomes 0
    print(num)
    num -= 1
```

Assignment Operators

Assignment operators are used to assign values to variables. They can also perform operations like addition, subtraction, multiplication, etc., while assigning values.

Operator	Example	Equivalent Expression (m=15)	Result
=	y = a+b	y = 10 + 20	30
+=	m +=10	m = m+10	25
-=	m -=10	m = m-10	5
*=	m *=10	m = m*10	150
/=	m /=10	m = m/10	1.5
%=	m %=10	m = m%10	5
**=	m**=2	m = m**2 or $m = m^2$	225
//=	m//=10	m = m//10	1

1. Basic Assignment (=)

```
x = 10
print(x)  # Output: 10
```

2.Addition Assignment (+=)

```
x = 5
x += 3  # x = x + 3
print(x)  # Output: 8
```

3. Subtraction Assignment (-=)

```
x = 10
x -= 4  # x = x - 4
print(x)  # Output: 6
```

4.Multiplication Assignment (*=)

```
x = 7
x *= 2  # x = x * 2
print(x)  # Output: 14
```

5. Division Assignment (/=)

```
x = 9
x /= 3  # x = x / 3
print(x)  # Output: 3.0
```

6.Floor Division Assignment (//=)

```
x = 10
x //= 3  # x = x // 3
print(x)  # Output: 3
```

7.Modulus Assignment (%=)

```
x = 10
x %= 3  # x = x % 3
print(x)  # Output: 1
```

8.Exponentiation Assignment (**=)

```
x = 2
x **= 3  # x = x ** 3
print(x)  # Output: 8
```

Identity and Membership Operators

1.Identity Operators (is, is not)

Identity operators compare the memory addresses of two objects to check if they refer to the same

Operator	Meaning	Example
Is	True if the operands are identical	X is true
Is not	True if the operands are not identical	X is not true

Examples of Identity Operators

```
a = [1, 2, 3]
b = a
c = [1, 2, 3]

print(a is b)   # True (same object)
print(a is c)   # False (different objects with same values)
print(a is not c) # True (because they are different objects)
```

2.Membership Operators (in, not in)

Membership operators check if a value is present in a sequence (like a string, list, tuple, or dictionary).

Operator	Meaning	Example
In	True if value/variable found in the sequence	5 in x
Not in	True if value/variable is not found in sequence	5 not in x

Examples of Membership Operators

```
fruits = ["apple", "banana", "cherry"]
print("banana" in fruits)  # True
print("grape" not in fruits)  # True

text = "Hello, world!"
print("Hello" in text)  # True
print("bye" not in text)  # True
```

Real-World Example

```
user_roles = ["admin", "editor", "viewer"]
if "admin" in user_roles:
    print("Access granted!")
else:
    print("Access denied.")
```

Chapter 4
Control Flow - Decision Making

if, elif, and else Statements

Conditional statements allow us to execute different blocks of code based on certain conditions.

1.if Statement

The if statement executes a block of code only if the condition is True.

```
age = 18
if age >= 18:
   print("You are eligible to vote.")
```

Output:

```
You are eligible to vote.
```

2.if-else Statement

The else block runs if the if condition is False.

```
age = 16
if age >= 18:
   print("You can vote.")
else:
   print("You cannot vote yet.")
```

Output:

```
You cannot vote yet.
```

3.if-elif-else Statement

1.elif (short for "else if") allows checking multiple conditions.
2.The first condition that evaluates to True gets executed, and the rest are ignored.

```
marks = 85

if marks >= 90:
   print("Grade: A")
elif marks >= 80:
   print("Grade: B")
elif marks >= 70:
   print("Grade: C")
else:
   print("Grade: D"
```

Output:

```
Grade: B
```

4.Nested if Statements

You can place an if inside another if to check multiple conditions.

```
age = 20
has_voter_id = True

if age >= 18:
   if has_voter_id:
      print("You can vote.")
   else:
      print("You need a voter ID.")
else:
   print("You are too young to vote.")
```

Output:

You can vote.

5.Using if with Logical Operators (and, or, not)

```
temperature = 30
humidity = 70

if temperature > 25 and humidity > 60:
    print("It's hot and humid!")
```

Output:

It's hot and humid!

6.Short if Statement (Ternary Operator)

Python allows a compact one-line if-else statement.

```
age = 20
status = "Adult" if age >= 18 else "Minor"
print(status)
```

Output:

Adult

Real-World Example

```
username = "admin"
password = "1234"

if username == "admin" and password == "1234":
    print("Login successful!")
else:
    print("Invalid credentials.")
```

Nested Conditionals

A nested conditional means placing one if statement inside another. This allows for checking multiple conditions where one condition depends on another.

Syntax of Nested Conditionals

```
if condition1:
   if condition2:
      # Code block executes if both conditions are True
   else:
      # Code block executes if condition1 is True but condition2 is False
else:
   # Code block executes if condition1 is False
```

Example 1: Checking Age and ID for Voting

```
age = 20
has_voter_id = True

if age >= 18:  # Outer condition
   if has_voter_id:  # Inner condition
      print("You are eligible to vote.")
   else:
      print("You need a voter ID to vote.")
else:
   print("You are too young to vote.")
```

Output:

You are eligible to vote.

Short Circuit Evaluation

What is Short-Circuit Evaluation?

Short-circuit evaluation means that in a logical expression, Python stops evaluating as soon as the final result is determined. This improves performance by avoiding unnecessary computations.

1.Short-Circuit with and Operator

The and operator stops evaluating if the first condition is False because the whole expression will be False regardless of the other conditions.

Example:

```
x = 0
y = 10

if x != 0 and y / x > 2:
    print("Condition met!")
else:
    print("Short-circuited: Division by zero avoided!")
```

Output:

Short-circuited: Division by zero avoided!

Chapter 5
Loops and Iterations

1.for Loop

A for loop is used to iterate over a sequence (list, tuple, string, dictionary, range, etc.) and execute a block of code multiple times.

1.Basic Syntax

```
for variable in sequence:
    # Code to execute in each iteration
```

2.Looping Through a List

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

3.Using range() in a for Loop

The range(start, stop, step) function generates a sequence of numbers.

```
for i in range(1, 6):  # Loops from 1 to 5
    print(i)
```

4.Looping Through a String

```
for letter in "Python":
    print(letter)
```

5.Looping Through a Dictionary

```
student = {"name": "Hari", "age": 20, "grade": "A"}
for key, value in student.items():
    print(key, ":", value)
```

6.Using break in a for Loop

1.break exits the loop when a condition is met.

```
for num in range(1, 10):
    if num == 5:
        break  # Stops at 5
    print(num)
```

7.Using continue in a for Loop

1.continue skips the current iteration and moves to the next.

```
for num in range(1, 6):
    if num == 3:
        continue  # Skips 3
    print(num)
```

8.Using else with for Loop

The else block runs only if the loop completes without break.

```
for num in range(1, 4):
    print(num)
else:
    print("Loop finished!")
```

9.Nested for Loop

```
for i in range(1, 4):
    for j in range(1, 4):
        print(i, j)
```

10.Practical Example: Multiplication Table

```
num = 5
for i in range(1, 11):
    print(f"{num} x {i} = {num * i}")
```

2.while Loop

A while loop repeats a block of code as long as a given condition is True.

1.Basic Syntax

```
while condition:
    # Code to execute
```

2.Example: Print Numbers from 1 to 5

```
i = 1
while i <= 5:
    print(i)
    i += 1  # Increment to avoid infinite loop
```

3.Infinite Loop (Be Careful!)

```
while True:  # Infinite loop
    print("This will run forever!")
```

4.Using break in while Loop

```
i = 1
while i <= 10:
    if i == 5:
        break  # Stops when i is 5
    print(i)
    i += 1
```

5.Using continue in while Loop

```
i = 0
while i < 5:
    i += 1
    if i == 3:
        continue  # Skips printing 3
    print(i)
```

3.break, continue, and pass Statements

These statements control the flow of loops (for and while), allowing you to alter their execution.

1 break Statement

1.Stops the loop immediately when encountered.
2.The program exits the loop and moves to the next code block.

Example: Stop Loop When i Reaches 3

```
for i in range(1, 6):
    if i == 3:
        break  # Exits the loop when i == 3
    print(i)
```

2 continue Statement

- **Skips the current iteration and moves to the next one.**
- **The loop does not stop, it just skips certain values.**
-

Example: Skip Number 3

```
for i in range(1, 6):
    if i == 3:
        continue  # Skips printing 3
    print(i)
```

3 pass Statement

1.Does nothing, acts as a placeholder.
2.Used when you need a syntactically correct block but don't want to execute anything.

Example: Using pass in a Loop

```
for i in range(5):
    pass  # Placeholder, avoids syntax error
```

4. Looping with range()

The range() function is commonly used with loops to generate a sequence of numbers.

1 Basic Syntax of range()

range(start, stop, step)
1.start → (Optional) The starting number (default is 0).
2.stop → The ending number (not included in the range).
3.step → (Optional) The difference between each number (default is 1).

2 Using range() with a for Loop

Example 1: Counting from 0 to 4

```
for i in range(5):  # Starts from 0, goes up to 4
    print(i)
```

3 Specifying start and stop Values

Example 2: Counting from 2 to 6

```
for i in range(2, 7):  # Starts from 2, stops at 6
    print(i)
```

4 Using step to Control the Increment

Example 3: Counting by 2s

```
for i in range(1, 10, 2):  # Start at 1, step by 2, stop before 10
    print(i)
```

5 Looping in Reverse

Example 4: Counting Down from 5 to 1

```
for i in range(5, 0, -1):  # Start at 5, step down by 1, stop before 0
    print(i)
```

6 Using range() with list()

Example 5: Creating a List of Numbers

```
numbers = list(range(1, 6))
print(numbers)
```

7 Looping Over Indexes in a List

Example 6: Using range() with len()

```
fruits = ["Apple", "Banana", "Cherry"]
for i in range(len(fruits)):
    print(f"Index {i}: {fruits[i]}")
```

Chapter 6
Functions and Modules

1.Defining Functions (def)

A function in Python is a reusable block of code that performs a specific task. Functions help in code reusability, modularity, and readability.

1 Defining a Function Using def

Syntax:

```
def function_name(parameters):
    """Optional docstring"""
    # Code block (Function body)
    return value  # Optional
```

2 Creating a Simple Function

Example 1: Function Without Parameters

```
def greet():
    print("Hello, welcome to Python!")
# Calling the function
greet()
```

3 Function with Parameters

Example 2: Function with One Parameter

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

greet("Santhosh")
```

4 Function with Multiple Parameters

Example 3: Add Two Numbers

```
def add_numbers(a, b):
    return a + b

result = add_numbers(5, 3)
print("Sum:", result)
```

5 Function with Default Parameter Values

Example 4: Default Parameter

```
def power(base, exponent=2):  # Default value = 2
    return base ** exponent

print(power(3))  # Uses default exponent=2
print(power(3, 3))  # Overrides default, exponent=3
```

6 Function with Return Statement

Example 5: Returning a Value

```
def square(num):
    return num * num

result = square(4)
print("Square:", result)
```

7 Function with Variable Arguments (*args)

Example 6: Sum of Multiple Numbers

```
def sum_all(*numbers):
    return sum(numbers
print(sum_all(1, 2, 3, 4, 5))
```

2.Function Parameters and Return Values

Functions in Python allow us to pass values (parameters) and return results (return values). These features make functions flexible and reusable.

1 Function Parameters

Parameters (also called arguments) allow functions to receive inputs.
Types of Function Parameters
1.Positional Parameters
2.Default Parameters
3.Arbitrary Arguments (*args)
4.Keyword Arguments (kwargs)**

2 Positional Parameters

Values are passed in order to the function.
Example: Positional Arguments

```
def greet(name, age):
    print(f"Hello {name}, you are {age} years old!")
greet("Santhosh", 21)
```

3 Default Parameters

Default values are used if no argument is passed.
Example: Default Argument

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

greet()       # Uses default value
greet("Chinni") # Overrides default value
```

4 Arbitrary Arguments (*args)

Used when multiple arguments are passed, but their count is unknown.

```
def total_sum(*numbers):
    return sum(numbers)

print(total_sum(5, 10, 15))
print(total_sum(1, 2, 3, 4, 5, 6))
```

Example: Using *args

5 Keyword Arguments (**kwargs)

Used when multiple named arguments are passed.

Example: Using **kwargs

```
def student_info(**details):
    for key, value in details.items():
        print(f"{key}: {value}")

student_info(name="Santhosh", age=21, course="Python")
```

6 Return Values

Functions can return values using the return keyword.

Example: Function Returning a Value

```
def square(num):
    return num * num

result = square(4)
print("Square:", result)
```

3.*args and **kwargs

In Python, *args and **kwargs are used to pass a variable number of arguments to a function.

1 *args (Arbitrary Positional Arguments)

- *args allows a function to accept multiple positional arguments as a tuple.
- We use * before a parameter name to indicate that it can take multiple values.

Example: Using *args

```
def add_numbers(*args):
    return sum(args)

print(add_numbers(1, 2, 3))      # Output: 6
print(add_numbers(5, 10, 15, 20)) # Output: 50
```

2 **kwargs (Arbitrary Keyword Arguments)

- **kwargs allows a function to accept multiple keyword arguments as a dictionary.
- We use ** before a parameter name to indicate that it can take key-value pairs.

Example: Using **kwargs

```
def student_info(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")
student_info(name="Santhosh", age=21, course="Python")
```

2 **kwargs (Arbitrary Keyword Arguments)

1.**kwargs allows a function to accept multiple keyword arguments as a dictionary.
2.We use ** before a parameter name to indicate that it can take key-value pairs.
Example: Using **kwargs

```
def student_info(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

student_info(name="Santhosh", age=21, course="Python")
```

3 Combining *args and **kwargs

We can use both *args and **kwargs together in a function.
Example: Mixing *args and **kwargs

```
def order_summary(order_id, *items, **details):
    print(f"Order ID: {order_id}")
      print("\nItems Ordered:")
    for item in items:
        print("-", item)
    print("\nOrder Details:")
    for key, value in details.items():
        print(f"{key}: {value}")
order_summary(
    101,
    "Pizza", "Burger", "Coke",
    name="Santhosh", address="Hyderabad", payment="Online"
)
```

4.Lambda Functions

A lambda function (also called an anonymous function) is a small, one-line function that does not require a name. It is used for short, simple operations where defining a full function is unnecessary.

1 Syntax of a Lambda Function

```
lambda arguments: expression
```

2 Example: Basic Lambda Function

```
square = lambda x: x * x
print(square(5))
```

3 Lambda Function with Multiple Arguments

```
add = lambda a, b: a + b
print(add(10, 20))
```

4 Using Lambda with map()

The map() function applies a lambda function to each element of a list.

Example: Squaring a List

```
numbers = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x ** 2, numbers))
print(squared)
```

5 Using Lambda with filter()

The filter() function selects elements from a list based on a condition.

Example: Filtering Even Numbers

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

5.Importing and Using Modules

1 What is a Module?

A module in Python is a file containing Python code (functions, classes, variables) that can be reused in other programs.

- Python has built-in modules (like math, random, os).
- You can also create your own modules.

2 Importing a Module

You can import a module using the import statement.

Example: Importing the math Module

```
import math
print(math.sqrt(25))  # Output: 5.0
print(math.pi)       # Output: 3.141592653589793
```

3 Importing Specific Functions

Instead of importing the whole module, you can import only specific functions.

Example: Importing Only sqrt and pi

```
from math import sqrt, pi
print(sqrt(16))  # Output: 4.0
print(pi)       # Output: 3.141592653589793
```

4 Importing a Module with an Alias

You can rename a module using as.

Example: Using an Alias for math

```
import math as m

print(m.sqrt(49))  # Output: 7.0
print(m.pi)       # Output: 3.141592653589793
```

5 Importing All Functions (*)

You can import everything from a module, but this is not recommended (it may cause conflicts).

Example: Import Everything

```
from math import *
print(sin(0))   # Output: 0.0
print(factorial(5))  # Output: 120
```

6 Creating Your Own Module

You can create a Python file and import it as a module.

Example: Creating a Custom Module

```
import greetings
print(greetings.hello("Santhosh"))  # Output: Hello, Santhosh!
print(greetings.goodbye("Chinni"))  # Output: Goodbye, Chinni!
```

7 Using dir() to List Module Contents

You can use dir() to see all functions in a module.

Example: Listing Functions in math

```
import math
print(dir(math))
```

8 Installing and Using External Modules

Some modules are not built-in, so you must install them using pip.

Example: Installing numpy

```
import numpy as np

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

Chapter 7
Data Structures - Lists and Tuples

1.Lists: Creation, Indexing, Slicing

A list in Python is a mutable, ordered collection that can store different data types like integers, strings, and even other lists.

1️⃣ Creating Lists

You can create a list using square brackets [] or the list() function.

Example: Creating Lists

```
# Empty List
empty_list = []
# List with Integers
numbers = [1, 2, 3, 4, 5]
# List with Mixed Data Types
mixed_list = [10, "Santhosh", 3.14, True]
# List using list() function
letters = list("Python")
print(numbers)     # Output: [1, 2, 3, 4, 5]
print(mixed_list)  # Output: [10, 'Santhosh', 3.14, True]
print(letters)     # Output: ['P', 'y', 't', 'h', 'o', 'n']
```

2️⃣ Indexing Lists

Lists use zero-based indexing, meaning the first element is at index 0.

Example: Accessing Elements

```
fruits = ["Apple", "Banana", "Cherry", "Date"]

print(fruits[0])  # Output: Apple
print(fruits[2])  # Output: Cherry
```

3️⃣ Slicing Lists

Slicing allows you to extract a subset of a list.

Syntax:

list[start:end:step]

1.start: The index where slicing begins (default = 0).

2.end: The index where slicing stops (not included).

3.step: The interval between elements (default = 1).

2.List Methods (append(), remove(), sort(), etc.)

1️⃣ Adding Elements to a List

🔹 append() → Add an element to the end

```
fruits = ["Apple", "Banana"]
fruits.append("Cherry")
print(fruits)  # Output: ['Apple', 'Banana', 'Cherry']
```

2️⃣ Removing Elements from a List

🔹 remove() → Remove a specific element

```
fruits.remove("Mango")
print(fruits)  # Output: ['Apple', 'Banana', 'Cherry', 'Orange', 'Grapes']
```

3️⃣ Finding Elements

🔹 index() → Find the index of an element

```
numbers = [10, 20, 30, 40, 50]
print(numbers.index(30))  # Output: 2
```

4️⃣ Sorting and Reversing

🔹 sort() → Sort the list (ascending by default)

```
numbers = [5, 2, 9, 1, 5, 6]
numbers.sort()
print(numbers)  # Output: [1, 2, 5, 5, 6, 9]
```

5 Copying a List

🔹 copy() → Create a copy of a list

```
original = [1, 2, 3]
copy_list = original.copy()
print(copy_list)  # Output: [1, 2, 3]
```

3.List Comprehensions

List comprehension is a concise and efficient way to create lists in Python. It helps write clean, readable, and faster code.

🔹 Basic Syntax

```
new_list = [expression for item in iterable if condition]
```

- expression → What to do with each item
- iterable → Any iterable (list, tuple, range, etc.)
- condition (optional) → Filter items based on a condition

Examples:

1. Creating a list from a range

```
squares = [x**2 for x in range(1, 6)]
print(squares)  # Output: [1, 4, 9, 16, 25]
```

2. Filtering even numbers

```
evens = [x for x in range(10) if x % 2 == 0]
print(evens)  # Output: [0, 2, 4, 6, 8]
```

3. Applying a function to each element

```
words = ["hello", "world", "python"]
upper_words = [word.upper() for word in words]
print(upper_words)  # Output: ['HELLO', 'WORLD', 'PYTHON']
```

4. Using if-else in list comprehension

```
labels = ["Even" if x % 2 == 0 else "Odd" for x in range(5)]
print(labels)  # Output: ['Even', 'Odd', 'Even', 'Odd', 'Even']
```

4.Tuples: Immutable Sequences

A tuple is an ordered, immutable collection of elements in Python. Unlike lists, tuples cannot be modified after creation, making them useful for storing fixed sets of data.

1. Using Parentheses ():

```
numbers = (1, 2, 3, 4, 5)
print(numbers)  # Output: (1, 2, 3, 4, 5)
```

2. Without Parentheses (Tuple Packing):

```
person = "Santhosh", 22, "India"
print(person)  # Output: ('Santhosh', 22, 'India')
```

3. Single Element Tuple (Comma Needed!):

```
single = (5,)
print(type(single))  # Output: <class 'tuple'>
```

Accessing Elements

Tuples support indexing and slicing like lists.

```
t = (10, 20, 30, 40, 50)

print(t[0])   # Output: 10  (First element)
print(t[-1])  # Output: 50  (Last element)
print(t[1:4])  # Output: (20, 30, 40) (Slicing)
```

Chapter 8:
Dictionaries and Sets

Dictionary Basics: Keys and Values

A dictionary in Python is an unordered, mutable collection that stores data in key-value pairs. It is defined using curly braces {} or the dict() function.
Creating a Dictionary

1. Using {}

```
student = {"name": "Santhosh", "age": 22, "course": "Python"}
print(student)
# Output: {'name': 'Santhosh', 'age': 22, 'course': 'Python'}
```

2. Using dict()

```
person = dict(name="Chinni", city="Hyderabad", age=21)
print(person)
# Output: {'name': 'Chinni', 'city': 'Hyderabad', 'age': 21}
```

Modifying Dictionary

1. Adding a New Key-Value Pair

```
student["grade"] = "A"
print(student)
# Output: {'name': 'Santhosh', 'age': 22, 'course': 'Python', 'grade': 'A'}
```

2. Updating an Existing Value

```
student["age"] = 23
print(student)
# Output: {'name': 'Santhosh', 'age': 23, 'course': 'Python', 'grade': 'A'}
```

2.Dictionary Methods (get(), update(), pop(), etc.)

Dictionaries in Python provide several built-in methods to manipulate key-value pairs efficiently. Here's a breakdown of commonly used methods with examples.

1. get() – Retrieve Value Safely

Returns the value for the specified key. If the key doesn't exist, it returns None (or a default value if provided).

```
student = {"name": "Santhosh", "age": 22, "course": "Python"}
print(student.get("name"))  # Output: Santhosh
print(student.get("grade"))  # Output: None (Key does not exist)
print(student.get("grade", "Not Assigned"))  # Output: Not Assigned
```

2. update() – Merge or Update Dictionary

Updates the dictionary with new key-value pairs or another dictionary.

```
student.update({"age": 23, "grade": "A"})  # Updates existing key & adds new one
print(student)
# Output: {'name': 'Santhosh', 'age': 23, 'course': 'Python', 'grade': 'A'}
# Using another dictionary
extra_info = {"city": "Hyderabad", "college": "Sri Indu"}
student.update(extra_info)
print(student)
# Output: {'name': 'Santhosh', 'age': 23, 'course': 'Python', 'grade': 'A', 'city': 'Hyderabad', 'college': 'Sri Indu'}
```

3. pop() – Remove and Return a Value

Removes the specified key and returns its value. If the key doesn't exist, it raises a KeyError.

```
removed_value = student.pop("age")
print(removed_value)  # Output: 23
print(student)  # 'age' key is removed
# Using a default value to avoid KeyError
print(student.pop("age", "Key not found"))  # Output: Key not found
```

4. popitem() – Remove and Return the Last Item

Removes and returns the last key-value pair (in Python 3.7+)

```
pair = student.popitem()
print(pair)  # Example Output: ('college', 'Sri Indu')
print(student)  # 'college' key is removed
```

5. keys() – Get All Keys

Returns a list-like view of all keys.

```
print(student.keys())  # Output: dict_keys(['name', 'course', 'grade', 'city'])
```

6. values() – Get All Values

Returns a list-like view of all values.

```
print(student.values())  # Output: dict_values(['Santhosh', 'Python', 'A',
'Hyderabad'])
```

7. items() – Get All Key-Value Pairs

Returns a list-like view of all (key, value) pairs.

```
print(student.items())
# Output: dict_items([('name', 'Santhosh'), ('course', 'Python'), ('grade', 'A'),
('city', 'Hyderabad')]
```

8. clear() – Remove All Elements

Deletes all key-value pairs, making the dictionary empty.

```
student.clear()
print(student)  # Output: {}
```

10. setdefault() – Get or Set a Default Value

Returns the value of a key if it exists; otherwise, inserts the key with a default value.

```
name = student.setdefault("name", "Unknown")
print(name)  # Output: 'Santhosh' (already exists)
city = student.setdefault("city", "Hyderabad")
print(city)  # Output: 'Hyderabad' (added as new key)
print(student)  # 'city' key is now in dictionary
```

3.Set Operations (union, intersection, difference)

A set in Python is an unordered, mutable collection of unique elements. It supports various mathematical operations like union, intersection, and difference.

1. Creating a Set

```
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}
print(A)  # Output: {1, 2, 3, 4, 5}
print(B)  # Output: {4, 5, 6, 7, 8}
```

2. Union (| or union())

Combines all elements from both sets without duplicates.

```
C = A | B  # Using `|` operator
print(C)  # Output: {1, 2, 3, 4, 5, 6, 7, 8}
D = A.union(B)  # Using `union()` method
print(D)  # Output: {1, 2, 3, 4, 5, 6, 7, 8}
```

3. Intersection (& or intersection())

Returns only the elements that are common in both sets.

```
C = A & B  # Using `&` operator
print(C)  # Output: {4, 5}
D = A.intersection(B)  # Using `intersection()` method
print(D)  # Output: {4, 5}
```

4. Difference (- or difference())

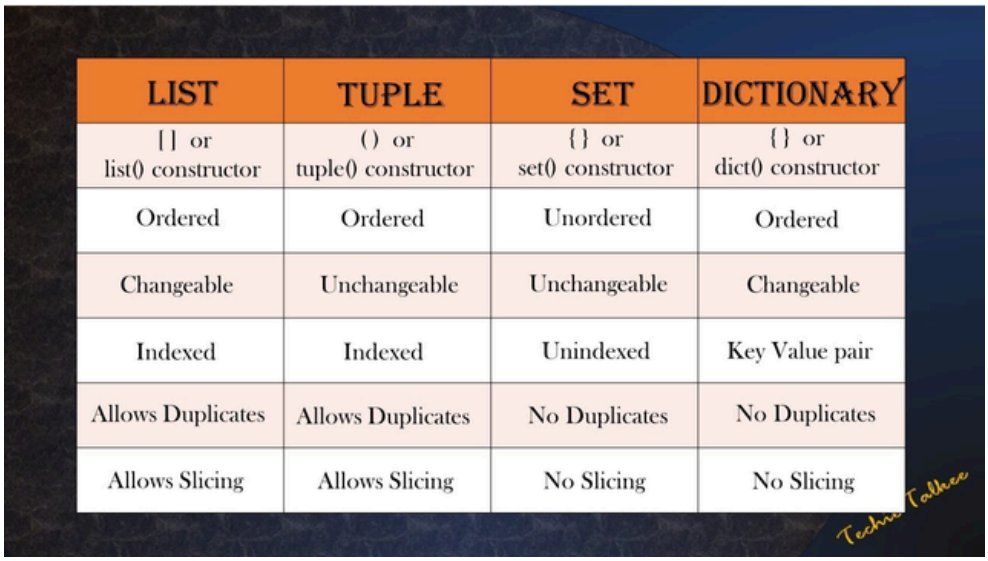
Returns elements that are only in the first set, removing common elements.

```
C = A - B  # Using `-` operator
print(C)  # Output: {1, 2, 3}
D = A.difference(B)  # Using `difference()` method
print(D)  # Output: {1, 2, 3}
E = B - A
print(E)  # Output: {6, 7, 8}  (elements only in B)
```

4.When to Use Dictionaries vs. Sets

Both dictionaries and sets are built-in data structures in Python, but they serve different purposes. Let's break down their key differences and when to use each.

1. Key Differences Between Dictionaries and Sets



LIST	TUPLE	SET	DICTIONARY
[] or list() constructor	() or tuple() constructor	{} or set() constructor	{} or dict() constructor
Ordered	Ordered	Unordered	Ordered
Changeable	Unchangeable	Unchangeable	Changeable
Indexed	Indexed	Unindexed	Key Value pair
Allows Duplicates	Allows Duplicates	No Duplicates	No Duplicates
Allows Slicing	Allows Slicing	No Slicing	No Slicing

2. When to Use a Dictionary?

Example Use Cases:

- **Storing user data ({ "name": "Santhosh", "age": 22 })**
- **Counting occurrences ({"apple": 3, "banana": 2})**
- **Caching results for fast lookups**
-

Example: Storing Student Information

```
student = {"name": "Santhosh", "age": 22, "course": "Python"}
print(student["name"])  # Output: Santhosh
```

Example: Word Frequency Counter

```
words = ["apple", "banana", "apple", "orange", "banana", "apple"]
word_count = {}
for word in words:
   word_count[word] = word_count.get(word, 0) + 1
print(word_count)  # Output: {'apple': 3, 'banana': 2, 'orange': 1}
```

3. When to Use a Set?

Use a set when you need a collection of unique items or need to perform set operations (union, intersection, difference, etc.).

Example Use Cases:

- **Removing duplicates from a list**
- **Checking membership (if x in set)**
- **Performing set operations (union, intersection, difference)**

Example: Removing Duplicates from a List

```
numbers = [1, 2, 2, 3, 4, 4, 5]
unique_numbers = set(numbers)
print(unique_numbers)  # Output: {1, 2, 3, 4, 5}
```

Example: Checking Membership (Faster than List)

```
prime_numbers = {2, 3, 5, 7, 11}
print(7 in prime_numbers)  # Output: True
print(9 in prime_numbers)  # Output: False
```

Example: Set Operations

```
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}
print(A | B)  # Union: {1, 2, 3, 4, 5, 6, 7, 8}
print(A & B)  # Intersection: {4, 5}
print(A - B)  # Difference: {1, 2, 3}
```

5. Conclusion

- **Use a dictionary (dict) when you need to map keys to values and perform fast lookups.**
- **Use a set (set) when you need a collection of unique items or want to perform set operations efficiently.**

Chapter 9:
Working with Strings

1.String Methods (strip(), split(), join(), etc.)

Strings in Python come with powerful built-in methods that help in text processing and manipulation. Let's explore some of the most useful string methods.

1. strip() – Remove Whitespace or Specific Characters

Removes leading and trailing whitespace (spaces, newlines, tabs) or specified characters.

```
text = "   Hello, Python!   "
print(text.strip())  # Output: "Hello, Python!"

custom_text = "---Hello---"
print(custom_text.strip("-"))  # Output: "Hello"
```

Related Methods:

- lstrip() → Removes leading (left) spaces/characters.
- rstrip() → Removes trailing (right) spaces/characters.

```
txt = "  Hello  "
print(txt.lstrip())  # Output: "Hello  "
print(txt.rstrip())  # Output: "  Hello"
```

2. split() – Convert String to a List

Splits a string into a list based on a delimiter (default: space).

```
sentence = "Python is awesome"
words = sentence.split()
print(words)  # Output: ['Python', 'is', 'awesome']
```

Split only a limited number of times:

```
data = "name:age:location"
print(data.split(":", 1))  # Output: ['name', 'age:location']
```

3. join() – Convert List to a String

Joins elements of a list into a single string using a specified separator.

```
words = ['Python', 'is', 'awesome']
sentence = " ".join(words)
print(sentence)  # Output: "Python is awesome"
```

4. replace() – Replace Substring in a String

Replaces a specific substring with another substring.

```
text = "I love Java"
new_text = text.replace("Java", "Python")
print(new_text)  # Output: "I love Python"
```

Replace multiple occurrences:

```
message = "apple apple apple"
print(message.replace("apple", "banana"))
# Output: "banana banana banana"
```

Replace only a limited number of times:

```
msg = "hello hello hello"
print(msg.replace("hello", "hi", 2))
# Output: "hi hi hello"
```

5. find() and index() – Search for Substrings

- find() returns the index of the first occurrence of a substring.
- index() works the same but raises an error if the substring is not found.

```
text = "Python is fun"
print(text.find("is"))  # Output: 7
print(text.index("is"))  # Output: 7
```

Handling missing substrings:

```
print(text.find("Java"))  # Output: -1
# print(text.index("Java"))  # Error: ValueError
```

2.String Formatting (f-strings, format())

String formatting in Python allows you to create dynamic strings by inserting variables into them. There are multiple ways to format strings, but the most commonly used methods are f-strings and the format() method.

1. f-strings (Formatted String Literals)

Introduced in Python 3.6, f-strings are the most concise and efficient way to format strings.

Syntax:

```
name = "Santhosh"
age = 21
print(f"My name is {name} and I am {age} years old.")
```

2. The format() Method

The format() method was introduced in Python 3.0 and allows placeholders {} to be replaced with variables.

Basic Syntax:

```
name = "Santhosh"
age = 21
print("My name is {} and I am {} years old.".format(name, age))
```

3. Formatting Numbers and Decimal Places

Both f-strings and format() allow number formatting.

Using f-strings:

```
pi = 3.1415926535
print(f"Pi rounded to 2 decimal places: {pi:.2f}")
```

4. Formatting Numbers with Commas

You can format large numbers with commas for better readability.

Using f-strings:

```
num = 1000000
print(f"Number with comma: {num:,}")
```

3.String Manipulation and Common Operations

Python provides powerful string manipulation techniques, allowing you to modify, format, and analyze strings efficiently.

1. Basic String Operations

Creating a String

```
text = "Hello, Python!"
```

Accessing Characters

```
print(text[0])   # H
print(text[-1])  # !
```

Slicing a String

```
print(text[0:5])    # Hello
print(text[:5])     # Hello
print(text[7:])     # Python!
print(text[-7:-1])  # Python
```

2. Modifying Strings

Converting Case

```
text = "Python Programming"
print(text.upper())   # PYTHON PROGRAMMING
print(text.lower())   # python programming
print(text.title())   # Python Programming
print(text.capitalize())  # Python programming
print(text.swapcase())    # pYTHON pROGRAMMING
```

Stripping Whitespace

```
text = "  Hello, Python!  "
print(text.strip())   # "Hello, Python!"
print(text.lstrip())  # "Hello, Python!  "
print(text.rstrip())  # "  Hello, Python!"
```

3. Checking String Content

Checking for Substrings

```
text = "Python is amazing"
print("Python" in text)   # True
print("Java" not in text) # True
```

Finding the Index of a Substring

```
text = "Learn Python Programming"
print(text.find("Python"))  # 6
print(text.index("Python")) # 6 (similar to find but raises an error if not found)
```

4. Splitting and Joining Strings

Splitting a String into a List

```
text = "apple,banana,orange"
print(text.split(","))  # ['apple', 'banana', 'orange']
```

5. Formatting Strings

Using f-strings

```
name = "Santhosh"
age = 21
print(f"My name is {name} and I am {age} years old.")
```

6. Reversing a String

```
text = "Python"
print(text[::-1])  # nohtyP
```

7. Checking if a String is Alphabetic/Numeric

```
print("Hello".isalpha())  # True
print("12345".isdigit())  # True
print("Hello123".isalnum())  # True
print("  ".isspace())  # True
```

Chapter 10
File Handling

1.Reading and Writing Files (open(), read(), write())

In Python, file handling is done using built-in functions like open(), read(), and write(). Here's how you can use them:

1. Opening a File (open())

The open() function is used to open a file. It has the syntax:

```
file = open("filename", "mode")
```

Modes available:

- "r" → Read mode (default, file must exist)
- "w" → Write mode (creates a new file or overwrites an existing file)
- "a" → Append mode (adds content to an existing file)
- "r+" → Read and write
- "w+" → Write and read (overwrites)
- "a+" → Append and read
-

2. Reading a File (read(), readline(), readlines())

Once a file is opened in read mode, you can read its contents using:

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

3. Writing to a File (write(), writelines())

Writing to a file requires "w" or "a" mode.

```
file = open("example.txt", "w")  # Opens file for writing (overwrites)
file.write("Hello, World!\n")  # Writes text
file.close()
```

4. Using with Statement (Best Practice)

Using with open() automatically closes the file after use.

```
with open("example.txt", "r") as file:
    content = file.read()
    print(content)
# No need to call file.close()
```

2.Working with Different File Modes (r, w, a, r+)

In Python, when working with files, different file modes determine how a file is opened and how data can be read or written. Below are the commonly used file modes:

1. Read Mode (r)

1.Opens a file for reading only.
2.If the file does not exist, it raises a FileNotFoundError.
3.The file pointer is placed at the beginning of the file.with open("example.txt", "r") as file:

```
content = file.read()
print(content)
```

2. Write Mode (w)

- Opens a file for writing.
- If the file does not exist, it creates a new one.
- If the file exists, it overwrites the content.

```
with open("example.txt", "w") as file:
 file.write("Hello, this is a new file content.")
```

3. Append Mode (a)

Opens a file for appending new data at the end.
If the file does not exist, it creates a new one.
Existing content is not erased.

```
with open("example.txt", "a") as file:
    file.write("\nThis is appended content.")
```

3.Handling File Exceptions

Handling file exceptions in programming, especially in Python, is crucial to ensure that your program runs smoothly even when unexpected issues arise, such as missing files, permission errors, or incorrect file formats.

Common File Exceptions in Python

1. FileNotFoundError – When the file does not exist.
2. PermissionError – When the user does not have the required permissions.
3. IsADirectoryError – When a directory is provided instead of a file.
4. IOError – General input/output errors.
5. EOFError – Unexpected end of file.

Best Practices for Handling File Exceptions

Use a try-except block to catch and handle exceptions gracefully.

Example 1: Handling File Not Found

```
try:
    with open("data.txt", "r") as file:
        content = file.read()
        print(content)
except FileNotFoundError:
    print("Error: The file was not found. Please check the file path.")
```

Example 2: Using finally to Ensure Cleanup

```
try:
    file = open("data.txt", "r")
    content = file.read()
    print(content)
except FileNotFoundError:
    print("Error: File not found.")
finally:
    if 'file' in locals() and not file.closed:
        file.close()
        print("File closed successfully.")
```

Chapter 11
Exception Handling

1.Try, except, finally, and else

In Python, the try, except, finally, and else blocks are used for handling exceptions. Here's how each of them works:

1. try Block

- The code inside the try block is executed first.
- If an exception occurs, it is caught by the except block.
- If no exception occurs, the else block (if present) is executed.

2. except Block

- This block is executed if an exception occurs inside the try block.
- Multiple except blocks can handle different exceptions.

3. else Block (Optional)

- If no exception occurs in the try block, the else block is executed.
- It is used when you want to run some code only if no errors occur.

4. finally Block (Optional)

- The finally block is always executed, regardless of whether an exception occurs or not.
- It is commonly used for cleanup tasks like closing files or database connections.

Example 1: Multiple except blocks

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except ValueError:
    print("Invalid input! Please enter a number.")
except ZeroDivisionError:
    print("Cannot divide by zero!")
```

2.Handling Multiple Exceptions

In Python, you can handle multiple exceptions using different approaches:

1. Using Multiple except Blocks

You can catch specific exceptions separately by defining multiple except blocks.

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except ValueError:
    print("Invalid input! Please enter a valid number.")
except ZeroDivisionError:
    print("Cannot divide by zero!")
```

2. Using a Single except Block with Multiple Exceptions

You can handle multiple exceptions in a single block using a tuple.

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except (ValueError, ZeroDivisionError) as e:
    print(f"Error: {e}")
```

3. Using a Generic except Block

Catching all exceptions using except Exception can be useful in some cases but should be used carefully.

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except Exception as e:
    print(f"An error occurred: {e}")
```

4. Using finally with Multiple Exceptions

The finally block always executes, whether an exception occurs or not.

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except (ValueError, ZeroDivisionError) as e:
    print(f"Error: {e}")
finally:
    print("Execution completed!")
```

5. Using else with try-except

The else block runs only if no exception occurs.

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except (ValueError, ZeroDivisionError) as e:
    print(f"Error: {e}")
else:
    print(f"Result: {result}")
finally:
    print("Execution completed!")
```

3.Raising Custom Exceptions

In Python, you can raise custom exceptions by defining your own exception classes. This is useful when you need to handle specific error cases in a more descriptive and structured way.

Steps to Raise Custom Exceptions:

1. Define a Custom Exception Class
   - Inherit from Exception or a built-in exception class like ValueError.
   - Optionally, define an __init__ method to customize the exception.
2. Raise the Custom Exception
   - Use the raise keyword when an error condition is met.

Example:

```
# Define a custom exception
class CustomError(Exception):
    def __init__(self, message):
        self.message = message
        super().__init__(self.message)

# Function using custom exception
def check_value(x):
    if x < 0:
        raise CustomError("Value cannot be negative")
    return f"Valid value: {x}"

# Testing the function
try:
    print(check_value(-5))
except CustomError as e:
    print(f"Caught an error: {e}")
```

Output:

Caught an error: Value cannot be negative

Why Use Custom Exceptions?

- Provides clearer error messages specific to your application.
- Makes debugging easier by categorizing errors.
- Helps in maintaining clean code by avoiding generic exceptions like Exception.

Chapter 12
Object-Oriented Programming (OOP)

1.Classes and Objects

In Python, you can raise custom exceptions by defining your own exception classes. This is useful when you need to handle specific error cases in a more descriptive and structured way.

Steps to Raise Custom Exceptions:

1. Define a Custom Exception Class
   - Inherit from Exception or a built-in exception class like ValueError.
   - Optionally, define an __init__ method to customize the exception.
2. Raise the Custom Exception
   - Use the raise keyword when an error condition is met.

Example:

```
# Define a custom exception
class CustomError(Exception):
    def __init__(self, message):
        self.message = message
        super().__init__(self.message)

# Function using custom exception
def check_value(x):
    if x < 0:
        raise CustomError("Value cannot be negative")
    return f"Valid value: {x}"

# Testing the function
try:
    print(check_value(-5))
except CustomError as e:
    print(f"Caught an error: {e}")
```

2.Constructor (__init__) and Instance Variables

In Python, a constructor is a special method called __init__, which is automatically invoked when a new instance of a class is created. It is used to initialize instance variables (attributes specific to each object).

Constructor (__init__ method)

- The __init__ method is called when an object is created.
- It is used to assign values to instance variables.
- The self parameter refers to the instance of the class.
-

Instance Variables

- These are variables that belong to an instance of a class.
- They are defined inside the constructor (__init__) using self.

Example:

```
class Student:
    def __init__(self, name, age, course):
        self.name = name        # Instance variable
        self.age = age          # Instance variable
        self.course = course    # Instance variable

    def display_info(self):
        print(f"Name: {self.name}, Age: {self.age}, Course: {self.course}")

# Creating objects (instances)
student1 = Student("Hari", 21, "Python")
student2 = Student("Santhosh", 22, "Java")

# Accessing instance variables
student1.display_info()  # Output: Name: Hari, Age: 21, Course: Python
student2.display_info()  # Output: Name: Santhosh, Age: 22, Course: Java
```

3.Methods and self

In Python, methods are functions that are associated with objects and can operate on their data. When a method is defined inside a class, it typically takes self as its first parameter.

What is self?

- self represents the instance of the class and allows access to its attributes and methods.
- It must be explicitly included as the first parameter in instance methods.
- It helps differentiate between instance variables and local variables.

Example of Methods and self

```
class Student:
    def __init__(self, name, age):
        self.name = name  # Instance variable
        self.age = age    # Instance variable

    def display_info(self):  # Instance method
        print(f"Name: {self.name}, Age: {self.age}")
# Creating an object
student1 = Student("Santhosh", 21)

# Calling the method
student1.display_info()  # Output: Name: Santhosh, Age: 21
```

Types of Methods in Python

1. Instance Methods (Use self):
   - Operate on instance attributes.
   - Example: display_info() above.
2. Class Methods (Use cls):
   - Use the @classmethod decorator.
   - Work with class attributes, not instance attributes.

4.Inheritance and Polymorphism

Inheritance and Polymorphism in Object-Oriented Programming (OOP)

Inheritance and polymorphism are two fundamental concepts in Object-Oriented Programming (OOP) that enhance code reusability and flexibility.

1. Inheritance

Inheritance allows a child class (subclass) to acquire properties and behaviors (methods) from a parent class (superclass). This promotes code reuse and hierarchical relationships between classes.

Key Points:

- The child class inherits attributes and methods from the parent class.
- The child class can extend or override methods from the parent class.
- Helps in implementing code reusability and modular programming.

```
# Parent class
class Animal:
    def __init__(self, name):
        self.name = name
    def speak(self):
        return "Animal makes a sound"
# Child class inheriting from Animal
class Dog(Animal):
    def speak(self):
        return "Woof! Woof!"
# Child class inheriting from Animal
class Cat(Animal):
    def speak(self):
        return "Meow! Meow!"
# Creating objects
dog = Dog("Buddy")
cat = Cat("Kitty")
print(dog.name, "says:", dog.speak())  # Buddy says: Woof! Woof!
print(cat.name, "says:", cat.speak())  # Kitty says: Meow! Meow!
```

Types of Inheritance:

1. **Single Inheritance –** One child class inherits from one parent class.
2. **Multiple Inheritance –** A child class inherits from multiple parent classes.
3. **Multilevel Inheritance –** A class inherits from another class, which in turn inherits from another class.
4. **Hierarchical Inheritance –** Multiple child classes inherit from the same parent class.
5. **Hybrid Inheritance –** A combination of two or more types of inheritance.

2. Polymorphism

Polymorphism allows objects of different classes to be treated as objects of a common superclass. It enables methods to have the same name but different behaviors in different classes.

Key Points:

- A single method can be implemented in different ways in different classes.
- Helps in flexibility and scalability of the code.
-

Example (Method Overriding in Python)

```
class Bird:
    def fly(self):
        return "Birds can fly"

class Sparrow(Bird):
    def fly(self):
        return "Sparrows can fly fast"
class Penguin(Bird):
    def fly(self):
        return "Penguins cannot fly"
# Creating objects
sparrow = Sparrow()
penguin = Penguin()
print(sparrow.fly())  # Sparrows can fly fast
print(penguin.fly())  # Penguins cannot fly
```

Types of Polymorphism:

1. **Method Overloading** – Defining multiple methods with the same name but different parameters (Not directly supported in Python).
2. **Method Overriding** – Redefining a parent class method in the child class.
3. **Operator Overloading** – Using operators (+, -, *, etc.) with user-defined meanings.

Example (Operator Overloading)

```
class Number:
    def __init__(self, value):
        self.value = value

    def __add__(self, other):
        return self.value + other.value

num1 = Number(10)
num2 = Number(20)

print(num1 + num2)  # Output: 30
```

5.Encapsulation and Abstraction

Encapsulation and Abstraction are two fundamental concepts of Object-Oriented Programming (OOP) in languages like Java, Python, and C++.

Encapsulation

Encapsulation is the practice of bundling data (variables) and methods (functions) that operate on the data into a single unit (class). It restricts direct access to some of the object's components, which helps prevent accidental modification.

🔹 Key Features of Encapsulation:

- Protects data by restricting access using access modifiers (private, protected, public).
- Provides getter and setter methods to access and update private variables.
- Increases data security and code maintainability.

✅ Example in Python:

```
class BankAccount:
    def __init__(self, account_number, balance):
        self.__account_number = account_number  # Private variable
        self.__balance = balance  # Private variable

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
            print(f"Deposited: {amount}, New Balance: {self.__balance}")

    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount
            print(f"Withdrawn: {amount}, New Balance: {self.__balance}")
        else:
            print("Insufficient funds!")

    def get_balance(self):
        return self.__balance  # Getter method
```

```
# Usage
account = BankAccount(12345, 5000)
account.deposit(2000)
account.withdraw(3000)
print("Balance:", account.get_balance())
```

Abstraction

Abstraction is the process of hiding implementation details and only exposing the essential features of an object. It helps simplify complex systems by providing a clear interface.

🔹 Key Features of Abstraction:

- Hides unnecessary details from the user.
- Achieved using abstract classes and interfaces.
- Improves code reusability and scalability.
-

Example in Python (Using Abstract Class):

```
from abc import ABC, abstractmethod
class Vehicle(ABC):  # Abstract class
    @abstractmethod
    def start(self):
        pass  # Abstract method
    @abstractmethod
    def stop(self):
        pass  # Abstract method
class Car(Vehicle):
    def start(self):
        print("Car is starting...")
    def stop(self):
        print("Car is stopping...")
# Usage
my_car = Car()
my_car.start()
my_car.stop()
```

Chapter 13:
Working with Modules and Packages

1.Built-in Modules (math, random, os, etc.)

Python provides several built-in modules that offer useful functions and tools. Here are some commonly used built-in modules:

1. math Module (Mathematical Functions)

Provides mathematical functions like square root, logarithms, trigonometric functions, etc.

```
import math
print(math.sqrt(25))  # Output: 5.0
print(math.factorial(5))  # Output: 120
print(math.pi)  # Output: 3.141592653589793
```

2. random Module (Random Number Generation)

Used for generating random numbers, selecting random choices, etc.

```
import random
print(random.randint(1, 10))  # Random integer between 1 and 10
print(random.choice(['apple', 'banana', 'cherry']))  # Random choice from a list
print(random.uniform(1, 5))  # Random float between 1 and 5
```

3. os Module (Operating System Interface)

Provides functions to interact with the operating system.

```
import os
print(os.name)  # Name of the OS
print(os.getcwd())  # Get current working directory
os.mkdir("test_folder")  # Create a new directory
```

4. sys Module (System-Specific Parameters and Functions)

- **Gives access to system-related information.**

```
import sys
print(sys.version)  # Python version
print(sys.platform)  # OS platform
sys.exit()  # Exit the program
```

5. time Module (Time-Related Functions)

Helps in handling time-related tasks.

```
import time
print(time.time())  # Current time in seconds
time.sleep(2)  # Pause execution for 2 seconds
print(time.ctime())  # Get current time as a string
```

6. datetime Module (Date and Time Functions)

Used for working with dates and times.

```
from datetime import datetime
now = datetime.now()
print(now)  # Current date and time
print(now.strftime("%Y-%m-%d %H:%M:%S"))  # Format date
```

7. json Module (Working with JSON Data)

Helps in encoding and decoding JSON data.

```
import json
data = {"name": "John", "age": 30}
json_data = json.dumps(data)  # Convert to JSON string
print(json_data)
print(json.loads(json_data))  # Convert JSON string to Python dictionary
```

10. socket Module (Network Communication)

Used for working with network sockets.

```
import socket
host = socket.gethostname()
print(host)  # Output: System hostname
```

Creating and Importing Custom Modules

Creating and importing custom modules in Python is an essential skill for organizing code and reusing functions across different programs. Here's a breakdown of the process:

1. Creating a Custom Module

A module in Python is simply a file with a .py extension that contains Python code, such as functions, classes, or variables.

Example: Creating a module (mymodule.py)

```
# mymodule.py
def greet(name):
    return f"Hello, {name}!"

def add(a, b):
    return
```

2. Storing and Using Modules

- Place the module file (mymodule.py) in the same directory as the script that imports it.
- If the module is in a different directory, modify sys.path or set the PYTHONPATH environment variable.

Using a module from a different directory

```
import sys
sys.path.append("/path/to/module")

import mymodule
```

3. Using Built-in importlib for Dynamic Importing

```
import importlib

mymodule = importlib.import_module("mymodule")
print(mymodule.greet("Santhosh"))
```

3.Installing and Using External Libraries (pip)

Installing and Using External Libraries in Python (pip)
Python has a vast ecosystem of third-party libraries that extend its functionality. These libraries can be installed and managed using pip, Python's package manager.

1.Using Installed Libraries

Once you've installed a library using pip, you can start using it in your Python programs by importing it. Here's a step-by-step guide on how to effectively use installed libraries

2. Upgrading a Library

To update a package:
pip install --upgrade library_name

3. Uninstalling a Library

If you no longer need a library, uninstall it using:
import requests

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

4. Virtual Environments (Recommended)

It is a good practice to use virtual environments to manage dependencies separately for different projects.

Creating a Virtual Environment

python -m venv myenv

Activate it:

- **Windows: myenv\Scripts\activate**
- **Mac/Linux: source myenv/bin/activate**

Now, you can install packages inside this isolated environment.

Chapter 14
Working with Databases (SQLite & MySQL)

1.Introduction to Databases

Introduction to Databases

A database is an organized collection of data that allows efficient storage, retrieval, and management of information. It serves as a backbone for applications and systems that require structured data handling, such as websites, banking systems, and enterprise applications.

Why Use Databases?

1. **Efficient Data Management –** Stores and organizes large volumes of data efficiently.
2. **Data Integrity & Consistency –** Ensures that stored data remains accurate and reliable.
3. **Security –** Provides authentication and authorization to protect sensitive information.
4. **Concurrency Control –** Supports multiple users accessing data simultaneously.
5. **Data Recovery –** Allows data backup and restoration in case of failure.

Types of Databases

1. **Relational Databases** (RDBMS)
   - Uses tables with rows and columns to store data.
   - Follows Structured Query Language (SQL).
   - Examples: MySQL, PostgreSQL, Oracle, SQL Server.
2. **NoSQL Databases**
   - Suitable for unstructured or semi-structured data.
   - Does not rely on traditional tables; uses key-value pairs, documents, or graphs.
   - Examples: MongoDB, Cassandra, Redis, Neo4j.
3. **Cloud Databases**
   - Hosted on cloud platforms for scalability and availability.
   - Examples: AWS RDS, Google Cloud Firestore, Azure SQL Database.

4.Time-Series Databases

- Specially designed for tracking changes over time.
- Examples: InfluxDB, TimescaleDB.

Database Management System (DBMS)

A DBMS is software that allows users to create, manage, and interact with databases.

- Examples: MySQL, MongoDB, SQLite, Firebase.

Functions of DBMS:

- Data Storage, Retrieval, and Management
- Multi-user Access
- Security and Backup
- Data Integrity Enforcement

Basic Database Operations

- CREATE – Creating tables and defining structures.
- READ – Retrieving data using queries.
- UPDATE – Modifying existing data.
- DELETE – Removing unnecessary records.

2.Connecting to SQLite/MySQL

Connecting to SQLite and MySQL in Python

1. Connecting to SQLite

SQLite is a lightweight database that doesn't require a separate server. You can use the sqlite3 module in Python.

Steps to Connect to SQLite:

```
import sqlite3
# Connect to database (or create it if it doesn't exist)
conn = sqlite3.connect('my_database.db')
# Create a cursor object to execute SQL commands
cursor = conn.cursor()
# Example: Creating a table
cursor.execute('''CREATE TABLE IF NOT EXISTS users (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
```

```
name TEXT NOT NULL,
 age INTEGER)'''
# Commit changes and close the connection
conn.commit()
conn.close()
```

2. Connecting to MySQL

To connect to MySQL, you need the mysql-connector-python package. Install it if you haven't:

Steps to Connect to MySQL:

```
import mysql.connector

# Connect to MySQL server
conn = mysql.connector.connect(
   host="your_host",     # Example: "localhost"
   user="your_user",     # Example: "root"
   password="your_pass", # Your MySQL password
   database="your_db"    # Your database name
)

# Create a cursor object
cursor = conn.cursor()

# Example: Creating a table
cursor.execute('''CREATE TABLE IF NOT EXISTS users (
                id INT AUTO_INCREMENT PRIMARY KEY,
                name VARCHAR(255) NOT NULL,
                age INT)''')

# Commit changes and close the connection
conn.commit()
conn.close()
```

4.Using sqlite3 and mysql-connector-python

if you want to use both sqlite3 and mysql-connector-python in the same Python project, you can follow this guide.

Installing Required Packages

Ensure you have the necessary libraries installed:

```
pip install mysql-connector-python
```

Using SQLite3 in Python

Connecting to an SQLite Database

```
import sqlite3
# Connect to SQLite database (creates a file if it doesn't exist)
conn = sqlite3.connect("my_database.db")
cursor = conn.cursor()
# Create a table
cursor.execute('''CREATE TABLE IF NOT EXISTS users (id INTEGER PRIMARY KEY, name TEXT, age INTEGER)''')
# Insert data
cursor.execute("INSERT INTO users (name, age) VALUES (?, ?)", ("Alice", 25))
# Commit and close
conn.commit()
conn.close()
```

Using MySQL Connector in Python

Connecting to a MySQL Database

```
import mysql.connector

# Connect to MySQL server
conn = mysql.connector.connect(
    host="localhost",
    user="your_username",
    password="your_password",
    database="your_database"
```

```
cursor = conn.cursor()
# Create a table
cursor.execute('''CREATE TABLE IF NOT EXISTS users (id INT AUTO_INCREMENT
PRIMARY KEY, name VARCHAR(255), age INT)''')
# Insert data
cursor.execute("INSERT INTO users (name, age) VALUES (%s, %s)", ("Bob", 30))
# Commit and close
conn.commit()
conn.close()
```

Migrating Data from SQLite to MySQL

If you want to transfer data from an SQLite database to a MySQL database:

```
import sqlite3
import mysql.connector
# Connect to SQLite
sqlite_conn = sqlite3.connect("my_database.db")
sqlite_cursor = sqlite_conn.cursor()
# Connect to MySQL
mysql_conn = mysql.connector.connect(
    host="localhost",
    user="your_username",
    password="your_password",
    database="your_database"
mysql_cursor = mysql_conn.cursor()
# Fetch all data from SQLite
sqlite_cursor.execute("SELECT * FROM users")
rows = sqlite_cursor.fetchall()
# Insert data into MySQL
for row in rows:
    mysql_cursor.execute("INSERT INTO users (id, name, age) VALUES (%s, %s,
%s)", row)
# Commit and close connections
mysql_conn.commit()
```

Chapter 15
Web Scraping with Python

1.Introduction to Web Scraping

Introduction to Web Scraping

Web scraping is the process of extracting data from websites using automated scripts or programs. It enables users to collect, analyze, and store data from the web efficiently. Web scraping is widely used in industries such as e-commerce, finance, research, and marketing.

Why Use Web Scraping?

1. **Data Collection –** Automates the retrieval of large volumes of data.
2. **Market Research –** Gathers competitor prices, product details, and customer reviews.
3. **Lead Generation –** Extracts contact information from business directories.
4. **Sentiment Analysis –** Collects user feedback from social media and reviews.
5. **News Aggregation –** Compiles articles from multiple sources.

How Web Scraping Works

1. **Send a Request – The scraper sends an HTTP request to the target website.**
2. **Retrieve the HTML – The website returns the page's HTML content.**
3. **Parse the Data – The scraper extracts useful data from the HTML using parsing libraries.**
4. **Store the Data – The extracted data is saved in formats like CSV, JSON, or databases.**

Common Web Scraping Tools & Libraries

- **Python**
  - **BeautifulSoup – Parses HTML/XML data easily.**
  - **Scrapy – A full-fledged web scraping framework.**
  - **Selenium – Automates browser interactions for dynamic pages.**
  - **Requests – Sends HTTP requests to web pages.**
- **Other Tools**
  - **Puppeteer – A headless browser tool for JavaScript-based scraping.**
  - **Octoparse – A no-code visual scraping tool.**

Ethical Considerations & Legal Issues

- Respect Robots.txt – Websites specify scraping rules in robots.txt.
- Avoid Overloading Servers – Limit requests to prevent server crashes.
- No Personal Data Scraping – Comply with GDPR and other data protection laws.
- Obtain Permission – Some websites require legal consent for scraping.

2.Using requests to Fetch Web Pages

Using the requests library in Python, you can fetch web pages easily. Below is a simple guide:

1. Install requests

If you haven't already, install it using:

```
pip install requests
```

2. Handling Errors Gracefully

It’s good practice to handle errors:

```
    response = requests.get(url, timeout=5)  # Set a timeout
    response.raise_for_status()  # Raises an error for HTTP errors (4xx and 5xx)
    print("Page Content:", response.text)
except requests.exceptions.RequestException as e:
    print("Error:", e)
```

5. Fetching JSON Data

If the page returns JSON data:

```
response = requests.get("https://jsonplaceholder.typicode.com/todos/1")
data = response.json()  # Convert to Python dictionary
print(data)
```

6. Passing Parameters

For URLs that require query parameters:

```
params = {"q": "Python", "sort": "recent"}
response = requests.get("https://example.com/search", params=params)
print(response.url)  # Shows the final URL with parameters
```

3.Parsing HTML with BeautifulSoup

Parsing HTML with BeautifulSoup is a common technique in web scraping and data extraction from web pages. BeautifulSoup is a Python library that allows you to parse HTML and XML documents easily.

Basic Usage

1. Import Required Libraries

```
from bs4 import BeautifulSoup
import requests
```

2. Fetching and Parsing HTML

```
url = "https://example.com"
response = requests.get(url)

# Create a BeautifulSoup object
soup = BeautifulSoup(response.text, "html.parser")
```

Finding Elements in HTML

1. Finding by Tag Name

```
title = soup.title  # Gets the <title> tag
print(title.text)   # Prints the text inside <title>
```

2. Finding by ID

```
element = soup.find(id="main-content")
print(element)
```

3. Finding by Class Name

```
elements = soup.find_all(class_="article")
for element in elements:
    print(element.text)
```

Navigating HTML Elements

1. Parent and Child Elements

```
parent = soup.find("p").parent  # Gets the parent element of <p>
children = soup.body.find_all("div")  # Finds all <div> inside <body>
```

2. Next and Previous Elements

```
next_sibling = soup.find("h2").find_next_sibling()
prev_sibling = soup.find("h2").find_previous_sibling()
```

Extracting Text and Attributes

```
text = soup.get_text()  # Gets all text from HTML
print(text)

first_link = soup.find("a")
print(first_link["href"])  # Extracts the href attribute
```

Advanced Searching with CSS Selectors

```
headings = soup.select("h1, h2, h3")  # Selects all <h1>, <h2>, <h3>
specific = soup.select_one(".main-class")  # Selects first element with
class="main-class"
```

Example: Scraping a Website

```
url = "https://news.ycombinator.com/"
response = requests.get(url)
soup = BeautifulSoup(response.text, "html.parser")

# Extract all headlines
headlines = soup.find_all("a", class_="storylink")

for headline in headlines:
    print(headline.text)
    print(headline["href"])
```

4.Extracting Data and Saving Results

Extracting data and saving results can involve multiple approaches, depending on the source of the data and the desired format for storage. Here's a general workflow:

1. Identify the Data Source

- Web Scraping: Extract data from websites using tools like BeautifulSoup, Scrapy, or Selenium.
- APIs: Use requests to fetch data from web APIs (requests, http.client).
- Databases: Query data from SQL or NoSQL databases (MySQL, PostgreSQL, MongoDB).
- Local Files: Read from CSV, JSON, Excel, or text files (pandas, openpyxl).

2. Extract Data

- Web Scraping Example (using BeautifulSoup):

```
import requests
from bs4 import BeautifulSoup

url = "https://example.com"
response = requests.get(url)
soup = BeautifulSoup(response.text, 'html.parser')

data = soup.find_all("p")  # Extracting paragraph texts
extracted_data = [p.text for p in data]
```

3. Save the Extracted Data

- Save to CSV:

```
import pandas as pd

df = pd.DataFrame(data)
df.to_csv("output.csv", index=False)
```

Save to a Database:

```
cursor.execute("INSERT INTO users (name, age) VALUES (%s, %s)", ("John", 30))
connection.commit()
```

Chapter 16
Data Visualization with Matplotlib and Seaborn

1.Introduction to matplotlib

Introduction to Matplotlib

Matplotlib is a popular Python library used for data visualization. It provides a wide variety of plotting options to create high-quality static, animated, and interactive graphs.

Why Use Matplotlib?

- **Simple to use and highly customizable.**
- **Works well with other Python libraries like NumPy and Pandas.**
- **Supports various plot types, such as line plots, bar charts, histograms, scatter plots, and more.**
- **Allows exporting graphs in different formats (PNG, PDF, SVG, etc.).**
- **Interactive features like zooming and panning.**

Creating a Simple Line Plot

```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5]
y = [10, 20, 25, 30, 50]

plt.plot(x, y, marker='o', linestyle='-', color='b', label="Line 1")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.title("Simple Line Plot")
plt.legend()
plt.show()
```

Common Types of Plots

1.Line Plot:

```
plt.plot(x, y)
```

2.Plotting Line, Bar, and Scatter Plots

Here's a simple Python script using Matplotlib to plot line, bar, and scatter plots:

```
import matplotlib.pyplot as plt
import numpy as np
# Sample data
x = np.array([1, 2, 3, 4, 5])
y = np.array([10, 20, 15, 25, 30])
# Create a figure with subplots
fig, axs = plt.subplots(1, 3, figsize=(15, 5))
# Line plot
axs[0].plot(x, y, marker='o', linestyle='-', color='b', label='Line Plot')
axs[0].set_title('Line Plot')
axs[0].set_xlabel('X-axis')
axs[0].set_ylabel('Y-axis')
axs[0].legend()
# Bar plot
axs[1].bar(x, y, color='g', label='Bar Plot')
axs[1].set_title('Bar Plot')
axs[1].set_xlabel('X-axis')
axs[1].set_ylabel('Y-axis')
axs[1].legend()
# Scatter plot
axs[2].scatter(x, y, color='r', label='Scatter Plot')
axs[2].set_title('Scatter Plot')
axs[2].set_xlabel('X-axis')
axs[2].set_ylabel('Y-axis')
axs[2].legend()
# Show the plots
plt.tight_layout()
plt.show()
```

Explanation:

- Line Plot: Uses plot() with markers and a line connecting the points.
- Bar Plot: Uses bar() to create vertical bars.
- Scatter Plot: Uses scatter() to plot individual points.

3.Styling Graphs and Adding Labels

Styling graphs and adding labels make data visualization more effective and visually appealing. Here's how you can enhance your graphs using Python's Matplotlib library:

1. Adding Labels and Titles

- plt.xlabel(): Adds a label to the x-axis.
- plt.ylabel(): Adds a label to the y-axis.
- plt.title(): Sets the title of the graph.
- plt.legend(): Adds a legend to distinguish different data series.
-

2. Customizing the Appearance

- Colors: You can use colors like 'red', 'blue', or hex codes like #ff5733.
- Line Styles: '-' (solid), '--' (dashed), ':' (dotted), etc.
- Markers: 'o' (circle), 's' (square), '^' (triangle), etc.
-

3. Changing Font Styles

You can modify font properties using:

```
plt.xlabel("X-Axis Label", fontsize=12, fontweight='bold', color='blue')
plt.ylabel("Y-Axis Label", fontsize=12, fontstyle='italic', color='green')
plt.title("Graph Title", fontsize=14, fontweight='bold', color='purple')
```

4.Using seaborn for Advanced Visualization

Seaborn is a powerful Python library for statistical data visualization, built on top of Matplotlib. It provides advanced visualization capabilities with simple and intuitive syntax.

1. Why Use Seaborn for Advanced Visualization?

- **Enhances Matplotlib's capabilities with more aesthetic and informative visualizations.**
- **Built-in themes and color palettes for better visual appeal.**
- **Simplifies the creation of complex plots.**
- **Supports Pandas DataFrames for direct visualization of datasets.**
-

2. Advanced Visualizations with Seaborn

A. Heatmaps (Visualizing Correlations)

Heatmaps are useful for visualizing correlations and relationships between numerical variables.

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

# Sample Data
data = pd.DataFrame({
   'A': [1, 2, 3, 4, 5],
   'B': [2, 3, 4, 5, 6],
   'C': [5, 4, 3, 2, 1]
})

# Compute correlation matrix
corr = data.corr()
# Create heatmap
sns.heatmap(corr, annot=True, cmap='coolwarm', linewidths=0.5)
plt.title("Correlation Heatmap")
plt.show()
```

B. Regression Plot (Visualizing Trends)

Regression plots show the relationship between two variables along with a regression line.

```
# Load example dataset
tips = sns.load_dataset("tips")

# Regression plot
sns.lmplot(x="total_bill", y="tip", data=tips, hue="sex", markers=["o", "s"])
plt.title("Regression Plot of Total Bill vs Tip")
plt.show()
```

C.Time Series Plots

Seaborn can handle time-series data effectively.

```
# Load example time-series dataset
flights = sns.load_dataset("flights")

# Convert to pivot table
flights_pivot = flights.pivot("month", "year", "passengers")

# Create a heatmap for time-series data
sns.heatmap(flights_pivot, cmap="coolwarm", annot=True, fmt="d")
plt.title("Heatmap of Flight Passengers Over Time")
plt.show()
```

3. Customizing Seaborn Plots

A. Changing Themes

Seaborn comes with built-in themes:

```
sns.set_theme(style="darkgrid")  # Options: darkgrid, whitegrid, dark, white, ticks
```

B. Custom Color Palettes

```
sns.set_palette("pastel")  # Other options: deep, muted, bright, dark, colorblind
```

4. Combining Multiple Plots with Seaborn

You can use Seaborn with matplotlib to create subplots.

```
fig, axes = plt.subplots(1, 2, figsize=(12, 5))

sns.boxplot(x="species", y="sepal_length", data=df, ax=axes[0])
axes[0].set_title("Box Plot")

sns.violinplot(x="species", y="sepal_length", data=df, ax=axes[1])
axes[1].set_title("Violin Plot")

plt.tight_layout()
plt.show()
```

Chapter 17
Machine Learning Basics

1.Introduction to Machine Learning

Machine Learning (ML) is a branch of Artificial Intelligence (AI) that enables computers to learn from data and make decisions or predictions without being explicitly programmed. It focuses on developing algorithms that can identify patterns, adapt to new data, and improve performance over time.

Why Machine Learning?

- Automation: Reduces human effort in repetitive tasks.
- Pattern Recognition: Identifies hidden insights from large datasets.
- Improved Decision-Making: Supports businesses, healthcare, and finance.
- Scalability: Handles complex problems efficiently.

Types of Machine Learning

1. Supervised Learning
   - The model learns from labeled data.
   - Examples: Spam detection, fraud detection, medical diagnosis.
   - Algorithms: Linear Regression, Decision Trees, Random Forest, Neural Networks.
2. Unsupervised Learning
   - The model identifies patterns in unlabeled data.
   - Examples: Customer segmentation, anomaly detection.
   - Algorithms: Clustering (K-Means, DBSCAN), Dimensionality Reduction (PCA).
3. Reinforcement Learning
   - The model learns by interacting with an environment.
   - Examples: Self-driving cars, game AI.
   - Algorithms: Q-Learning, Deep Q Networks (DQN).

Key Components of Machine Learning

- Dataset: Collection of data used to train the model.
- Features: Characteristics used for making predictions.
- Algorithm: The method used to train the model.

- Training & Testing: Evaluating model performance.
- Evaluation Metrics: Accuracy, Precision, Recall, F1-score.

Applications of Machine Learning

- Healthcare: Disease prediction, medical imaging.
- Finance: Credit scoring, fraud detection.
- E-commerce: Recommendation systems, customer behavior analysis.
- Autonomous Vehicles: Object detection, navigation.
- Cybersecurity: Threat detection, anomaly detection.

2.Using scikit-learn for ML Models

Scikit-learn is a powerful machine learning (ML) library in Python that provides tools for building and deploying ML models efficiently. Below is an overview of using scikit-learn for different ML tasks:

1. Installing Scikit-learn

pip install scikit-learn

2. Basic Workflow in Scikit-learn

The typical workflow for using scikit-learn includes:

1. Importing necessary libraries
2. Loading or generating a dataset
3. Splitting the dataset into training and testing sets
4. Selecting and training a model
5. Making predictions
6. Evaluating model performance

2. Basic Workflow in Scikit-learn

The typical workflow for using scikit-learn includes:

1. Importing necessary libraries
2. Loading or generating a dataset
3. Splitting the dataset into training and testing sets
4. Selecting and training a model
5. Making predictions
6. Evaluating model performance

3. Example: Training a Classification Model

Here's an example using the Iris dataset and a Support Vector Machine (SVM) classifier:

```
import numpy as np
import pandas as pd
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score

# Load dataset
iris = datasets.load_iris()
X, y = iris.data, iris.target

# Split into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Feature scaling (optional but recommended for SVM)
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Train the model
model = SVC(kernel='linear')  # Using a linear kernel
model.fit(X_train, y_train)

# Make predictions
y_pred = model.predict(X_test)
# Evaluate performance
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.2f}")
```

4. Example: Training a Regression Model

Let's use a Linear Regression model on the Boston Housing dataset:

```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error

# Load dataset
boston = datasets.load_boston()
X, y = boston.data, boston.target

# Split dataset
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Train model
regressor = LinearRegression()
regressor.fit(X_train, y_train)

# Predict
y_pred = regressor.predict(X_test)

# Evaluate
mse = mean_squared_error(y_test, y_pred)
print(f"Mean Squared Error: {mse:.2f}")
```

5. Commonly Used Scikit-learn Algorithms

- **Classification:** Logistic Regression, SVM, Decision Trees, Random Forest, Naive Bayes, K-Nearest Neighbors (KNN)
- **Regression:** Linear Regression, Ridge Regression, Lasso Regression, Decision Trees, Random Forest
- **Clustering:** K-Means, DBSCAN, Agglomerative Clustering
- **Dimensionality Reduction:** PCA, t-SNE

3.Supervised vs. Unsupervised Learning.

1. Supervised Learning

Supervised learning is a type of machine learning where the model is trained using labeled data. This means the input data has corresponding output labels, and the model learns the relationship between them.

✅ **Key Features:**

- **Uses labeled data (input-output pairs).**
- **The goal is to map inputs to correct outputs.**
- **Requires human intervention for data labeling.**
- **Common algorithms: Linear Regression, Logistic Regression, Decision Trees, Random Forest, Support Vector Machines (SVM), Neural Networks.**

✅ **Examples:**

- **Spam detection: Classify emails as "spam" or "not spam."**
- **Fraud detection: Identify fraudulent transactions in banking.**
- **Medical diagnosis: Predict disease based on patient data.**
-

2. Unsupervised Learning

Unsupervised learning is a type of machine learning where the model is trained on unlabeled data and finds hidden patterns without explicit guidance.

✅ **Key Features:**

- **Uses unlabeled data (no predefined output).**
- **The goal is to discover hidden patterns and relationships in data.**
- **No human intervention is needed for labeling.**
- **Common algorithms: K-Means Clustering, Hierarchical Clustering, Principal Component Analysis (PCA), Autoencoders, DBSCAN.**

✅ **Examples:**

- **Customer segmentation: Grouping customers based on purchasing behavior.**
- **Anomaly detection: Identifying unusual network activity for cybersecurity.**
- **Recommendation systems: Suggesting similar movies or products (e.g., Netflix, Amazon).**

4..Basic Model Training and Evaluation

Model training and evaluation are fundamental steps in machine learning. Here's a breakdown of the basic process:

1. Data Preparation

Before training a model, you need to prepare the dataset:

- Collect Data: Gather relevant data for the problem.
- Preprocess Data: Handle missing values, normalize/scale numerical features, and encode categorical features.
- Split Data: Divide the dataset into:
  - Training Set (typically 70-80%): Used to train the model.
  - Validation Set (optional, 10-15%): Used to tune hyperparameters.
  - Test Set (10-20%): Used for final evaluation.

2. Model Training

This step involves choosing and training a model:

- Select a Model: Choose an algorithm (e.g., Linear Regression, Decision Trees, Neural Networks).
- Train the Model: Fit the model on the training data.
- Tune Hyperparameters: Optimize parameters using techniques like Grid Search or Random Search.
-

Example in Python (using sklearn):

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.datasets import load_iris

# Load dataset
data = load_iris()
X_train, X_test, y_train, y_test = train_test_split(data.data, data.target,
test_size=0.2, random_state=42)

# Train model
model = LogisticRegression(max_iter=200)
model.fit(X_train, y_train)
```

4..Basic Model Training and Evaluation

Model training and evaluation are fundamental steps in machine learning. Here's a breakdown of the basic process:

1. Data Preparation

Before training a model, you need to prepare the dataset:

- Collect Data: Gather relevant data for the problem.
- Preprocess Data: Handle missing values, normalize/scale numerical features, and encode categorical features.
- Split Data: Divide the dataset into:
  - Training Set (typically 70-80%): Used to train the model.
  - Validation Set (optional, 10-15%): Used to tune hyperparameters.
  - Test Set (10-20%): Used for final evaluation.

2. Model Training

This step involves choosing and training a model:

- Select a Model: Choose an algorithm (e.g., Linear Regression, Decision Trees, Neural Networks).
- Train the Model: Fit the model on the training data.
- Tune Hyperparameters: Optimize parameters using techniques like Grid Search or Random Search.
-

Example in Python (using sklearn):

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.datasets import load_iris

# Load dataset
data = load_iris()
X_train, X_test, y_train, y_test = train_test_split(data.data, data.target,
test_size=0.2, random_state=42)

# Train model
model = LogisticRegression(max_iter=200)
model.fit(X_train, y_train)
```

3. Model Evaluation

Evaluate the trained model using various metrics:

- Accuracy: (Correct Predictions) / (Total Predictions)
- Precision, Recall, F1-score (for classification tasks)
- Mean Squared Error (MSE), R^2 Score (for regression tasks)

Example:

```
from sklearn.metrics import accuracy_score

# Predict on test data
y_pred = model.predict(X_test)

# Evaluate model
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.2f}")
```

4. Improving the Model

If the performance is not satisfactory, consider:

- Feature Engineering: Add, remove, or transform features.
- Hyperparameter Tuning: Use GridSearchCV, RandomizedSearchCV.
- Try Different Algorithms: Compare different models.
- Increase Training Data: More data can improve performance.

Chapter 18
Introduction to Web Development with Flask/Django

1.Understanding Web Frameworks

Understanding Web Frameworks

A web framework is a software tool that provides a structured way to develop and deploy web applications. It simplifies web development by offering pre-built components, reusable code, and standardized development patterns.

Types of Web Frameworks

1. Front-End Frameworks

- **Focus on the client side (what the user sees).**
- **Examples:**
  - **React.js – A JavaScript library for building UI components.**
  - **Angular – A TypeScript-based framework for dynamic web applications.**
  - **Vue.js – A progressive framework known for its simplicity and flexibility.**

2. Back-End Frameworks

- **Handle server-side logic, database interactions, and authentication.**
- **Examples:**
  - **Django (Python) – A high-level framework with built-in security and scalability.**
  - **Flask (Python) – A lightweight and flexible micro-framework.**
  - **Express.js (JavaScript) – A minimal and fast framework for Node.js applications.**
  - **Spring Boot (Java) – A powerful framework for enterprise-level applications.**

3. Full-Stack Frameworks

- **Combine both front-end and back-end functionalities.**
- **Examples:**
  - **Next.js (React + Node.js) – A React-based framework with server-side rendering.**
  - **Ruby on Rails (Ruby) – A convention-over-configuration framework.**
  - **Laravel (PHP) – A modern framework for building scalable PHP applications.**

2.Setting Up a Simple Flask/Django App

Here's a simple guide to setting up a basic web app using Flask and Django.
Flask Setup (Lightweight Python Web Framework)
Flask is minimalistic and great for small applications and APIs.

1. Install Flask

pip install flask

2. Create a Simple Flask App

Create a file app.py and add the following:

```
from flask import Flask

app = Flask(__name__)

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

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

Django Setup (Full-Featured Web Framework)

Django is best for large-scale applications requiring authentication, ORM, and admin dashboards.

1. Install Django

pip install django

2. Create a Django Project

```
django-admin startproject myproject
cd myproject
```

3. Start the Django Development Server

python manage.py runserver

3.Handling Routes and Templates

Handling routes and templates is a key aspect of building web applications. If you're working with Flask (Python), Express.js (JavaScript/Node.js), or Django (Python), here's how routing and templating are managed in these frameworks:

Flask (Python)

Flask uses the render_template() function to render HTML templates stored in the templates/ folder.

Setting up Routes

```
from flask import Flask, render_template

app = Flask(__name__)

@app.route('/')
def home():
    return render_template('index.html')  # Renders index.html

@app.route('/about')
def about():
    return render_template('about.html')  # Renders about.html
if __name__ == '__main__':
    app.run(debug=True)
```

Template Example (templates/index.html)

```
<!DOCTYPE html>
<html>
<head>
    <title>Home Page</title>
</head>
<body>
    <h1>Welcome to Flask</h1>
</body>
</html>
```

Express.js (Node.js)

Express uses the res.render() function along with Pug, EJS, or Handlebars as the templating engine.

Setting up Express Routes

```
const express = require('express');
const app = express();

app.set('view engine', 'ejs'); // Using EJS for templates
app.set('views', './views');   // Templates stored in 'views' folder

app.get('/', (req, res) => {
   res.render('index', { title: 'Home Page' });
});

app.get('/about', (req, res) => {
   res.render('about', { title: 'About Page' });
});

app.listen(3000, () => {
   console.log("Server running on port 3000");
});
```

Django (Python)

Django follows the MTV (Model-Template-View) pattern.

Setting up Django Routes

In views.py:

```
from django.shortcuts import render

def home(request):
   return render(request, 'index.html')

def about(request):
   return render(request, 'about.html')
```

4.Working with Forms and Databases

Working with forms and databases is a fundamental part of web development, allowing users to input and store data efficiently. Here's an overview of how it works:

1. Creating a Form

A form is used to collect user input. This is typically done using HTML.

Example (HTML Form):

```
<form action="process.php" method="POST">
  <label for="name">Name:</label>
  <input type="text" id="name" name="name" required>

  <label for="email">Email:</label>
  <input type="email" id="email" name="email" required>

  <button type="submit">Submit</button>
</form>
```

2. Handling Form Submission

When the user submits the form, the data is sent to the backend (e.g., PHP, Python, Node.js).

Example (PHP Backend):

```
<?php
if ($_SERVER["REQUEST_METHOD"] == "POST") {
    $name = $_POST["name"];
    $email = $_POST["email"];

    // Database connection
    $conn = new mysqli("localhost", "root", "", "mydatabase");

    if ($conn->connect_error) {
        die("Connection failed: " . $conn->connect_error);
    }
```

```
$sql = "INSERT INTO users (name, email) VALUES ('$name', '$email')";

if ($conn->query($sql) === TRUE) {
echo "Record added successfully!";
} else {
echo "Error: " . $sql . "<br>" . $conn->error;
}

$conn->close();
}
?>
```

3. Fetching and Displaying Data

You can retrieve data from the database and display it on a webpage.

Example (PHP Fetch Data):

```
$conn = new mysqli("localhost", "root", "", "mydatabase");

$sql = "SELECT * FROM users";
$result = $conn->query($sql);

while ($row = $result->fetch_assoc()) {
   echo "Name: " . $row["name"] . " - Email: " . $row["email"] . "<br>";
}

$conn->close();
```

4. Securing Your Form (Validation & Security)

- Use prepared statements to prevent SQL injection.
- Validate user input (e.g., check for valid email format).
- Use server-side and client-side validation to improve security.

Chapter 19
Testing and Debugging in Python

1.Writing Unit Tests with unittest

Unit testing is essential for ensuring the reliability of your code. In Python, the built-in unittest module provides a framework for writing and running unit tests. Below is a step-by-step guide on writing unit tests using unittest.

1. Importing unittest

The unittest module is included in Python's standard library, so no extra installation is needed.

```
import unittest
```

2. Creating a Sample Function to Test

Let's say we have a simple function in math_operations.py that performs basic arithmetic:

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

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

def multiply(x, y):
    return x * y

def divide(x, y):
    if y == 0:
        raise ValueError("Cannot divide by zero")
    return x / y
```

3. Writing Unit Tests

We create a separate test file, e.g., test_math_operations.py, to test the functions:

```
import unittest
from math_operations import add, subtract, multiply, divide

class TestMathOperations(unittest.TestCase):

    def test_add(self):
        self.assertEqual(add(3, 5), 8)
        self.assertEqual(add(-1, 1), 0)
        self.assertEqual(add(0, 0), 0)

    def test_subtract(self):
        self.assertEqual(subtract(10, 5), 5)
        self.assertEqual(subtract(-1, -1), 0)
        self.assertEqual(subtract(0, 5), -5)

    def test_multiply(self):
        self.assertEqual(multiply(2, 3), 6)
        self.assertEqual(multiply(-2, 3), -6)
        self.assertEqual(multiply(0, 5), 0)

    def test_divide(self):
        self.assertEqual(divide(10, 2), 5)
        self.assertEqual(divide(-10, 2), -5)
        self.assertRaises(ValueError, divide, 10, 0)

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

2.Debugging with pdb

pdb (Python Debugger) is a built-in Python module used for debugging code by allowing you to set breakpoints, inspect variables, and step through execution.

Basic Commands in pdb

1. **Starting pdb**

Run a script with pdb:

python -m pdb script.py

2.Navigation Commands

- **c (continue) → Resume execution until the next breakpoint.**
- **n (next) → Execute the next line of code.**
- **s (step) → Step into function calls.**
- **q (quit) → Exit the debugger.**
- **r (return) → Run until the function returns.**

3.Variable Inspection

- **p var → Print the value of var.**
- **pp var → Pretty-print var (useful for dictionaries/lists).**
- **whos → List all variables in scope.**

4.Breakpoints

- **b 10 → Set a breakpoint at line 10.**
- **b my_function → Break at the start of my_function.**
- **cl → Clear all breakpoints.**

5.Stack Navigation

  - **l → List source code around the current line.**
  - **w → Show the current stack frame.**
  - **up / down → Move up/down the call stack.**

Example Debugging Session

```
def add(a, b):
    result = a + b
    return result
```

```
def main():
 x, y = 5, 10
 import pdb; pdb.set_trace() # Breakpoint
 sum_result = add(x, y)
 print(f"Sum: {sum_result}")

if __name__ == "__main__":
 main()
```

3.Common Python Errors and Fixes

Here are some common Python errors and their fixes:

1. SyntaxError: invalid syntax

Cause: This occurs when Python encounters incorrect syntax.

Fix:

```
print("Hello, World!")  # Use parentheses in Python 3
```

2. NameError: name 'x' is not defined

Cause: Trying to use a variable before defining it.

```
print(x)  # x is not defined before use
```

Fix:

```
x = 10
print(x)
```

3. TypeError: unsupported operand type(s)

Cause: Occurs when incompatible data types are used together.

```
num = 5
text = "hello"
print(num + text)  # Cannot add int and str
```

Fix:

```
print(str(num) + text)  # Convert int to string before concatenation
```

4. IndexError: list index out of range

Cause: Accessing an index that does not exist in the list.

```
my_list = [1, 2, 3]
print(my_list[5])  # Index 5 is out of range
```

Fix:

```
if len(my_list) > 5:
    print(my_list[5])
else:
    print("Index out of range")
```

5. KeyError: 'key'

Cause: Trying to access a key in a dictionary that doesn't exist.

```
data = {"name": "Alice"}
print(data["age"])  # 'age' key is missing
```

Fix:

```
print(data.get("age", "Key not found"))  # Use .get() method with default value
```

6. AttributeError: 'NoneType' object has no attribute 'x'

Cause: Trying to access an attribute on a None object.

```
x = None
print(x.upper())  # x is None, has no upper() method
```

Fix:

```
if x is not None:
    print(x.upper())
else:
    print("x is None")
```

Chapter 20
Advanced Python Concepts

1.Decorators and Generators

Decorators and Generators in Python

Both decorators and generators are advanced Python features that help in making code more readable, efficient, and reusable. Let's go through each concept in detail.

1. Generators

Generators are special types of iterators that allow you to iterate over data without storing the entire dataset in memory. They are defined using the yield keyword.

Why Use Generators?

- **Memory Efficient: They generate values on the fly instead of storing everything in memory.**
- **Lazy Evaluation: They compute values as needed, making them useful for large datasets.**
- **State Retention: Unlike normal functions, generators retain their state between function calls.**
-

Example: A Simple Generator

```
def count_up_to(n):
    count = 1
    while count <= n:
        yield count  # Produces a value and pauses execution
        count += 1

gen = count_up_to(5)
print(next(gen))  # Output: 1
print(next(gen))  # Output: 2
```

Generator Expression (Like List Comprehension)

You can create generators using generator expressions, similar to list comprehensions but with parentheses () instead of [].

```
squares = (x * x for x in range(5))
print(next(squares))  # Output: 0
print(next(squares))  # Output: 1
```

2. Decorators

A decorator is a function that takes another function as input and extends its functionality without modifying its actual code.

Why Use Decorators?

- Code Reusability: Helps avoid redundant code.
- Separation of Concerns: Keeps logic separate from enhancements.
- Useful for Logging, Authorization, Timing Functions, etc.

Basic Decorator Example

```
def decorator_func(original_func):
    def wrapper():
        print("Wrapper executed before", original_func.__name__)
        return original_func()
    return wrapper

@decorator_func  # Applying the decorator
def say_hello():
    print("Hello!")

say_hello()
```

Output:

```
Wrapper executed before say_hello
Hello!
```

2.Multi-threading and Parallel Processing

Multi-threading vs. Parallel Processing

Both multi-threading and parallel processing are techniques used to improve the performance of applications by executing multiple tasks simultaneously. However, they serve different purposes and work in different ways.

1. Multi-threading

Multi-threading is a programming technique that allows multiple threads (smaller units of a process) to execute concurrently within a single process. It is mainly used to achieve concurrency, not necessarily parallel execution.

Key Characteristics:

- **Multiple threads share the same memory space.**
- **Useful for I/O-bound tasks (e.g., downloading files, web scraping, GUI applications).**
- **Uses context switching, meaning that only one thread executes at a time in single-core CPUs.**
- **Threads communicate via shared memory.**

Example Use Cases:

- **Web browsers (handling multiple tabs)**
- **GUI applications (keeping UI responsive while processing)**
- **Asynchronous I/O operations (reading files, sending requests)**

Example (Python Multi-threading)

```
import threading

def print_numbers():
    for i in range(5):
        print(i)

# Creating two threads
thread1 = threading.Thread(target=print_numbers)
thread2 = threading.Thread(target=print_numbers)
```

```
# Starting threads
thread1.start()
thread2.start()
# Waiting for threads to finish
thread1.join()
thread2.join()
print("Threads finished execution")
```

2. Parallel Processing

Parallel processing involves dividing a task into smaller subtasks and executing them simultaneously on multiple CPU cores or even multiple machines. It is primarily used for CPU-bound tasks.

Key Characteristics:

- True parallel execution on multi-core processors.
- Independent processes (not just threads) with separate memory spaces.
- Best suited for CPU-intensive tasks like scientific computations, AI model training, and large data processing.

Example Use Cases:

- Machine learning model training
- Large-scale data processing (Pandas, NumPy)
- Scientific simulations
-

Example

(Python Multiprocessing)

```
import multiprocessing
def square(n):
    return n * n

if __name__ == "__main__":
    numbers = [1, 2, 3, 4, 5]
    with multiprocessing.Pool(processes=2) as pool:
        results = pool.map(square, numbers)
    print("Squared numbers:", results)
```

3.Working with Virtual Environments

Working with virtual environments is essential in Python development, as it helps isolate dependencies and avoid conflicts between different projects. Here's a quick guide:

1. Creating a Virtual Environment

To create a virtual environment, use the following command:

```
python -m venv myenv
```

2. Activating the Virtual Environment

- **On Windows**

```
myenv\Scripts\activate
```

On macOS/Linux:

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

3. Installing Packages

Once the virtual environment is activated, you can install packages using pip:

```
pip install package_name
```

For example:

```
pip install numpy pandas
```

4. Deactivating the Virtual Environment

To exit the virtual environment, simply run:

```
deactivate
```

5. Removing the Virtual Environment

If you want to delete the environment, deactivate it first and then remove the directory:

```
rm -rf myenv  # macOS/Linux
rd /s /q myenv  # Windows (Command Prompt)
```

4.Understanding Python's Memory Management

Understanding Python's Memory Management

Python's memory management is a crucial aspect of how the language handles variables, objects, and memory allocation. It ensures efficient use of memory through automatic garbage collection and dynamic memory allocation.

1. Memory Management in Python

Python uses dynamic memory allocation, meaning memory is allocated to variables and objects at runtime. The Python memory manager handles:

- **Object allocation and deallocation**
- **Memory reuse**
- **Garbage collection (GC)**

2. Memory Allocation in Python

Python's memory allocation is divided into three main parts:

a) Stack Memory

- **Stores function calls, local variables, and control flow.**
- **Memory is allocated and deallocated automatically when functions execute.**
- **Works in a Last-In-First-Out (LIFO) manner**

b) Heap Memory

- **Stores objects and dynamically allocated variables.**
- **Managed by Python's built-in memory manager.**
- **Objects are referenced using pointers.**

c) Python Memory Pool (Private Heap)

- **Python maintains a private heap for memory management.**
- **The PyMalloc allocator manages small objects.**
- **Large objects are directly handled by the system's memory allocator.**
- **3. Garbage Collection in Python**
- **Python automatically frees memory that is no longer in use using Garbage Collection (GC). It uses:**
- **Reference Counting: Each object has a reference count, and when it reaches zero, the memory is freed.**
- **Cyclic Garbage Collection: Detects and removes cyclic references where**

- objects reference each other.
- Manually Triggering Garbage Collection
- You can manually run garbage collection using:

```
import gc
gc.collect()  # Force garbage collection
```

Chapter 21
Python Project Ideas and Best Practices

1.Best Coding Practices and Style Guide (PEP 8)

Best Coding Practices and Style Guide (PEP 8)

PEP 8 (Python Enhancement Proposal 8) is the official style guide for writing Python code. Following PEP 8 ensures code readability, maintainability, and consistency across projects. Below are the key principles of PEP 8 and other best coding practices.

1. Code Layout & Formatting

Indentation

- **Use 4 spaces per indentation level.**
- **Do not use tabs (use spaces instead).**

```
def function():
 if True:
print("Follow PEP 8")
```

Maximum Line Length

- **Keep lines ≤ 79 characters (for general code).**
- **Keep docstrings and comments ≤ 72 characters.**

Blank Lines

- **Use two blank lines between top-level functions and classes.**
- **Use one blank line inside a class between methods.**

```
class MyClass:

   def method_one(self):
      pass

   def method_two(self):
      pass
```

2.Project Ideas for Beginners, Intermediate, and Advanced Levels

Here are some project ideas categorized by skill level:

Beginner Level

Ideal for those who are just starting and want to practice basic programming skills.

1. To-Do List App – A simple task manager with CRUD operations.
2. Calculator – A basic arithmetic calculator with a GUI.
3. Weather App – Fetch weather data from an API and display it.
4. Number Guessing Game – A fun terminal-based guessing game.
5. Digital Clock – A real-time clock using Python Tkinter or JavaScript.
6. Unit Converter – Convert temperature, weight, or currency.
7. Personal Portfolio Website – Showcase your skills and projects.
8. Simple Blog – Allow users to write, edit, and delete posts.
9. Random Password Generator – Generate strong passwords.
10. Notes App – Save and retrieve short notes with a simple UI.

Intermediate Level

For those who have some experience and want to build more interactive applications.

1. Expense Tracker – Track daily expenses and generate reports.
2. E-commerce Website – Basic cart functionality using HTML, CSS, JavaScript, and backend tech.
3. Chat Application – Real-time messaging using WebSockets.
4. Library Management System – Manage books, borrowers, and returns.
5. Quiz App – A multiple-choice quiz with a leaderboard.
6. Portfolio Website with CMS – A dynamic portfolio using React/Flask.
7. Social Media Dashboard – View analytics for social media engagement.
8. Image Recognition App – Identify objects in images using AI.
9. Fitness Tracker – Track workouts and diet progress.
10. Stock Price Prediction – Use machine learning to predict stock trends.

Advanced Level

For experienced developers looking for a challenge.

1. AI-Powered Chatbot – A chatbot using NLP like GPT or Rasa.
2. Face Recognition Attendance System – Detect faces for automated

3.Blockchain-Based Voting System – Secure online voting using blockchain.
4.Automated Resume Screening System – Filter resumes using AI.
5.AI-Based Code Generator – Generate code from text descriptions.
6.Virtual Assistant (like Jarvis) – AI assistant for voice commands.
7.Self-Driving Car Simulation – Use reinforcement learning to navigate.
8.Cybersecurity Threat Detection System – Detect cyber threats using ML.
9.AI-Powered Content Generator – Generate articles based on keywords.
10.Smart Home Automation – IoT-based home automation with AI integration.

3.Contributing to Open Source and GitHub

Contributing to open-source projects on GitHub is a great way to improve your coding skills, collaborate with developers, and build a strong portfolio. Here's how you can get started:

1. Understanding Open Source

Open-source software is publicly available code that anyone can use, modify, and distribute. Popular open-source projects include Linux, React, TensorFlow, and VS Code.

2. Setting Up GitHub

- Create a GitHub account if you don't have one.
- Install Git on your system.
- Configure Git with:

```
git config --global user.name "Your Name"
git config --global user.email "your_email@example.com"
```

3. Finding a Project to Contribute To

- Explore repositories on GitHub Explore.
- Use platforms like Good First Issues or Up For Grabs.
- Look for projects with labels like good first issue, help wanted, or documentation.

4. Forking and Cloning

- Fork the repository (creates a copy under your GitHub account).
- Clone it locally:

9. Creating a Pull Request (PR)

- Go to the original repository on GitHub.
- Click New Pull Request and select your branch.
- Write a clear description of your changes and submit the PR.

10. Engaging with the Community

- Respond to feedback from maintainers.
- Review other PRs to learn from peers.
- Join discussions and contribute to issues.

Bonus Tips

- Follow project guidelines (contribution guidelines, code of conduct).
- Keep your fork updated with:

git fetch upstream
git merge upstream/main

4.Python Career Path: What's Next?

Since you already have strong experience in Python, Full Stack Development, Data Science, Cybersecurity, and AI/ML, your next steps depend on your career goals. Here are some options you can explore based on your expertise:

🔹 Advanced Specialization

If you want to deepen your Python expertise, consider specializing in:

1. AI & Machine Learning 🧠
   - Learn Deep Learning (TensorFlow, PyTorch)
   - Work on NLP, Computer Vision, or Generative AI
   - Participate in Kaggle Competitions
2. Data Engineering 📊
   - Learn Apache Spark, Hadoop, Airflow
   - Work with Big Data Pipelines
   - Master SQL + NoSQL Databases
3. Cybersecurity with Python 🔐
   - Learn Ethical Hacking, Automation, and Threat Detection
   - Work on Penetration Testing & Security Tools
4. Automation & DevOps ⚙️
   - Learn Docker, Kubernetes, Terraform

- Work with CI/CD & Infrastructure as Code

🔹 Career Advancement

Since you have experience in mentorship, you can aim for leadership roles:

- Senior Developer → Team Lead
- Technical Architect → CTO Role (if you plan a startup)
- AI/ML Engineer → Researcher/Scientist

🔹 Freelancing & Side Projects

- Start freelancing on Upwork, Fiverr, Toptal
- Develop your own SaaS Product or AI Tool
- Contribute to Open Source Projects on GitHub

🔹 Certifications & Higher Studies

- AWS Certified Solutions Architect (For Cloud Computing)
- Google TensorFlow Developer Certificate (For AI/ML)
- CEH (Certified Ethical Hacker) (For Cybersecurity)
- Master's in Data Science or AI (If interested in research)
-

🔹 Hackathons & Competitions

Since you've done well in hackathons, continue participating in:

- Google Hash Code, Facebook Hackathon, or Codeforces Contests
- Build AI/ML models for real-world problems

This Python material is for reference to gain basic knowledge about python ; don't rely solely on it, and also refer to other internet resources for competitive exams. Thank you from CodTech.