Name : **Daraniya Neel**

Advance Python Programming

1. Printing on Screen

➢ Introduction to the print() function in Python.

- The print() function in Python is one of the most commonly used functions and serves as a basic tool for outputting information to the console.
- The print() function is used to display text or other output on the screen, making it useful for debugging, presenting results, or interacting with users in a terminal environment.

- Basic Syntax

  print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)

- Parameters

  `*objects`:

  The objects to be printed. Multiple objects can be separated by commas.

  Example: print("Hello", "World") outputs Hello World.

  `sep` (Optional):

  Specifies the string inserted between objects.

  Default: ' '.

  Example: print("Python", "is", "fun", sep='-') outputs Python-is-fun.

  `end` (Optional):

  Specifies the string appended at the end of the output.

  Default: '\n' (new line).

  Example: print("Hello", end='!') outputs Hello!.

  `file` (Optional):

  Specifies the output stream. Default is sys.stdout.

  Example: Redirecting output to a file:

```
with open('output.txt', 'w') as f:
    print("Hello, File!", file=f)
```

  `flush` (Optional):

  If True, flushes the output buffer immediately.

  Useful in real-time output situations.

➢ Formatting outputs using f-strings and format().

- Python provides powerful tools for formatting strings.
- Two of the most commonly used methods are f-strings and the format() method.

- Using f-strings

  Introduced in Python 3.6, f-strings allow you to embed expressions directly within string literals, using curly braces {}.

  Syntax:

  ```
  f"string with {expression}"
  ```

  Variable Insertion:

  ```
  name = "Alice"
  age = 25
  print(f"{name} is {age} years old.")
  ```

  Expressions:

  ```
  a, b = 5, 3

  print(f"The sum of {a} and {b} is {a + b}.")
  ```

- Using the format() Method

  The format() method is a versatile and widely supported string formatting tool.

  Syntax:

  ```
  "string with {} placeholders".format(values)
  ```

  Basic Replacement:

  ```
  name = "Bob"
  age = 30
  print("{} is {} years old.".format(name, age))
  ```

  Positional Arguments:

  ```
  print("{1} scored {0} in the exam.".format(95, "Charlie"))
  ```

2. Reading Data from Keyboard

➢ Using the input() function to read user input from the keyboard.

- The input() function in Python is used to read user input from the keyboard.
- It pauses the program's execution until the user types something and presses Enter.

- Syntax:

  input(prompt)

  `prompt` (Optional):

  A string displayed to the user before input is collected.

  If not provided, the function simply waits for user input.

- Basic Usage:

```
name = input("What is your name? ")
print(f"Hello, {name}!")
```

- Working with Data Types:

  The input() function always returns user input as a string.

  You may need to convert it to the desired type, such as an integer or a float, using type casting.

```
age = input("Enter your age: ")
age = int(age)  # Convert the string to an integer
print(f"You are {age} years old.")

height = input("Enter your height in meters: ")

height = float(height)  # Convert the string to a float

print(f"Your height is {height} meters.")
```

Converting user input into different data types (e.g., int, float, etc.).

- The input() function in Python always returns the user input as a string.
- To use the input as another data type, you need to explicitly convert it using Python's type conversion functions, such as int(), float(), list(), etc.

- Type Conversion Basics
  - The most commonly used functions for type conversion are:

    int(): Converts to an integer.

    float(): Converts to a floating-point number.

    str(): Converts to a string (useful when converting numbers back to strings for display)

list(): Converts a string to a list of characters or items split by a delimiter.

- Converting Input to Integer

```
age = input("Enter your age: ")  # User enters: 25
age = int(age)  # Convert to integer
print(f"You are {age} years old.")
```

- Converting Input to Float

```
price = input("Enter the price of the item: ")  # User enters: 12.99
price = float(price)  # Convert to float
print(f"The price is ${price:.2f}.")
```

- Converting Input to Boolean

```
is_member = input("Are you a member? (yes/no): ")  # User enters: yes
is_member = is_member.lower() in ['yes', 'y']  # Convert to boolean
print(f"Membership status: {is_member}")
```

3. Opening and Closing Files

- Opening files in different modes ('r', 'w', 'a', 'r+', 'w+').

- In Python, the open() function is used to open files, and the file mode determines how the file is accessed (read, write, append, etc.).

- Basic Syntax:

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

`filename`: The name (and path, if needed) of the file.
`mode`: The mode in which the file is opened (e.g., `'r'`, `'w'`, etc.).

- Read Mode ('r')

Used to read content from a file.
Raises a FileNotFoundError if the file doesn't exist.

```
with open("example.txt", "r") as file:
    content = file.read()
    print(content)
```

- Write Mode ('w')

  Used to write to a file.

  Overwrites the file if it exists; creates a new file if it doesn't.

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

- Append Mode ('a')

  Used to append content to the end of the file.

  Creates the file if it doesn't exist.

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

- Read and Write Mode ('r+')

  Allows both reading and writing.

  The file must exist; otherwise, an error is raised.

```
with open("example.txt", "r+") as file:
    content = file.read()
    print("Current content:", content)
    file.write("\nAdding more data.")
```

- Write and Read Mode ('w+')

  Allows both writing and reading.

  Overwrites the file if it exists; creates a new file if it doesn't.

```
with open("example.txt", "w+") as file:
    file.write("Writing new content.")
    file.seek(0)  # Move the cursor to the beginning to read
    content = file.read()
    print("Content after writing:", content)
```

- Append and Read Mode ('a+')

  Allows both appending and reading.

  Creates the file if it doesn't exist.

```
with open("example.txt", "a+") as file:
    file.write("\nAppended with a+ mode.")
    file.seek(0)  # Move the cursor to the beginning to read
    content = file.read()
```

```
print("Content after appending:", content)
```

➢ Using the open() function to create and access files.

- The open() function in Python is used to create, read, and write files.
- It provides a simple interface to interact with files in different modes.

- Syntax of open()

```
file_object = open(filename, mode)
```

`filename`: Name (and path, if applicable) of the file.
`mode`: Specifies the purpose of opening the file (e.g., reading, writing, appending).

- Creating a File

To create a new file, use the following modes:

'w' (Write mode): Creates a new file or overwrites an existing one.
'x' (Exclusive creation): Creates a file but raises an error if the file already exists.
'a' (Append mode): Creates the file if it doesn't exist.

- Example:

```
with open("newfile.txt", "w") as file:
        file.write("This is a newly created file.")
        print("File created successfully.")
```

➢ Closing files using close().

- In Python, when a file is opened using the open() function, it's crucial to close it after performing all operations. This is done using the close() method.
- Closing a file ensures that all changes are saved (if the file was opened in write mode) and that system resources associated with the file are released.

- Using close() Method
  - Basic Syntax

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

- Example: Writing and Closing a File

```
file = open("example.txt", "w")
file.write("This is a test.")
file.close()
```

4. Reading and Writing Files

➢ Reading from a file using read(), readline(), readlines().

➢ read()

This method reads the entire content of the file as a single string.
It is useful when you need all the data in one go.
However, it may not be memory-efficient for large files.

```
# Example using read()
with open("example.txt", "r") as file:
    content = file.read()
    print(content)
```

➢ readline()

This method reads a single line from the file.
It's useful for processing a file line by line without loading the entire file into memory.

```
# Example using readline()
with open("example.txt", "r") as file:
    line = file.readline()
    while line:
        print(line.strip())  # Strip removes trailing newline
        line = file.readline()
```

➢ readlines()

This method reads all lines of a file and returns them as a list of strings.
Each string corresponds to a line in the file.

```
with open("example.txt", "r") as file:
    lines = file.readlines()
    for line in lines:
        print(line.strip())
```

- Writing to a file using write() and writelines().

- write()

  The write() method writes a single string to the file.

  It is used when you want to add text to the file, either in one go or incrementally.

  Example:

```
with open("example.txt", "w") as file:
    file.write("Hello, World!\n")
    file.write("This is a new line.\n")
```

- writelines()

  The writelines() method writes a list of strings to the file.

  Each string in the list is written as-is, so you must include newline characters in the strings if you want separate lines.

  Example:

```
lines = ["First line\n", "Second line\n", "Third line\n"]

with open("example.txt", "w") as file:
    file.writelines(lines)
```

5. Exception Handling

- Introduction to exceptions and how to handle them using try, except, and finally.

- An exception is an error that occurs during the execution of a program.
- When an error occurs, Python stops the normal flow of the program and raises an exception.

- Examples of common exceptions include:

  `ZeroDivisionError`: Division by zero.

  `FileNotFoundError`: Attempting to open a file that does not exist.

  `ValueError`: Invalid argument passed to a function.

  `TypeError`: Invalid operation on data types.

- try and except :

The try block contains code that might raise an exception.
If an exception occurs, the except block is executed, allowing you to handle the error gracefully.

Example:

```
try:
    num = int(input("Enter a number: "))
    print(f"The number is {num}")
except ValueError:
    print("Invalid input! Please enter a valid number.")
```

- Handling Multiple Exceptions:

You can handle multiple exceptions by specifying different except blocks.

Example:

```
try:
    a = int(input("Enter numerator: "))
    b = int(input("Enter denominator: "))
    result = a / b
    print(f"The result is {result}")
except ZeroDivisionError:
    print("Error: Cannot divide by zero.")
except ValueError:
    print("Error: Please enter numeric values.")
```

- finally Block:

The finally block is always executed, regardless of whether an exception was raised or not.
It is typically used for cleanup operations (e.g., closing a file or releasing resources).

Example:

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

➢ Understanding multiple exceptions and custom exceptions.

➢ Handling Multiple Exceptions with Separate except Blocks

You can define separate except blocks to handle each exception type individually.

Example:

```
try:
    num = int(input("Enter numerator: "))
    denom = int(input("Enter denominator: "))
    result = num / denom
    print(f"Result: {result}")
except ZeroDivisionError:
    print("Error: Cannot divide by zero.")
except ValueError:
    print("Error: Invalid input. Please enter numeric values.")
```

➢ Handling Multiple Exceptions in a Single except Block

To handle multiple exceptions with a single block, use a tuple to group the exceptions.

Example:

```
try:
    num = int(input("Enter numerator: "))
    denom = int(input("Enter denominator: "))
    result = num / denom
    print(f"Result: {result}")
except (ZeroDivisionError, ValueError) as e:
    print(f"An error occurred: {e}")
```

➢ Custom Exceptions

- Defining a Custom Exception

  You can create a custom exception by subclassing Python's built-in Exception class.

  Example:

```
class CustomError(Exception):
    """A custom exception class"""
    Pass
```

- Raising a Custom Exception

Use the raise keyword to raise your custom exception.

Example:

```
class NegativeValueError(Exception):
    """Exception raised for negative values."""
def __init__(self, value):
    self.value = value
    self.message = f"Invalid value: {value}. Value must be non-
    negative."
    super().__init__(self.message)

# Function that raises the exception
def check_value(value):
    if value < 0:
        raise NegativeValueError(value)
        print(f"Value is valid: {value}")

try:
    check_value(-5)
except NegativeValueError as e:
    print(e)
```

6. Class and Object (OOP Concepts)

- Understanding the concepts of classes, objects, attributes, and methods in Python.

- Classes

  A class is a blueprint for creating objects.
  It defines the properties (attributes) and behaviors (methods) that the objects created from the class will have.

  Example:

```
class Person:
    # Class definition
    Pass
```

- Objects

An object is an instance of a class.
It represents a specific entity created from the class blueprint.

Example:

```
person1 = Person()  # Creating an object of the Person class
person2 = Person()  # Another object of the same class
print(type(person1))
```

- Attributes

Attributes are variables that store data about an object.
They represent the properties of the object.
In Python, attributes are defined in the class and are accessed using the dot (.) notation.

Example:

```
class Person:
def __init__(self, name, age):
        self.name = name  # Attribute
        self.age = age    # Attribute

# Creating objects
person1 = Person("Alice", 25)
person2 = Person("Bob", 30)

# Accessing attributes
print(person1.name)  # Output: Alice
print(person2.age)   # Output: 30
```

- Methods

Methods are functions defined inside a class that describe the behaviors of an object.
They are similar to functions but are associated with objects.

Example:

```
class Person:
        def __init__(self, name, age):
                self.name = name
                self.age = age

        def greet(self):  # Method
```

```
        return f"Hello, my name is {self.name} and I am {self.age} years old."

    # Creating an object
    person1 = Person("Alice", 25)

    # Calling a method
    print(person1.greet())  # Output: Hello, my name is Alice and I am 25 years
    old.
```

- Difference between local and global variables.

Local Variable	Global Variable
Declared inside a function.	Declared outside all functions.
Accessible only within the defining function.	Accessible throughout the program.
Created when the function starts, destroyed when it ends.	Exists throughout the program's execution.
Does not require a special keyword.	Requires the global keyword to modify inside a function.
Cannot be accessed outside its function.	Can be accessed from any part of the program.

7. Inheritance

- Single, Multilevel, Multiple, Hierarchical, and Hybrid inheritance in Python.

- Single Inheritance

  In single inheritance, a child class inherits from only one parent class.

  Example:

```
class Parent:
        def display(self):
                print("This is the Parent class.")

class Child(Parent):
        def show(self):
                print("This is the Child class.")
```

```
# Creating objects
child = Child()
child.display()  # Accessing Parent class method
child.show()     # Accessing Child class method
```

- Multilevel Inheritance

  In multilevel inheritance, a child class inherits from a parent class, and then another class inherits from the child class, forming a chain.

  Example:

```
class Grandparent:
    def display_grandparent(self):
        print("This is the Grandparent class.")

class Parent(Grandparent):
    def display_parent(self):
        print("This is the Parent class.")

class Child(Parent):
    def display_child(self):
        print("This is the Child class.")

# Creating objects
child = Child()
child.display_grandparent()  # Accessing Grandparent class method
child.display_parent()       # Accessing Parent class method
child.display_child()        # Accessing Child class method
```

- Multiple Inheritance

  In multiple inheritance, a child class inherits from more than one parent class. Python resolves conflicts using the Method Resolution Order (MRO).

  Example:

```
class Parent1:
    def feature1(self):
        print("Feature 1 from Parent1.")

class Parent2:
    def feature2(self):
        print("Feature 2 from Parent2.")
```

```
class Child(Parent1, Parent2):
    def feature3(self):
        print("Feature 3 from Child.")

# Creating objects
child = Child()
child.feature1()  # Accessing Parent1 class method
child.feature2()  # Accessing Parent2 class method
child.feature3()  # Accessing Child class method
```

- Hierarchical Inheritance

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

  Example:

```
class Parent:
    def common_feature(self):
        print("This is a feature of the Parent class.")

class Child1(Parent):
    def feature_child1(self):
        print("Feature specific to Child1.")

class Child2(Parent):
    def feature_child2(self):
        print("Feature specific to Child2.")

# Creating objects
child1 = Child1()
child2 = Child2()

child1.common_feature()  # Accessing Parent class method
child1.feature_child1()  # Accessing Child1 method
child2.common_feature()  # Accessing Parent class method
child2.feature_child2()  # Accessing Child2 method
```

- Hybrid Inheritance

  Hybrid inheritance is a combination of two or more types of inheritance, such as hierarchical and multiple inheritance. It is used to model complex relationships.

  Example:

```
class Parent:
        def feature_parent(self):
                print("This is a feature of the Parent class.")

class Child1(Parent):  # Hierarchical Inheritance
        def feature_child1(self):
                print("Feature specific to Child1.")

class Child2(Parent):  # Hierarchical Inheritance
        def feature_child2(self):
                print("Feature specific to Child2.")

class SubChild(Child1, Child2):  # Multiple Inheritance
        def feature_subchild(self):
                print("Feature specific to SubChild.")

# Creating objects
subchild = SubChild()
subchild.feature_parent()  # Accessing Parent class method
subchild.feature_child1()  # Accessing Child1 method
subchild.feature_child2()  # Accessing Child2 method
subchild.feature_subchild() # Accessing SubChild method
```

- Using the super() function to access properties of the parent class.

- The super() function is a built-in Python method used to access methods or properties of a parent class from a child class.
- This is especially useful in cases of inheritance when you need to call a parent class's methods or override them while still maintaining access to their original functionality.

- Access Parent Methods or Attributes:
  You can use super() to call a parent class's methods without explicitly naming the class.

  Example:

```
class Parent:
        def display(self):
                print("This is the Parent class method.")
```

```
class Child(Parent):
    def display(self):
        print("This is the Child class method.")
        super().display()  # Accessing the Parent class method

# Creating an object of the Child class
child = Child()
child.display()
```

- Using super() in the Constructor

  You can use super() to call the parent class's constructor (__init__) to initialize attributes of the parent class in addition to the child class's attributes.

  Example:

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

class Child(Parent):
    def __init__(self, name, age):
        super().__init__(name)  # Calling Parent's constructor
        self.age = age

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

# Creating an object of the Child class
child = Child("Alice", 25)
child.display()
```

- super() in Multiple Inheritance

  In multiple inheritance, super() ensures that the methods are called in the correct order following the Method Resolution Order (MRO).

  Example:

```
class A:
    def display(self):
        print("Class A method")

class B(A):
    def display(self):
```

```
            print("Class B method")
            super().display()

class C(B):
    def display(self):
            print("Class C method")
            super().display()

# Creating an object of Class C
obj = C()
obj.display()
```

8. Method Overloading and Overriding

- Method overloading: defining multiple methods with the same name but different parameters.

- Method Overloading allows defining multiple methods in the same class with the same name but different numbers or types of parameters.
- While some languages like Java or C++ support method overloading natively, Python does not. However, you can simulate it by using default arguments, variable-length arguments (*args, **kwargs), or other programming techniques.

- Python Does Not Natively Support Method Overloading:
  If two methods in the same class have the same name, the second one will override the first.

- Simulating Overloading:
  Use default arguments or *args and **kwargs to handle varying numbers or types of parameters.

- Default Arguments to Simulate Method Overloading
  Using default arguments, you can handle different cases with a single method definition.

  Example:

```
class Calculator:
    def add(self, a, b=0, c=0):
```

```
                return a + b + c

        # Creating an object
        calc = Calculator()
        print(calc.add(5))          # Adds one number (5)
        print(calc.add(5, 10))      # Adds two numbers (5 + 10)
        print(calc.add(5, 10, 15))  # Adds three numbers (5 + 10 + 15)
```

- Using *args for Variable-Length Arguments

  The *args parameter allows handling a varying number of arguments.

  Example:

```
        class Calculator:
            def add(self, *args):
                return sum(args)

        # Creating an object
        calc = Calculator()
        print(calc.add(5))            # Adds one number (5)
        print(calc.add(5, 10))        # Adds two numbers (5 + 10)
        print(calc.add(5, 10, 15))    # Adds three numbers (5 + 10 + 15)
```

- Method overriding: redefining a parent class method in the child class.

- Method overriding is a feature in object-oriented programming that allows a subclass to redefine or provide its own implementation for a method already defined in its parent class.
- his is used when a child class wants to customize or replace the behavior of a method inherited from the parent class.

- Inheritance: Overriding occurs in a subclass that inherits from a parent class.
- Same Method Signature: The method in the child class must have the same name as the method in the parent class.
- Dynamic Dispatch: Python automatically determines which method to call based on the type of the object at runtime.
- super(): You can call the parent class's method explicitly using the super() function.

- Example

```
    class Parent:
```

```
    def display(self):
        print("Display method in Parent class.")

class Child(Parent):
    def display(self):
        print("Display method in Child class.")

# Example usage
parent_obj = Parent()
parent_obj.display()  # Output: Display method in Parent class.

child_obj = Child()
child_obj.display()   # Output: Display method in Child class.

# Polymorphism example
parent_ref = Child()
parent_ref.display()  # Output: Display method in Child class.
```

9. SQLite3 and PyMySQL (Database Connectors)

- Introduction to SQLite3 and PyMySQL for database connectivity.

- 1. SQLite3

- What is SQLite?

  SQLite is a lightweight, self-contained, serverless database engine. It is built into Python's standard library, so no additional installation is required.
  It is widely used for small to medium-sized applications, prototyping, and local storage.

- Features of SQLite3

  No server process; the database is a file on disk.
  Fast and efficient for read-heavy operations.
  ACID-compliant (ensures reliable transactions).
  Ideal for lightweight applications, testing, and local storage.

- Using SQLite3 in Python

  The sqlite3 module in Python allows for seamless interaction with SQLite databases.

```
import sqlite3

# Connect to (or create) a database
connection = sqlite3.connect("example.db")

# Create a cursor object to execute SQL queries
cursor = connection.cursor()

# Create a table
cursor.execute('''
   CREATE TABLE IF NOT EXISTS users (
      id INTEGER PRIMARY KEY,
      name TEXT NOT NULL,
      age INTEGER
   )
''')

# Insert data
cursor.execute("INSERT INTO users (name, age) VALUES (?, ?)", ("Alice", 30))

# Commit the changes
connection.commit()

# Retrieve data
cursor.execute("SELECT * FROM users")
rows = cursor.fetchall()

for row in rows:
   print(row)

# Close the connection
connection.close()
```

➢ 2. PyMySQL

➢ What is PyMySQL?

PyMySQL is a pure Python library used to connect to a MySQL or MariaDB database. It supports standard SQL queries and allows developers to perform CRUD operations on a MySQL database.

- Features of PyMySQL

  Connects to remote MySQL/MariaDB servers.

  Supports SSL connections for secure communication.

  Compatible with most MySQL features, such as stored procedures and transactions.

- Using PyMySQL in Python

  PyMySQL provides methods to connect to a MySQL database, execute queries, and fetch results.

```
import pymysql

# Connect to the MySQL database
connection = pymysql.connect(
    host='localhost',
    user='your_username',
    password='your_password',
    database='your_database'
)

try:
    # Create a cursor object
    cursor = connection.cursor()

    # Create a table
    cursor.execute('''
        CREATE TABLE IF NOT EXISTS employees (
            id INT AUTO_INCREMENT PRIMARY KEY,
            name VARCHAR(100),
            salary FLOAT
        )
    ''')

    # Insert data
    cursor.execute("INSERT INTO employees (name, salary) VALUES (%s, %s)", ("John",
50000))

    # Commit the changes
    connection.commit()

    # Retrieve data
    cursor.execute("SELECT * FROM employees")
```

```
    rows = cursor.fetchall()

    for row in rows:
        print(row)

finally:
    # Close the connection
    connection.close()
```

- Creating and executing SQL queries from Python using these connectors.

➢ SQLite3: Creating and Executing SQL Queries

SQLite3 is part of Python's standard library, making it easy to set up and use.

```
import sqlite3

# Step 1: Connect to the SQLite database (or create it if it doesn't exist)
connection = sqlite3.connect("example.db")

# Step 2: Create a cursor object
cursor = connection.cursor()

# Step 3: Execute SQL queries
# Create a table
cursor.execute('''
    CREATE TABLE IF NOT EXISTS users (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        name TEXT NOT NULL,
        age INTEGER
    )
''')

# Insert data into the table
cursor.execute("INSERT INTO users (name, age) VALUES (?, ?)", ("Alice", 25))
cursor.execute("INSERT INTO users (name, age) VALUES (?, ?)", ("Bob", 30))

# Commit the changes
connection.commit()
```

```
# Retrieve and print data
cursor.execute("SELECT * FROM users")
rows = cursor.fetchall()
print("Users:")
for row in rows:
    print(row)

# Step 4: Close the connection
connection.close()
```

- PyMySQL: Creating and Executing SQL Queries

PyMySQL is used to connect to a MySQL or MariaDB database server. Before proceeding, ensure you have:

Installed PyMySQL (pip install pymysql).
Access to a MySQL database server.

```
import pymysql

# Step 1: Connect to the MySQL database

connection = pymysql.connect(

        host='localhost',

        user='your_username',

        password='your_password',

        database='your_database'

)

try:

 # Step 2: Create a cursor object

        cursor = connection.cursor()


# Step 3: Execute SQL queries

# Create a table

cursor.execute('''
```

```
CREATE TABLE IF NOT EXISTS employees (
        id INT AUTO_INCREMENT PRIMARY KEY,
        name VARCHAR(100),
        salary FLOAT
)
''')

# Insert data into the table
   cursor.execute("INSERT INTO employees (name, salary) VALUES (%s, %s)", ("John Doe", 55000))
 cursor.execute("INSERT INTO employees (name, salary) VALUES (%s, %s)", ("Jane Smith", 60000))

# Commit the changes
connection.commit()

# Retrieve and print data
 cursor.execute("SELECT * FROM employees")
   rows = cursor.fetchall()
 print("Employees:")
 for row in rows:
        print(row)

finally:
# Step 4: Close the connection
connection.close()
```

10. Search and Match Functions

- Using re.search() and re.match() functions in Python's re module for pattern matching.

- re.search()

  The re.search() function searches the entire string for the first match of the specified pattern.

  Syntax:

  ```
  re.search(pattern, string, flags=0)
  ```

  `pattern`: The regular expression pattern to search for.
  `string`: The string to search in.
  `flags`: Optional flags to modify the search behaviour.

  Key Characteristics:

  Scans the entire string.
  Returns a Match object if a match is found, otherwise returns None.

  Example:

  ```
  import re

  text = "Hello, welcome to Python programming."

  # Searching for a word anywhere in the string
  result = re.search(r"Python", text)

  if result:
      print("Match found:", result.group())  # Output: Match found: Python
  else:
      print("No match found")
  ```

- re.match()

  The re.match() function checks for a match only at the beginning of the string.

  Syntax:

  ```
  re.match(pattern, string, flags=0)
  ```

`pattern`: The regular expression pattern to match.
`string`: The string to search in.
`flags`: Optional flags to modify the match behavior.

Key Characteristics:

Matches only at the start of the string.
Returns a Match object if the match is found, otherwise returns None.

Example:

```
text = "Hello, welcome to Python programming."

# Matching a word at the start of the string
result = re.match(r"Hello", text)

if result:
    print("Match found:", result.group())  # Output: Match found: Hello
else:
    print("No match found")
```

➢ Difference between search and match.

re.search()	re.match()
Searches the entire string for a match.	Matches only at the beginning of the string.
Returns the first match found (anywhere).	Returns a match only if it starts at the beginning of the string.
Used to find a pattern anywhere in a string.	Used to validate that a string starts with a specific pattern.
Searching for substrings or keywords.	Validating structured input or prefixes.
Searching a keyword in a log or text file.	Checking if a string starts with "http".