Module 8 –
Working with RDBMS

Intro

Working with Relational Database Management Systems (RDBMS) in Python involves using libraries or frameworks that enable interaction with databases, executing queries, fetching data, and performing operations like insertion, deletion, and updates.

Python provides several libraries and frameworks for working with RDBMS, with **SQLAlchemy** and **Django ORM** being two of the most popular ones.

Here's a basic overview of how you can work with RDBMS in Python:

Intro

Choose an RDBMS: Before you start working with RDBMS in Python, you need to choose an RDBMS like MySQL, PostgreSQL, SQLite, or Oracle. Each of these may have slightly different configurations and connection methods.

Install Database Driver: Install the Python library that acts as a connector between your Python code and the chosen RDBMS. For example, for MySQL, you might use **mysql-connector-python**, for PostgreSQL, you might use **psycopg2**, and for SQLite, you typically don't need to install any additional drivers as SQLite is included in Python's standard library.

Establish Connection: Open a connection to your database using the appropriate driver and providing necessary connection parameters like host, port, username, password, and database name.

Intro

Execute Queries: Once the connection is established, you can execute SQL queries using Python. You can either write raw SQL queries or use an ORM (Object-Relational Mapping) like SQLAlchemy or Django ORM to interact with the database using Pythonic syntax rather than writing raw SQL.

Fetch Data: After executing SELECT queries, you can fetch the resulting data rows from the cursor object returned by the query execution.

Handle Transactions: If your operations involve multiple SQL queries that need to be executed as a single unit of work (e.g., transferring funds between accounts), you can handle transactions to ensure data consistency and integrity.

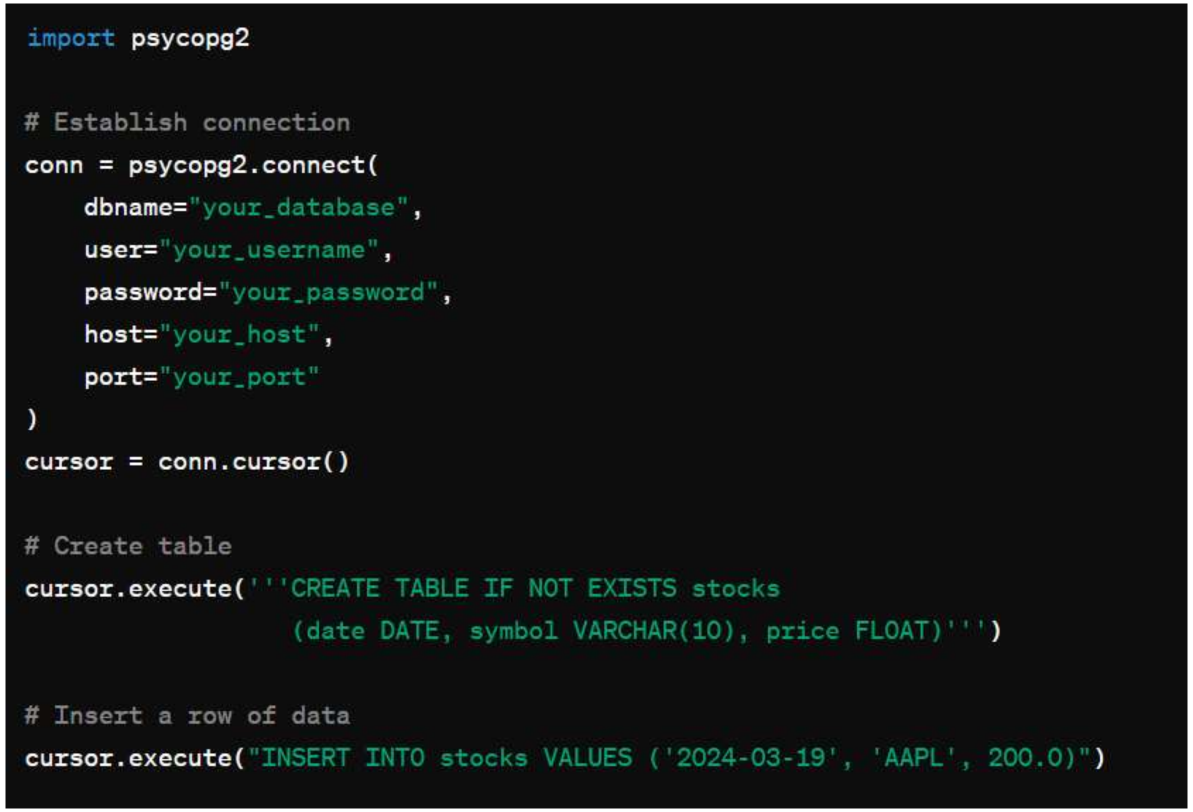
Close Connection: Once you're done with the database operations, close the connection to release resources and avoid memory leaks.

Interact with Postgres

To interact with a PostgreSQL (psql) database in Python, you'll typically use the **psycopg2** library, which is a PostgreSQL adapter for Python.

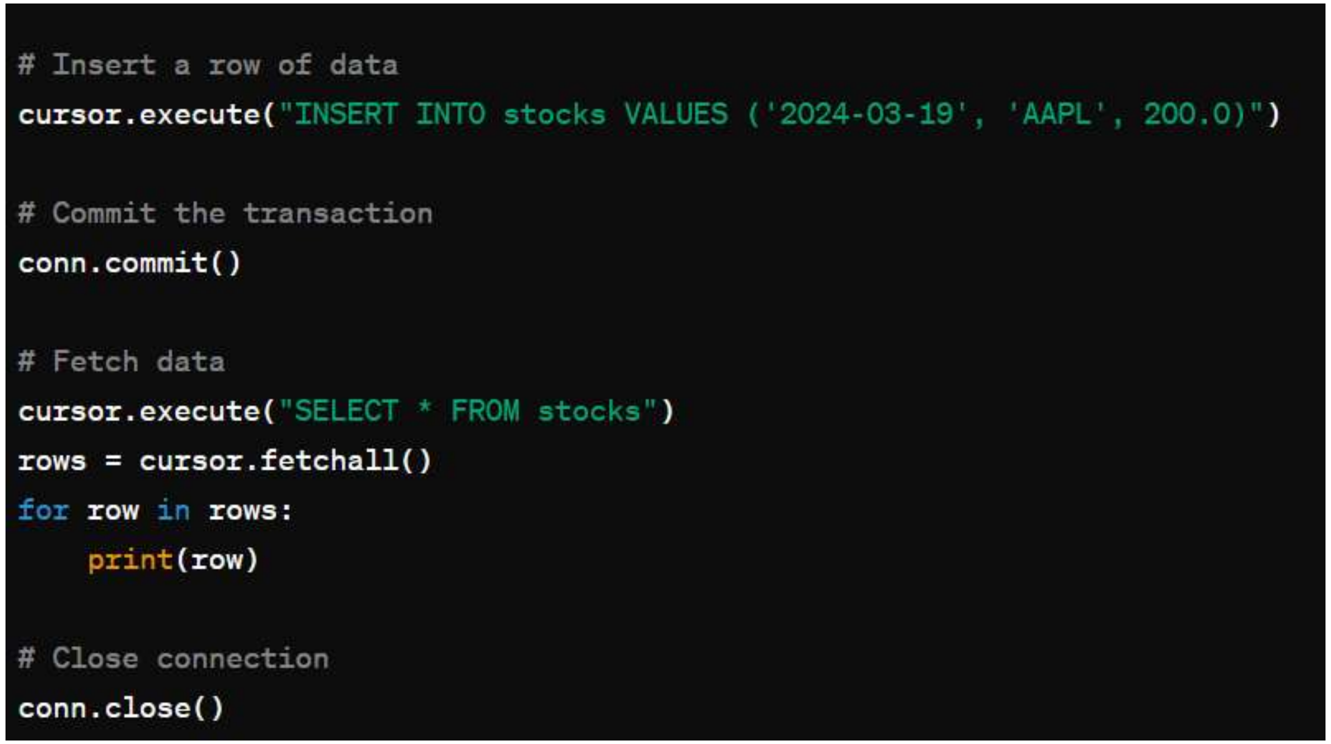
Below is an example demonstrating how to connect to a PostgreSQL database, create a table, insert data, fetch data, and close the connection using **psycopg2**:

Interact with Postgres



```python
import psycopg2

# Establish connection
conn = psycopg2.connect(
    dbname="your_database",
    user="your_username",
    password="your_password",
    host="your_host",
    port="your_port"
)
cursor = conn.cursor()

# Create table
cursor.execute('''CREATE TABLE IF NOT EXISTS stocks
                  (date DATE, symbol VARCHAR(10), price FLOAT)''')

# Insert a row of data
cursor.execute("INSERT INTO stocks VALUES ('2024-03-19', 'AAPL', 200.0)")
```

Interact with Postgres



```python
# Insert a row of data
cursor.execute("INSERT INTO stocks VALUES ('2024-03-19', 'AAPL', 200.0)")

# Commit the transaction
conn.commit()

# Fetch data
cursor.execute("SELECT * FROM stocks")
rows = cursor.fetchall()
for row in rows:
    print(row)

# Close connection
conn.close()
```

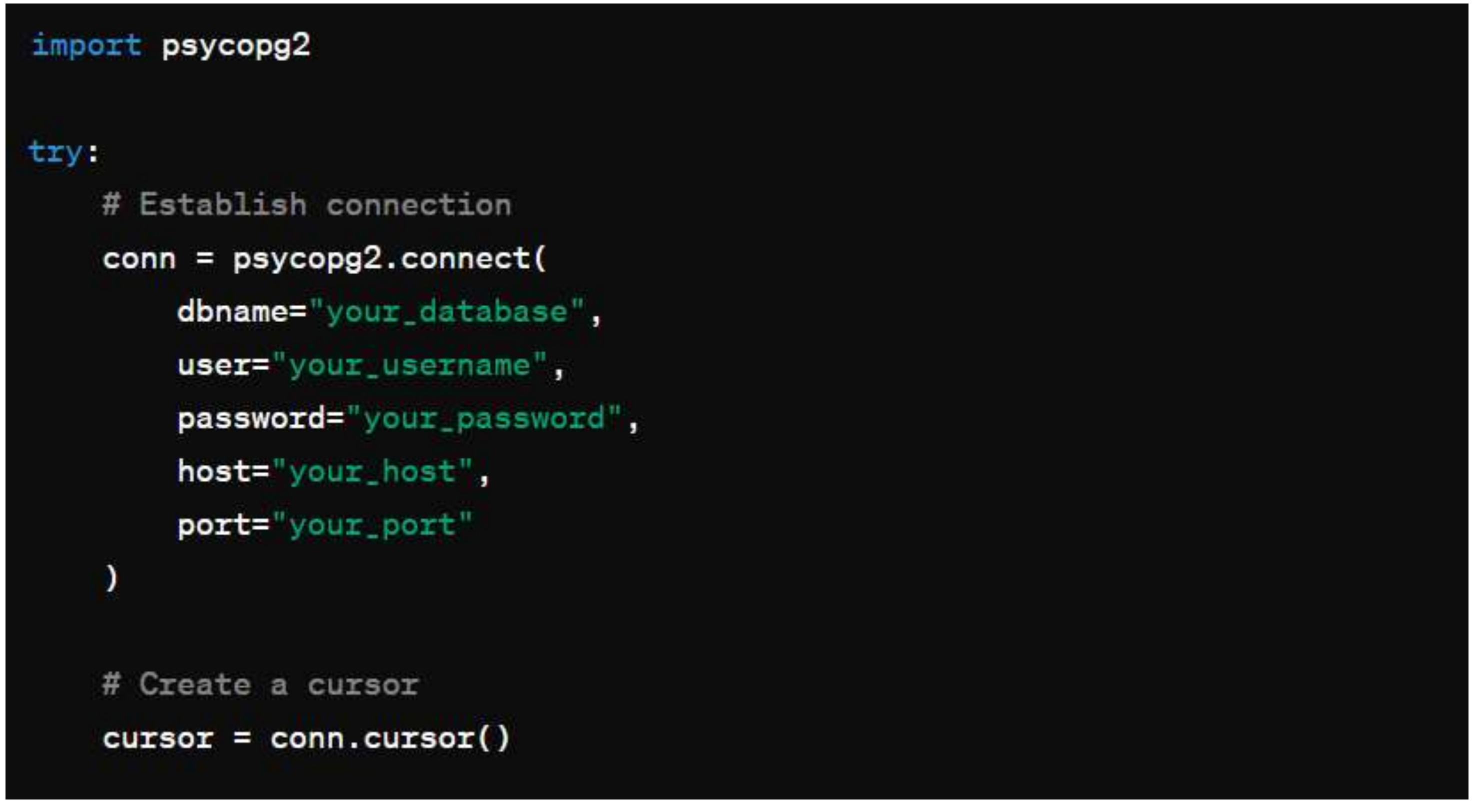
Interact with Postgres

Make sure to replace **"your_database"**, **"your_username"**, **"your_password"**, **"your_host"**, and **"your_port"** with your actual database credentials.

In this example, **psycopg2.connect()** establishes a connection to the PostgreSQL database.

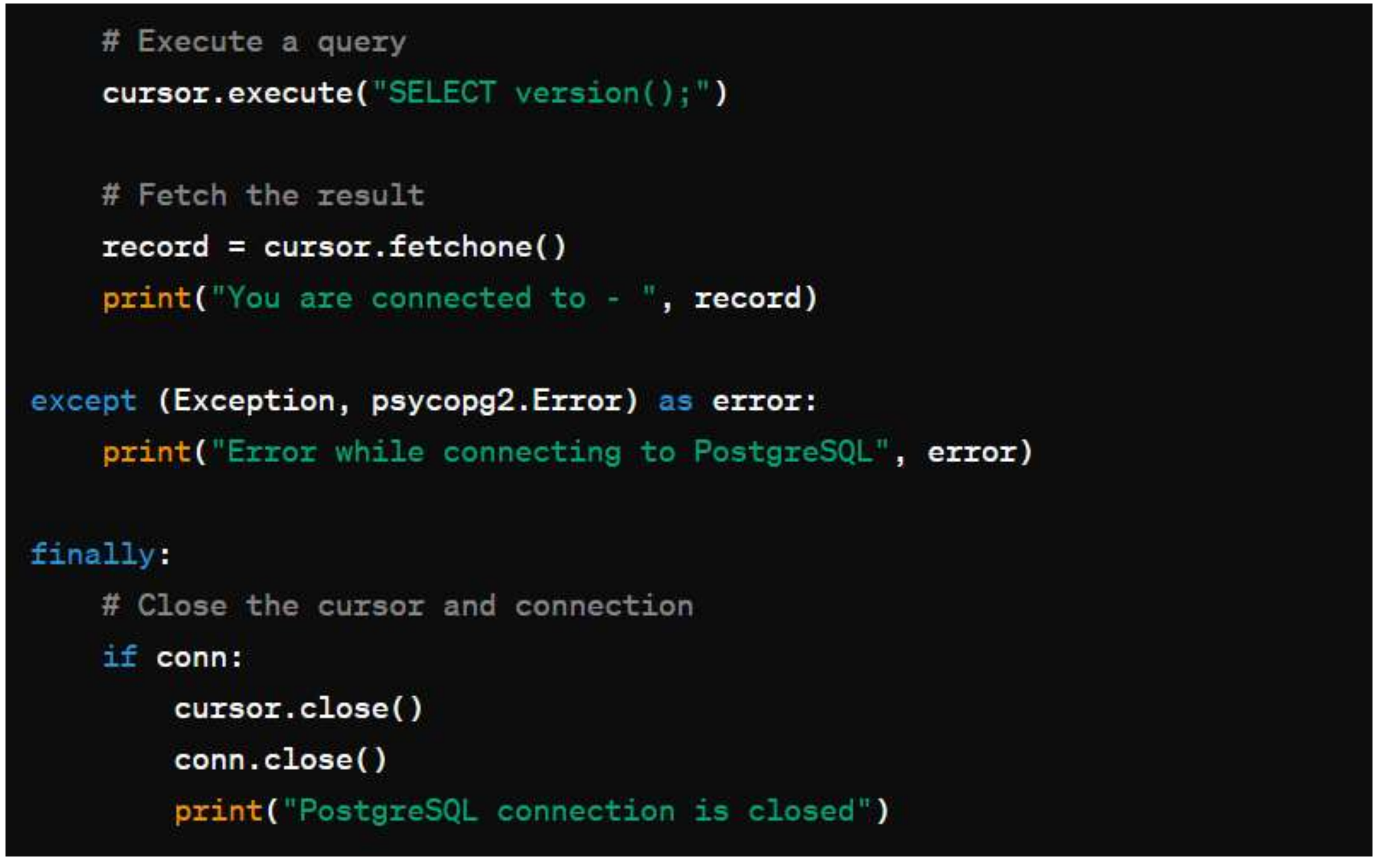
Then, SQL queries are executed using the cursor object (**cursor.execute()**), and the changes are committed to the database (**conn.commit()**). Finally, fetched data is printed, and the connection is closed (**conn.close()**).

Connection to RDBMS Psql



```python
import psycopg2

try:
    # Establish connection
    conn = psycopg2.connect(
        dbname="your_database",
        user="your_username",
        password="your_password",
        host="your_host",
        port="your_port"
    )

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

Connection to RDBMS Psql



```
    # Execute a query
    cursor.execute("SELECT version();")

    # Fetch the result
    record = cursor.fetchone()
    print("You are connected to - ", record)

except (Exception, psycopg2.Error) as error:
    print("Error while connecting to PostgreSQL", error)

finally:
    # Close the cursor and connection
    if conn:
        cursor.close()
        conn.close()
        print("PostgreSQL connection is closed")
```

Connection to RDBMS Psql

In this example:

Replace **"your_database"**, **"your_username"**, **"your_password"**, **"your_host"**, and **"your_port"** with your actual database credentials.

We use a **try-except-finally** block to handle connection, query execution, and resource closing. This ensures that the connection is properly closed even if an exception occurs.

Inside the **try** block, we establish a connection to the PostgreSQL database using **psycopg2.connect()** and create a cursor object to execute queries.

We execute a simple query (**SELECT version();**) to retrieve the PostgreSQL server version.

Connection to RDBMS Psql

We fetch the result using **cursor.fetchone()** and print it.

If an error occurs during the connection or query execution, it is caught and printed.

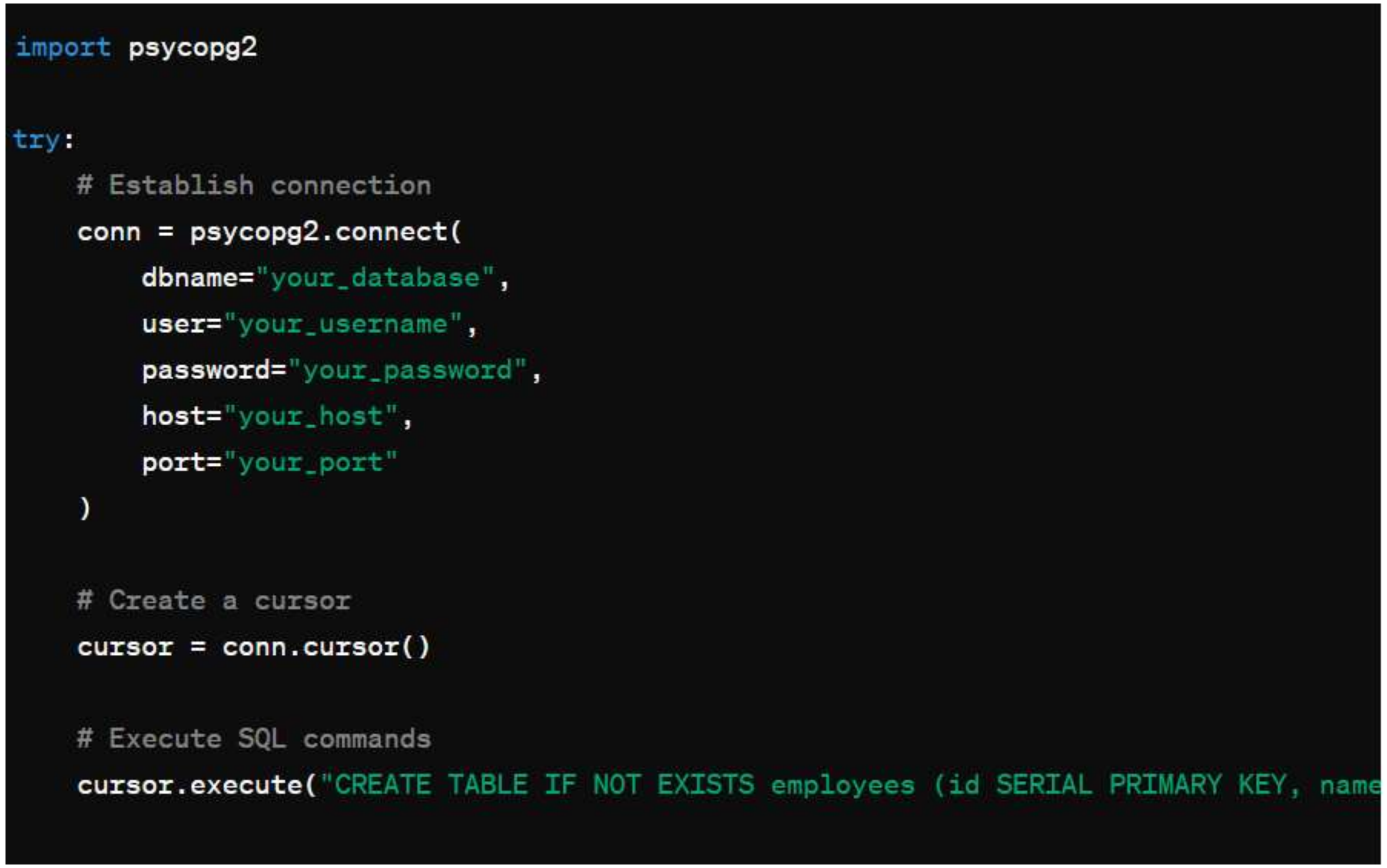
In the **finally** block, we ensure that the cursor and connection are properly closed using **cursor.close()** and **conn.close()** respectively.

Remember to install **psycopg2** if you haven't already by running **pip install psycopg2**. Additionally, ensure that your PostgreSQL server is running and that you have the correct credentials and access permissions.

Cursor creation

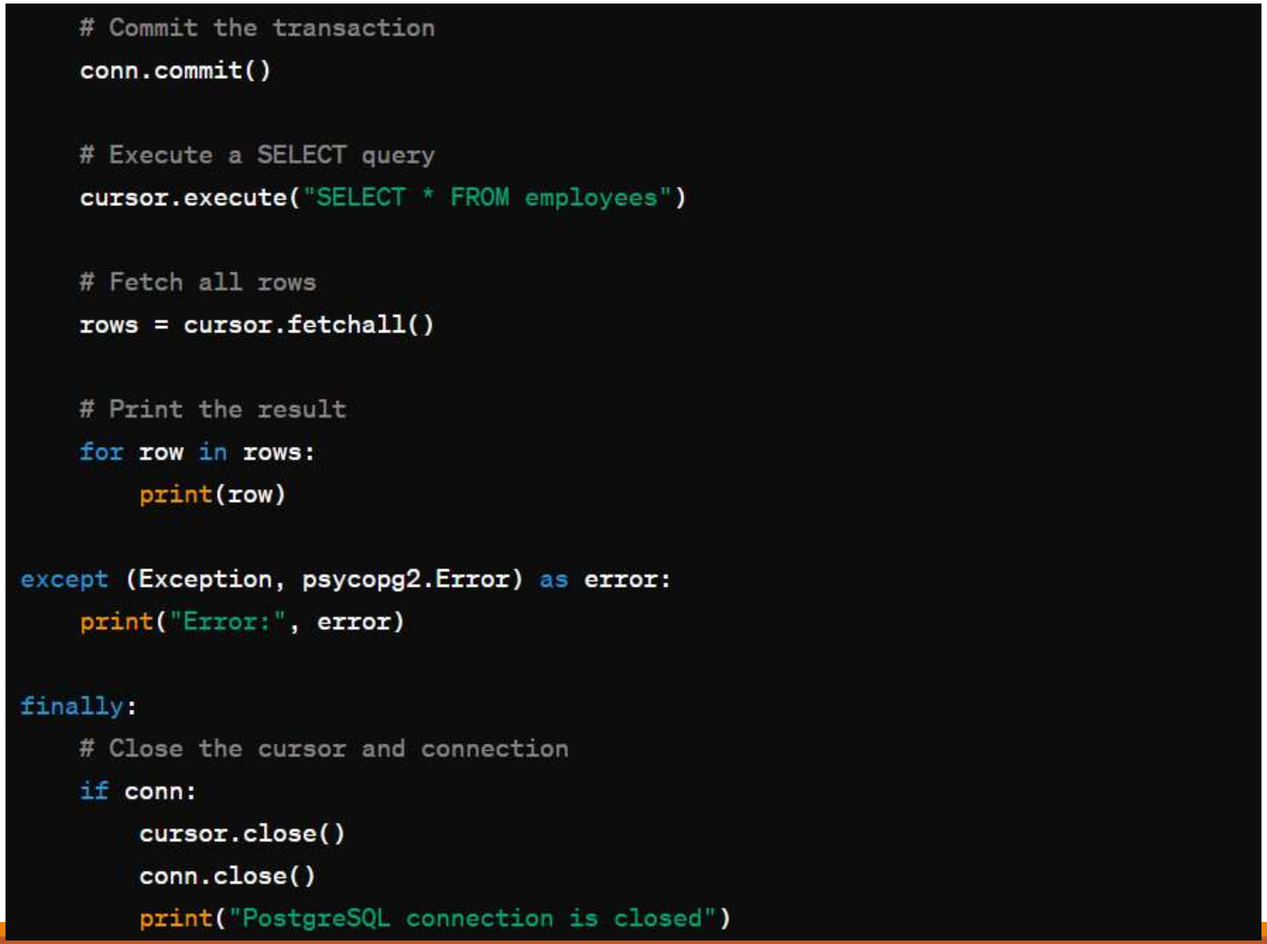
Creating a cursor in **psycopg2** allows you to execute SQL commands against a PostgreSQL database. Cursors are objects that enable you to interact with the database and retrieve query results. Here's how you can create a cursor and execute SQL commands using **psycopg2**:

Cursor creation



```
import psycopg2

try:
    # Establish connection
    conn = psycopg2.connect(
        dbname="your_database",
        user="your_username",
        password="your_password",
        host="your_host",
        port="your_port"
    )

    # Create a cursor
    cursor = conn.cursor()

    # Execute SQL commands
    cursor.execute("CREATE TABLE IF NOT EXISTS employees (id SERIAL PRIMARY KEY, name
```

Cursor creation



```
    # Commit the transaction
    conn.commit()

    # Execute a SELECT query
    cursor.execute("SELECT * FROM employees")

    # Fetch all rows
    rows = cursor.fetchall()

    # Print the result
    for row in rows:
        print(row)

except (Exception, psycopg2.Error) as error:
    print("Error:", error)

finally:
    # Close the cursor and connection
    if conn:
        cursor.close()
        conn.close()
        print("PostgreSQL connection is closed")
```

Cursor creation

We establish a connection to the PostgreSQL database as shown before.

Then, we create a cursor using **conn.cursor()**. This cursor is used to execute SQL commands.

We execute SQL commands like creating a table (**CREATE TABLE**), inserting data (**INSERT INTO**), and selecting data (**SELECT * FROM**). Parameterized queries are used to prevent SQL injection.

After executing **INSERT** queries, we commit the transaction using **conn.commit()**.

We execute a **SELECT** query to fetch all rows from the **employees** table using **cursor.execute()** and retrieve the result using **cursor.fetchall()**.

Finally, we print the fetched rows and close the cursor and connection in the **finally** block.

Fire Query & Collect Results

To fire a query and collect results from tables or queries in PostgreSQL using **psycopg2**, you can follow these steps:

Establish Connection: Start by establishing a connection to your PostgreSQL database.

Create a Cursor: Create a cursor object to execute SQL commands.

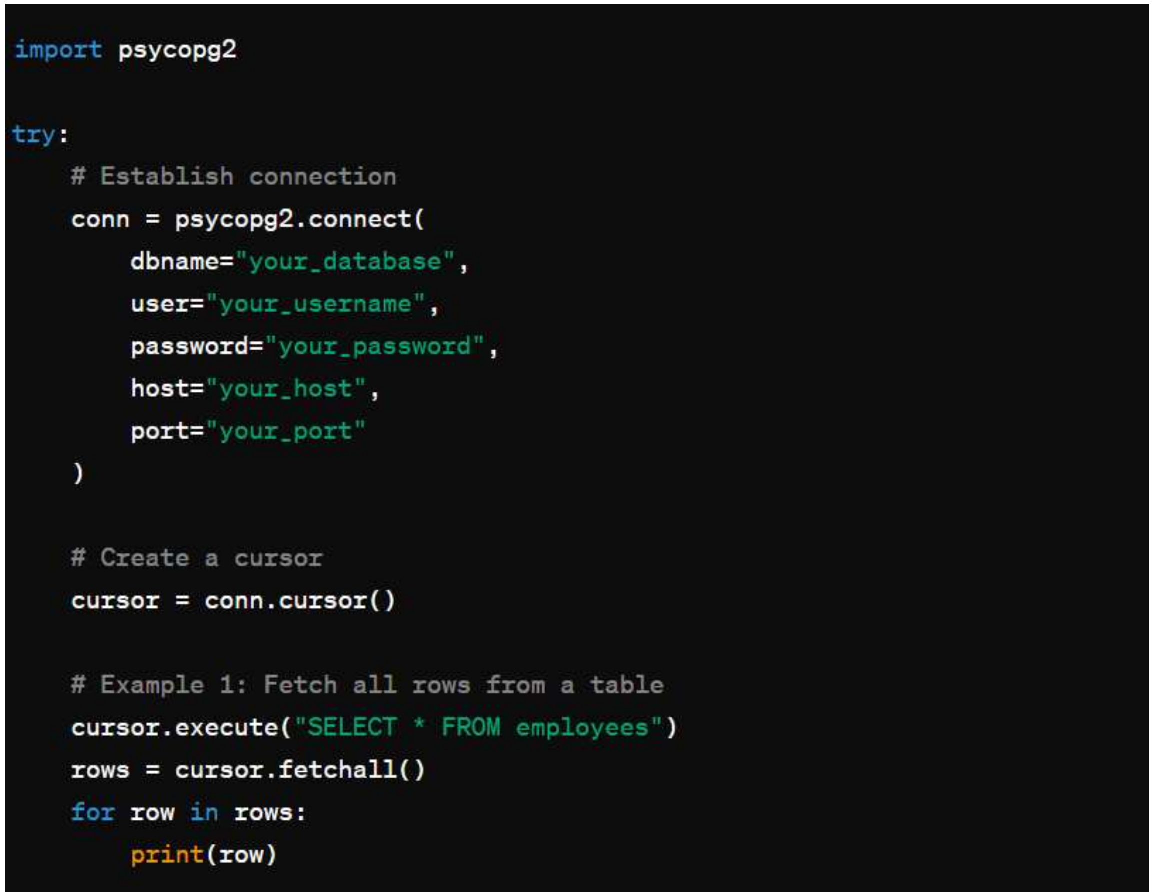
Execute Query: Execute your SQL query using the cursor's **execute()** method.

Fetch Results: Depending on the type of query, fetch the results using appropriate methods like **fetchone()**, **fetchall()**, or **fetchmany()**.

Process Results: Process the fetched results as needed.

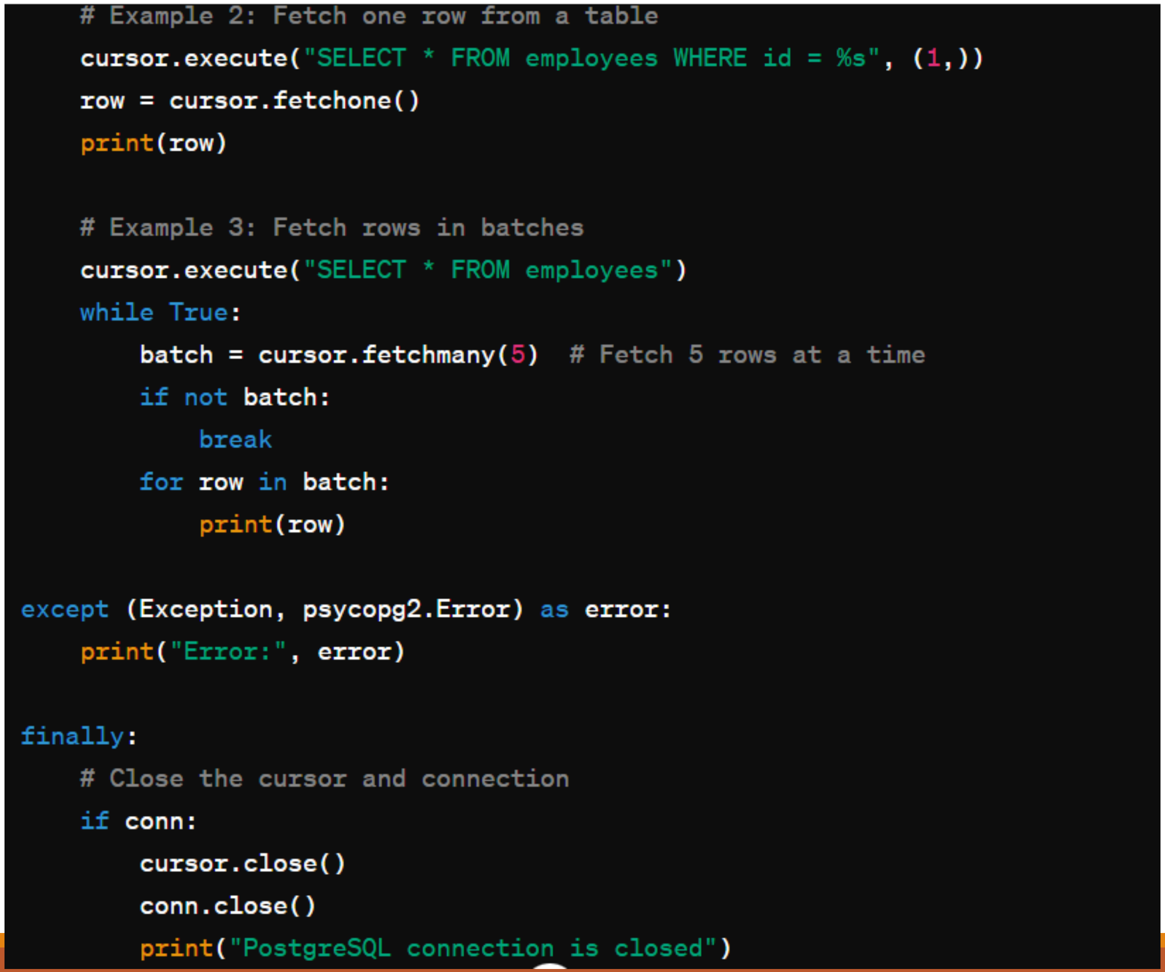
Close Cursor and Connection: Close the cursor and connection when done.

Fire Query & Collect Results



```python
import psycopg2

try:
    # Establish connection
    conn = psycopg2.connect(
        dbname="your_database",
        user="your_username",
        password="your_password",
        host="your_host",
        port="your_port"
    )

    # Create a cursor
    cursor = conn.cursor()

    # Example 1: Fetch all rows from a table
    cursor.execute("SELECT * FROM employees")
    rows = cursor.fetchall()
    for row in rows:
        print(row)
```

Fire Query & Collect Results



```
    # Example 2: Fetch one row from a table
    cursor.execute("SELECT * FROM employees WHERE id = %s", (1,))
    row = cursor.fetchone()
    print(row)

    # Example 3: Fetch rows in batches
    cursor.execute("SELECT * FROM employees")
    while True:
        batch = cursor.fetchmany(5)  # Fetch 5 rows at a time
        if not batch:
            break
        for row in batch:
            print(row)

except (Exception, psycopg2.Error) as error:
    print("Error:", error)

finally:
    # Close the cursor and connection
    if conn:
        cursor.close()
        conn.close()
        print("PostgreSQL connection is closed")
```

Fire Query & Collect Results

In this example:

We establish a connection to the PostgreSQL database.

We create a cursor object using **conn.cursor()**.

We execute various SQL queries using **cursor.execute()**.

We fetch results using methods like **fetchall()**, **fetchone()**, or **fetchmany()**.

We process the fetched results as needed.

Finally, we close the cursor and connection in the **finally** block.

Insert Data into Tables and Types

To insert data into tables in PostgreSQL using **psycopg2**, you can follow these steps:

Establish Connection: Begin by establishing a connection to your PostgreSQL database.

Create a Cursor: Create a cursor object to execute SQL commands.

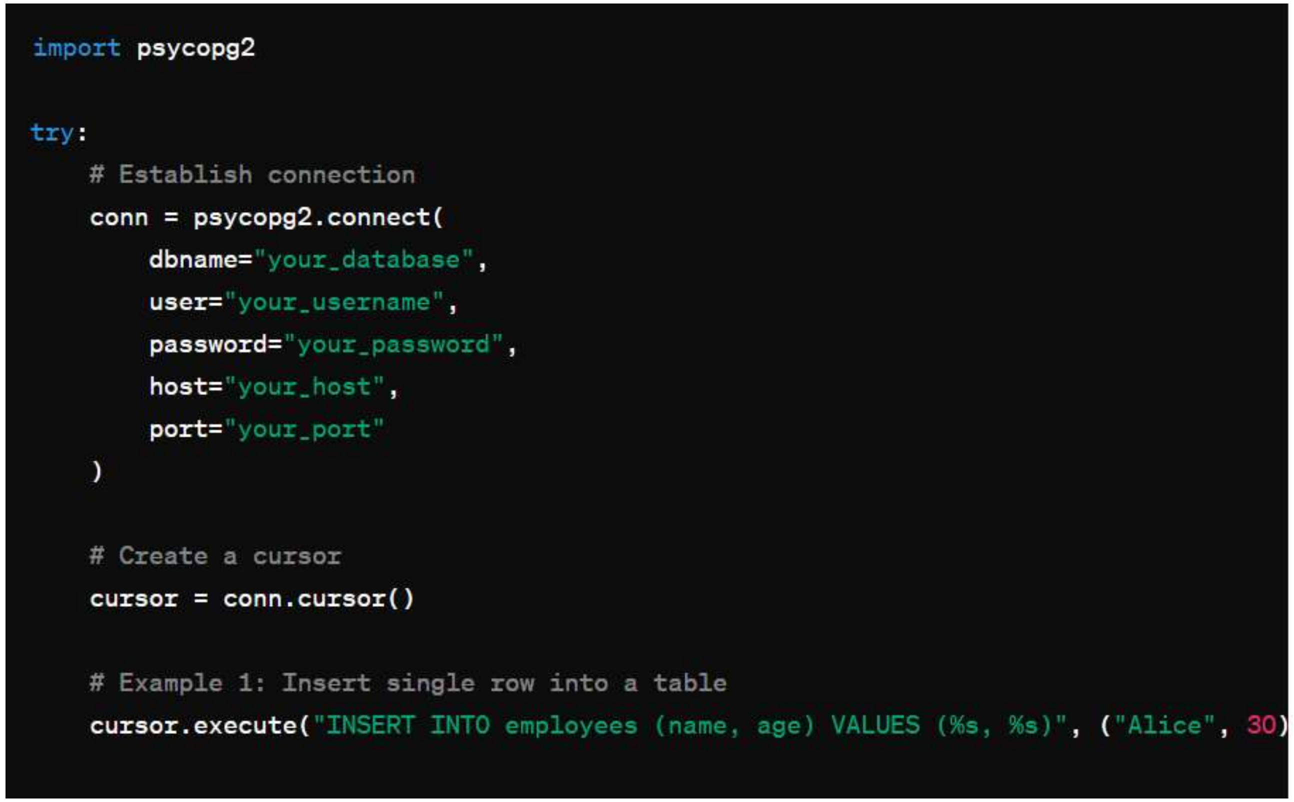
Execute INSERT Queries: Execute **INSERT** queries to add data into the tables.

Commit Changes: If you're inserting data, make sure to commit the transaction to save the changes to the database.

Close Cursor and Connection: Close the cursor and connection when done.

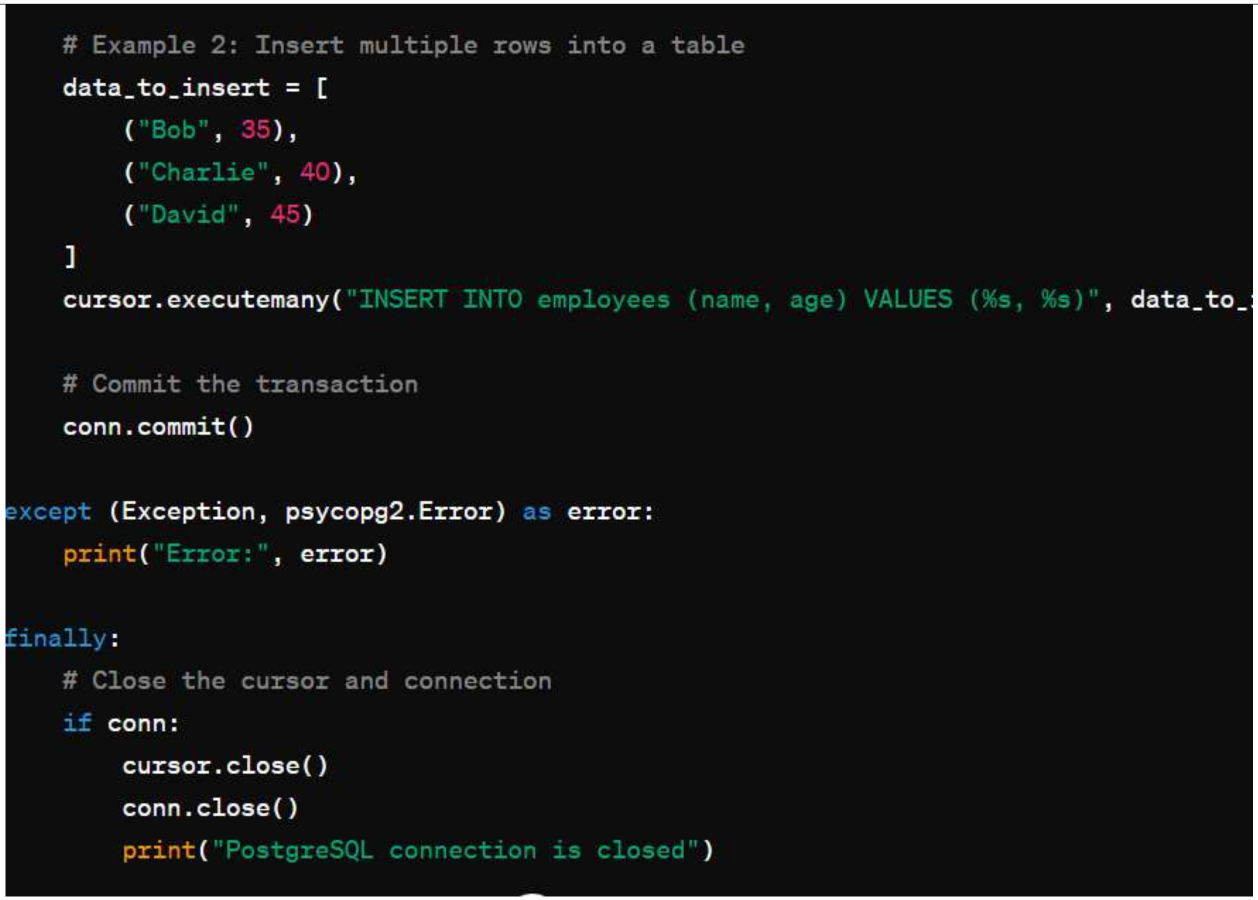
Here's an example demonstrating how to insert data into tables:

Insert Data into Tables and Types



```
import psycopg2

try:
    # Establish connection
    conn = psycopg2.connect(
        dbname="your_database",
        user="your_username",
        password="your_password",
        host="your_host",
        port="your_port"
    )

    # Create a cursor
    cursor = conn.cursor()

    # Example 1: Insert single row into a table
    cursor.execute("INSERT INTO employees (name, age) VALUES (%s, %s)", ("Alice", 30)
```

Insert Data into Tables and Types



```
    # Example 2: Insert multiple rows into a table
    data_to_insert = [
        ("Bob", 35),
        ("Charlie", 40),
        ("David", 45)
    ]
    cursor.executemany("INSERT INTO employees (name, age) VALUES (%s, %s)", data_to_i

    # Commit the transaction
    conn.commit()

except (Exception, psycopg2.Error) as error:
    print("Error:", error)

finally:
    # Close the cursor and connection
    if conn:
        cursor.close()
        conn.close()
        print("PostgreSQL connection is closed")
```

Insert Data into Tables and Types

In this example:

We establish a connection to the PostgreSQL database.

We create a cursor object using **conn.cursor()**.

We execute **INSERT** queries to add data into the **employees** table. You can insert single rows using **cursor.execute()** or multiple rows using **cursor.executemany()**.

After inserting data, we commit the transaction using **conn.commit()** to save the changes to the database.

Finally, we close the cursor and connection in the **finally** block.

Make sure to replace **"your_database"**, **"your_username"**, **"your_password"**, **"your_host"**, and **"your_port"** with your actual database credentials. Additionally, modify the table name and column names according to your database schema.

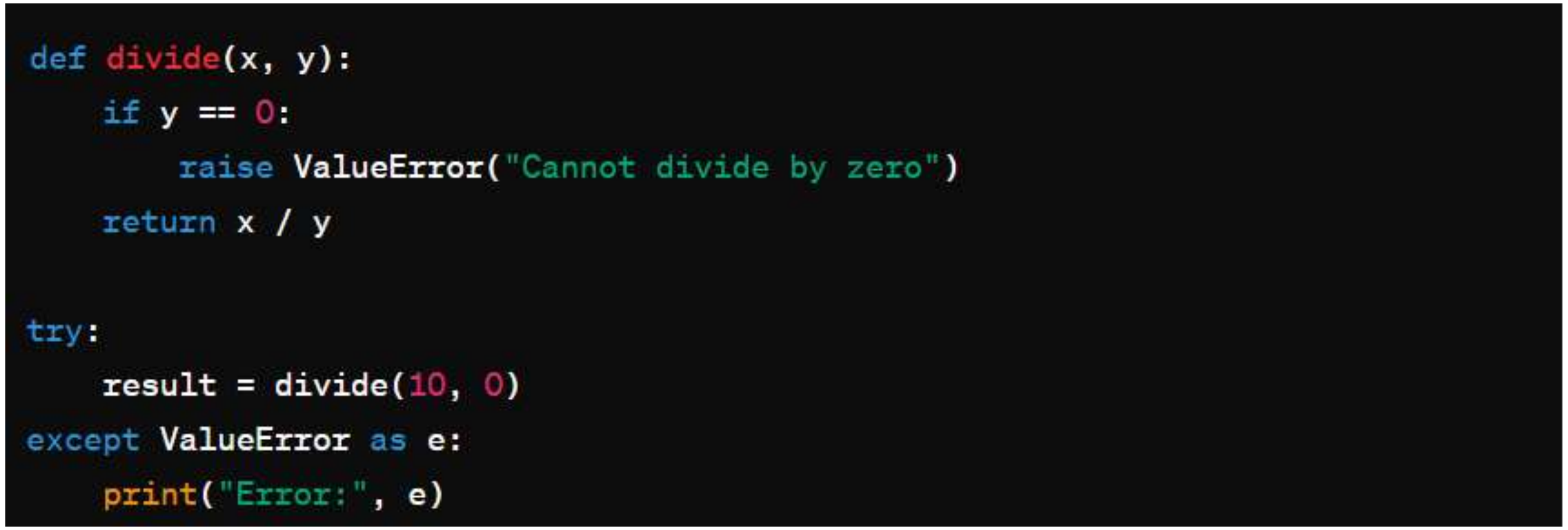
Module 9 – Debugging

Raising Exceptions

Raising Exceptions: Python's exception handling mechanism allows you to raise exceptions when encountering errors or unexpected conditions in your code.

You can use **‘raise’** statement to raise exceptions manually, providing helpful error messages to identify the issue.

Raising Exceptions

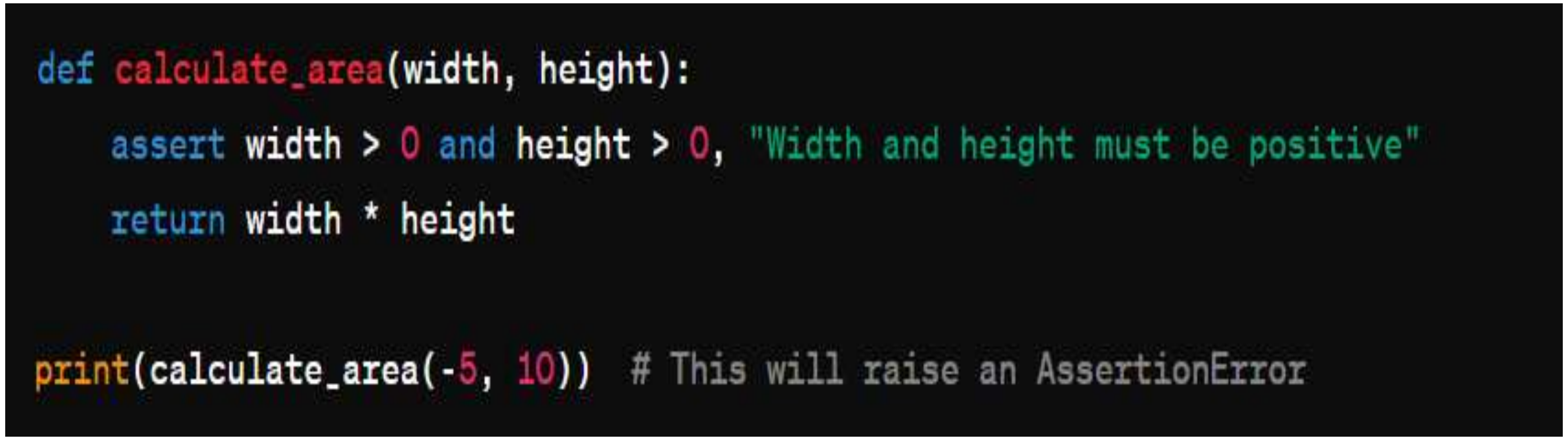


```
def divide(x, y):
    if y == 0:
        raise ValueError("Cannot divide by zero")
    return x / y

try:
    result = divide(10, 0)
except ValueError as e:
    print("Error:", e)
```

In this example, if **y** is 0, the **divide** function raises a **ValueError** with the message "Cannot divide by zero". The exception is caught in the **try-except** block, allowing you to handle the error gracefully.

Assertions

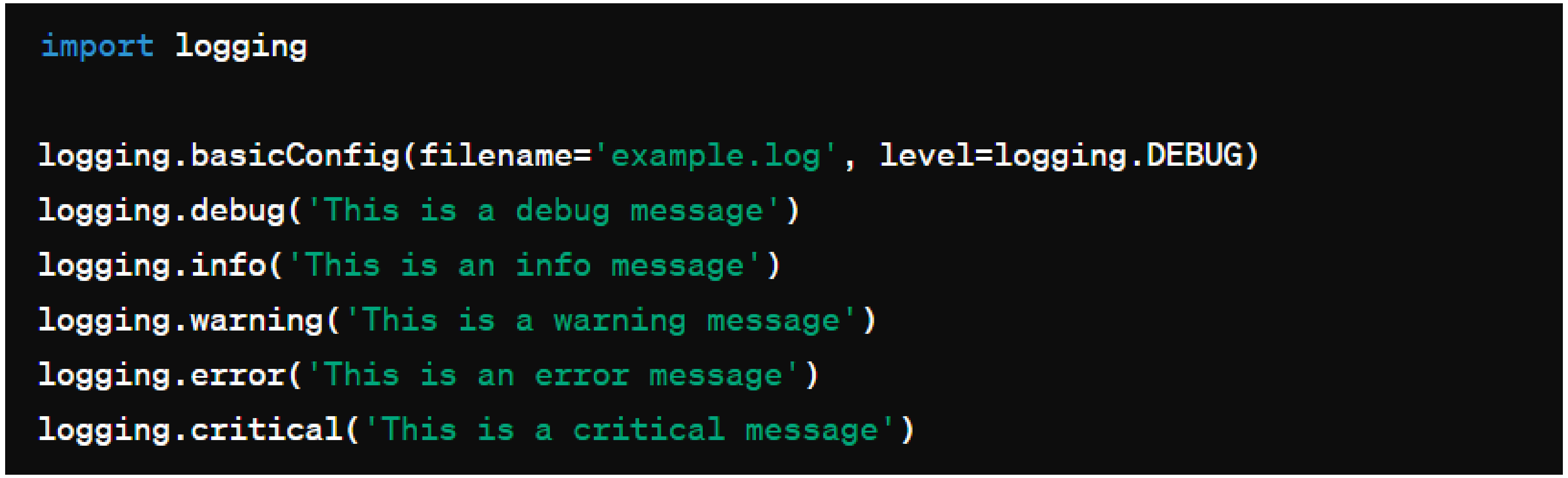
Assertions are used to check if a condition is true, and if it's not, it raises an **AssertionError**. Assertions are often used to check for conditions that should never occur during the execution of the program.



```python
def calculate_area(width, height):
    assert width > 0 and height > 0, "Width and height must be positive"
    return width * height

print(calculate_area(-5, 10))  # This will raise an AssertionError
```

Logging Module and File

The **‘logging’** module in Python provides a flexible framework for logging messages from your code. It supports different logging levels (**DEBUG, INFO, WARNING, ERROR, CRITICAL**) and allows you to direct log messages to different destinations such as the console, files, or network streams.

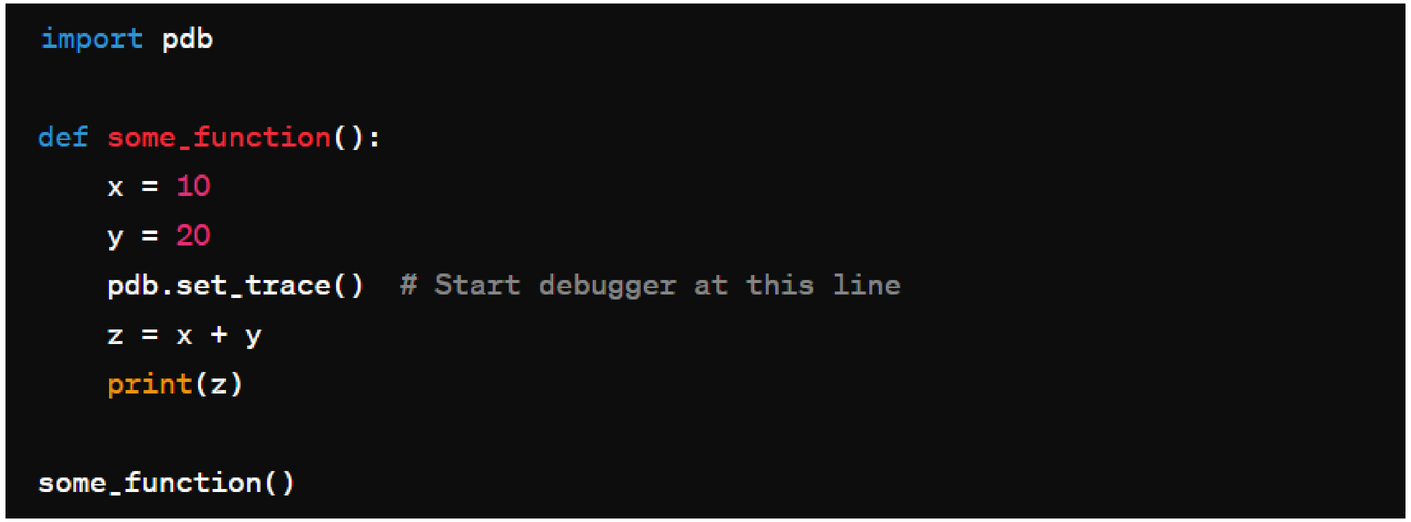


```python
import logging

logging.basicConfig(filename='example.log', level=logging.DEBUG)
logging.debug('This is a debug message')
logging.info('This is an info message')
logging.warning('This is a warning message')
logging.error('This is an error message')
logging.critical('This is a critical message')
```

IDLE’s Debugger

IDLE, Python's Integrated Development and Learning Environment, comes with a built-in debugger that allows you to interactively debug your code.

You can start the debugger from the Debug menu in IDLE or by using the **‘pdb’** module programmatically.



```python
import pdb

def some_function():
    x = 10
    y = 20
    pdb.set_trace()  # Start debugger at this line
    z = x + y
    print(z)

some_function()
```

Breakpoints

Python 3.7 introduced the **breakpoint()** function, which allows you to set breakpoints in your code. When executed, **breakpoint()** enters the debugger at that point in your code.

```
def some_function():
    x = 10
    y = 20
    breakpoint()  # Execution pauses here
    z = x + y
    print(z)

some_function()
```

Breakpoints

Once the debugger is activated at the **breakpoint()** call, you can step through the code and inspect variables just like with **pdb**.

These debugging techniques are essential for identifying and resolving issues in your Python code, making your development process smoother and more efficient.

Module 10 –
Working with CSV Files and JSON

CSV Modules and Delimiters

CSV Module in Python:

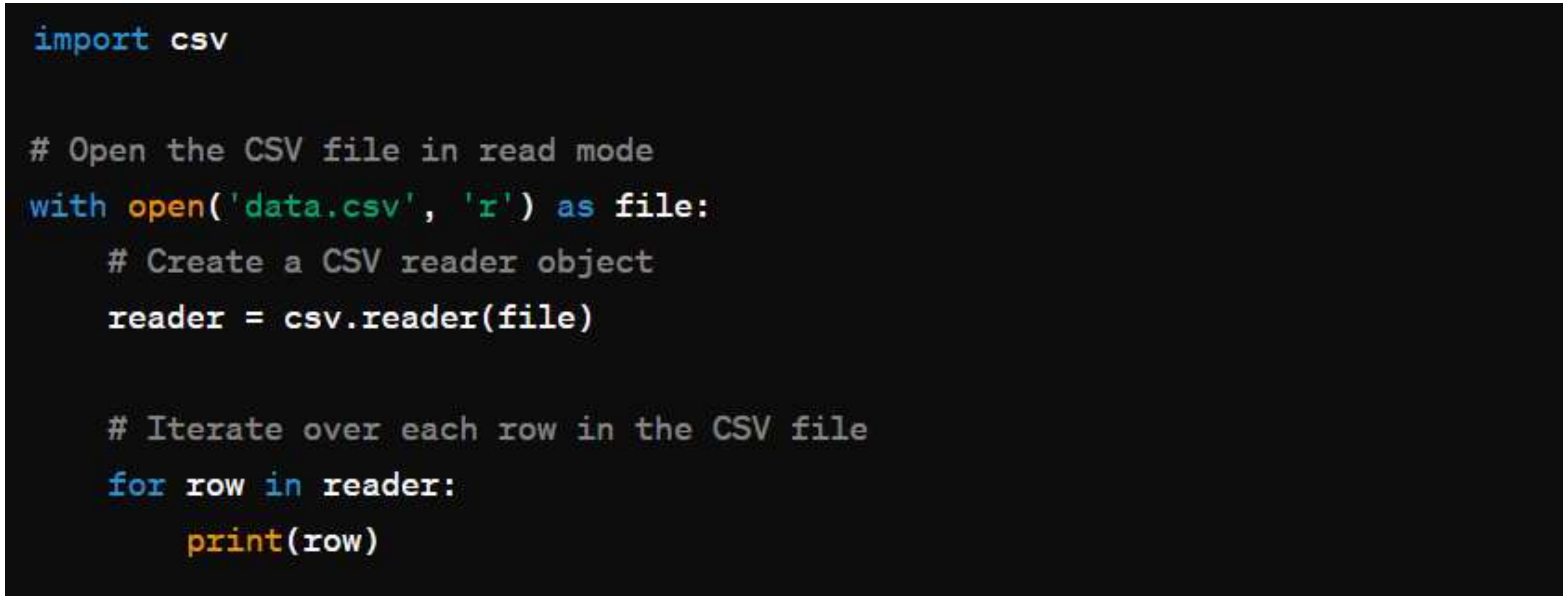
- The **csv** module in Python provides functionalities to read from and write to CSV files.
- It simplifies the process of handling CSV files by providing convenient functions and classes.

Delimiters:

- A delimiter is a character used to separate fields in a CSV file.
- By default, the comma (**,**) is used as the delimiter, but it can be customized based on the requirements of the data.

CSV Modules and Delimiters

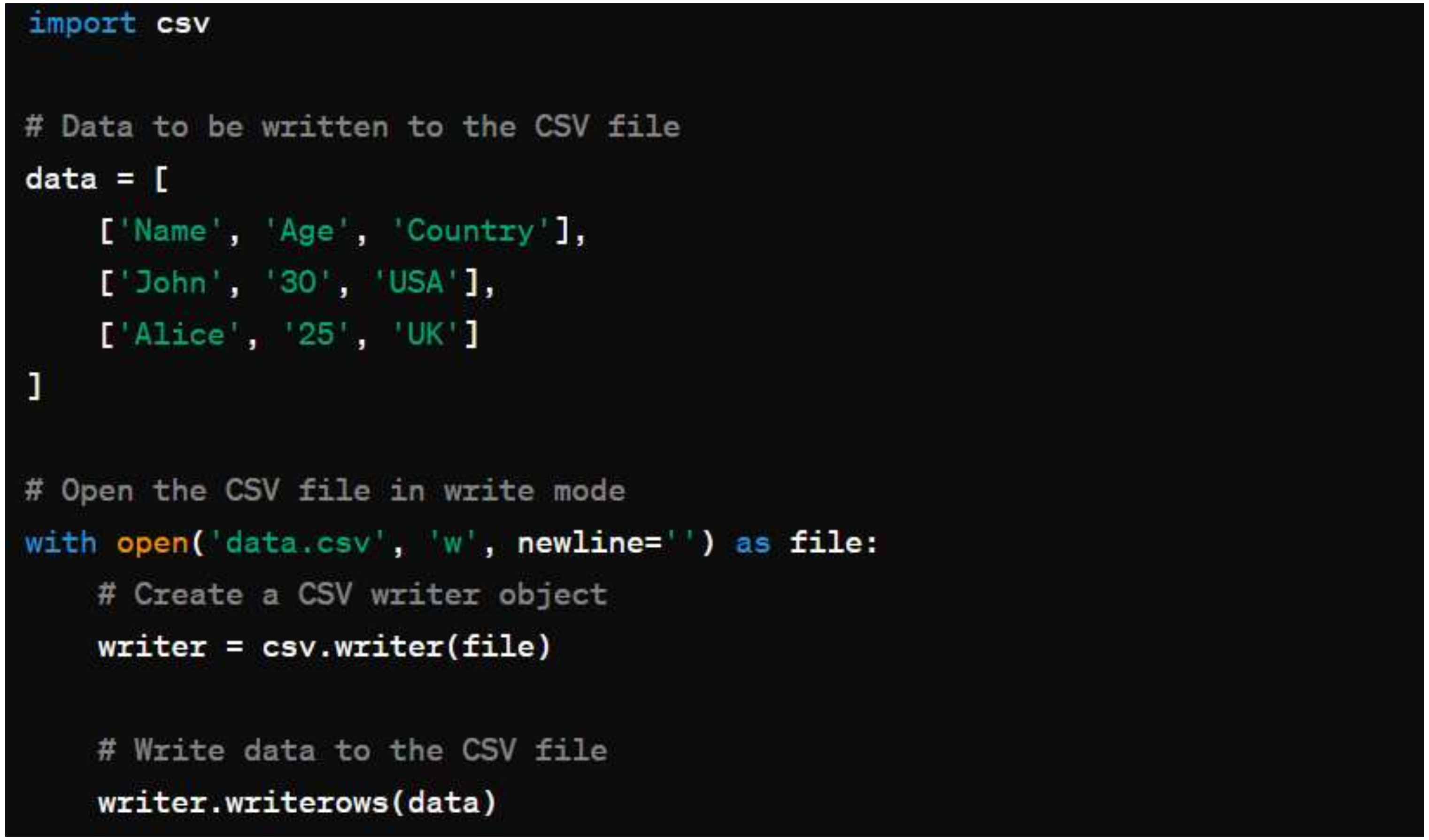
Reading from a CSV File:



```python
import csv

# Open the CSV file in read mode
with open('data.csv', 'r') as file:
    # Create a CSV reader object
    reader = csv.reader(file)

    # Iterate over each row in the CSV file
    for row in reader:
        print(row)
```

CSV Modules and Delimiters

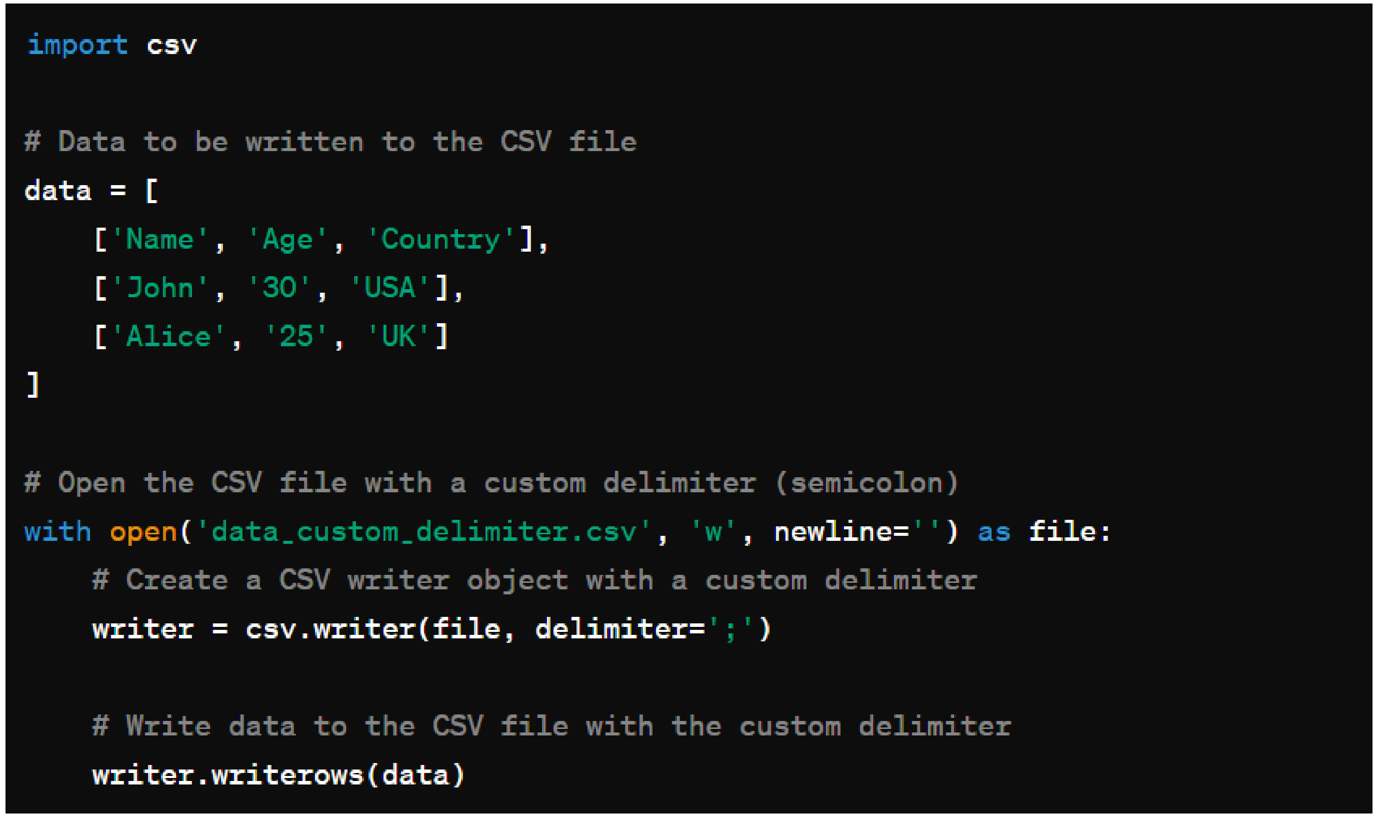
Writing to a CSV File:



```python
import csv

# Data to be written to the CSV file
data = [
    ['Name', 'Age', 'Country'],
    ['John', '30', 'USA'],
    ['Alice', '25', 'UK']
]

# Open the CSV file in write mode
with open('data.csv', 'w', newline='') as file:
    # Create a CSV writer object
    writer = csv.writer(file)

    # Write data to the CSV file
    writer.writerows(data)
```

CSV Modules and Delimiters

Customizing Delimiters:



```python
import csv

# Data to be written to the CSV file
data = [
    ['Name', 'Age', 'Country'],
    ['John', '30', 'USA'],
    ['Alice', '25', 'UK']
]

# Open the CSV file with a custom delimiter (semicolon)
with open('data_custom_delimiter.csv', 'w', newline='') as file:
    # Create a CSV writer object with a custom delimiter
    writer = csv.writer(file, delimiter=';')

    # Write data to the CSV file with the custom delimiter
    writer.writerows(data)
```

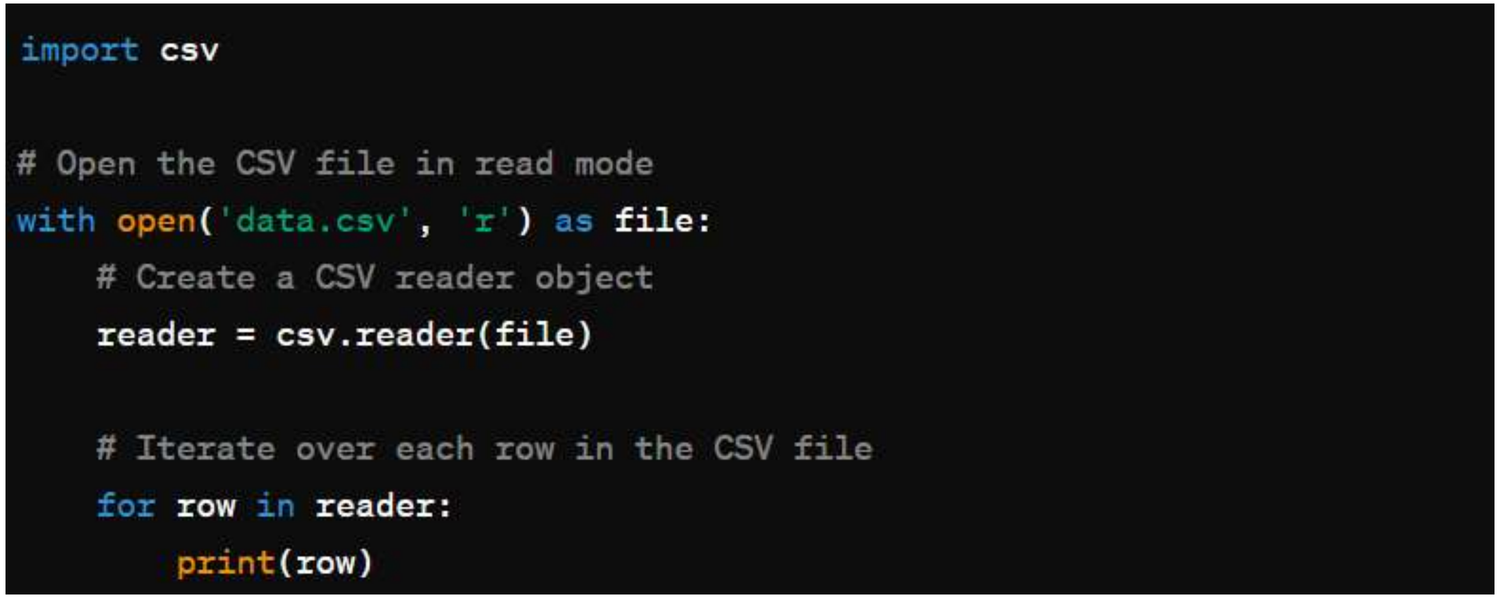
Reader and Writer Objects

Reader Object:

The **csv.reader** object is used to read data from a CSV file row by row.

It provides methods to iterate over each row and retrieve the data as a list of strings.

Reader and Writer Objects



```
import csv

# Open the CSV file in read mode
with open('data.csv', 'r') as file:
    # Create a CSV reader object
    reader = csv.reader(file)

    # Iterate over each row in the CSV file
    for row in reader:
        print(row)
```

In this example, the **csv.reader** object is created to read data from the 'data.csv' file.

The **for** loop iterates over each row in the CSV file, and **row** variable contains a list of values from each row.

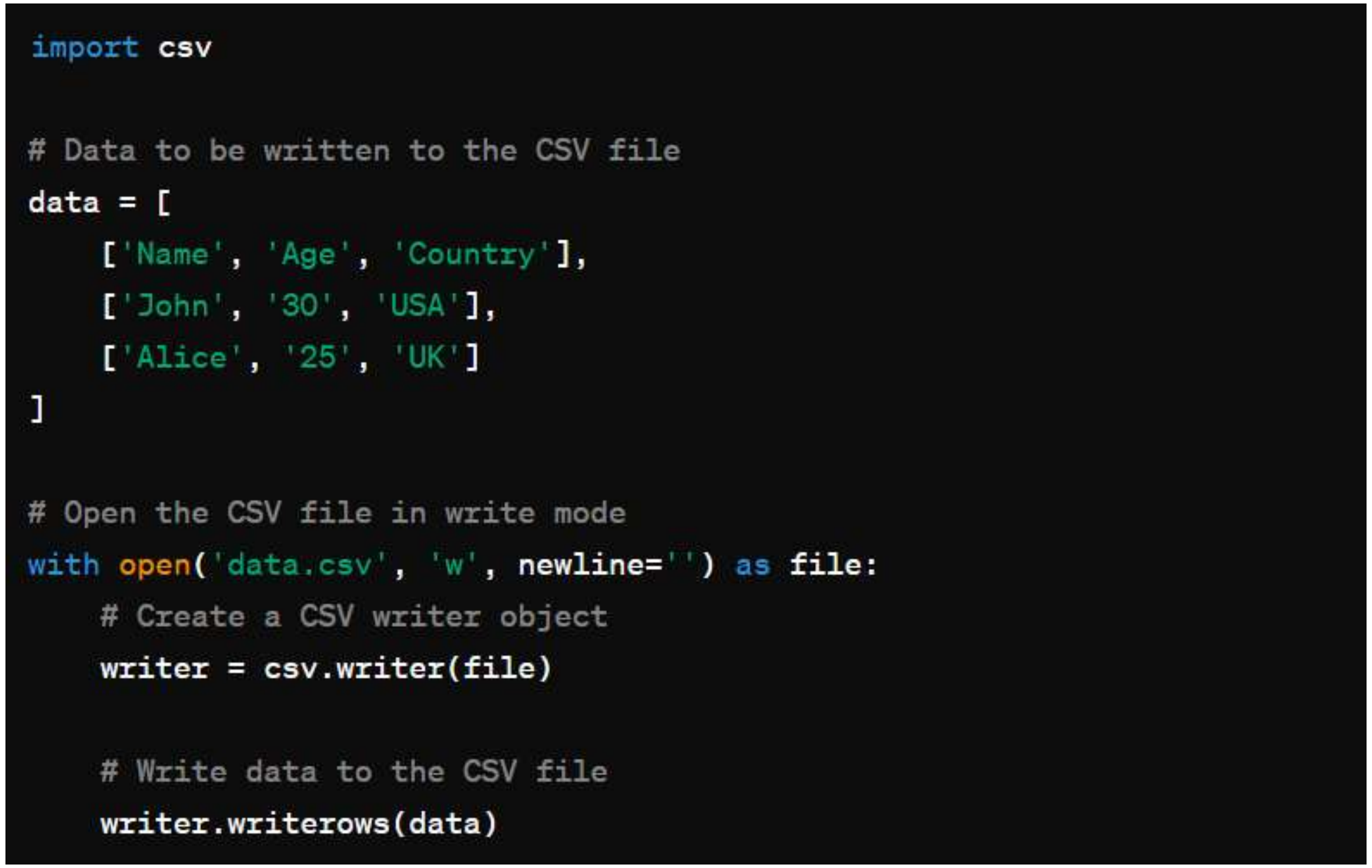
Reader and Writer Objects

Writer Object:

The **csv.writer** object is used to write data to a CSV file.

It provides methods to write rows of data to the file.

Reader and Writer Objects



```
import csv

# Data to be written to the CSV file
data = [
    ['Name', 'Age', 'Country'],
    ['John', '30', 'USA'],
    ['Alice', '25', 'UK']
]

# Open the CSV file in write mode
with open('data.csv', 'w', newline='') as file:
    # Create a CSV writer object
    writer = csv.writer(file)

    # Write data to the CSV file
    writer.writerows(data)
```

Reader and Writer Objects

In this example, the **csv.writer** object is created to write data to the 'data.csv' file.

The **writer.writerows()** method is used to write multiple rows of data to the CSV file.

Each inner list in the **data** list represents a row in the CSV file.

These Reader and Writer objects in the CSV module provide convenient ways to read from and write to CSV files in Python, making it easier to work with tabular data.

JSON Module and I/O Functions

1. **JSON Module**:

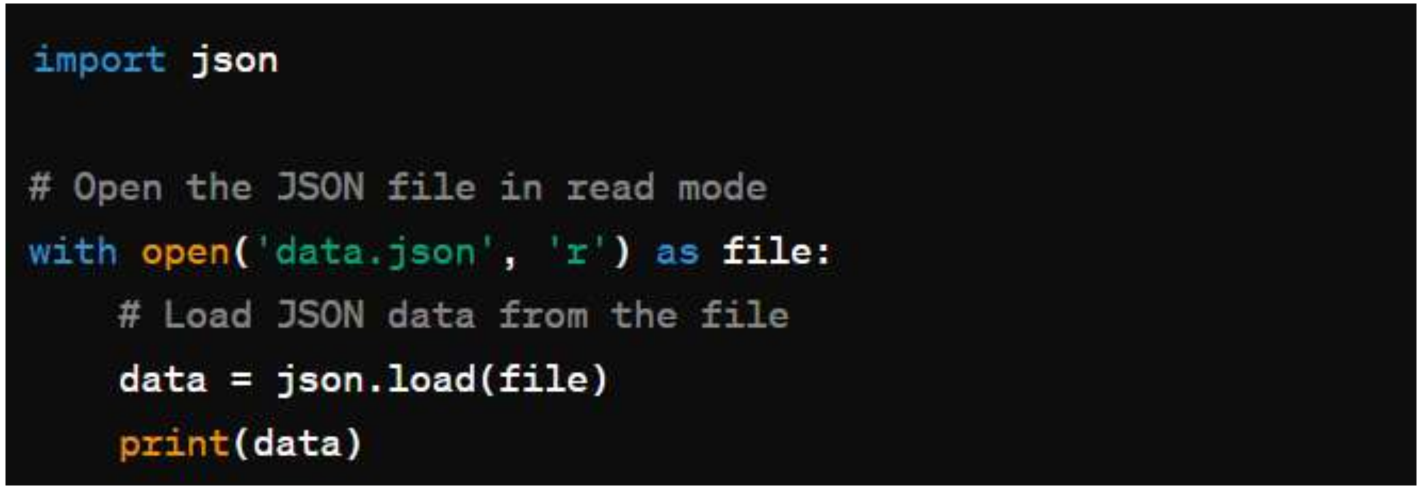
- The **json** module in Python provides functions for encoding and decoding JSON data.

- It allows you to serialize Python objects into JSON strings and deserialize JSON strings into Python objects.

JSON Module and I/O Functions

2. **Reading from JSON**:

- Use **json.load(file)** or **json.loads(json_string)** functions to read JSON data from a file object or a JSON-formatted string, respectively.

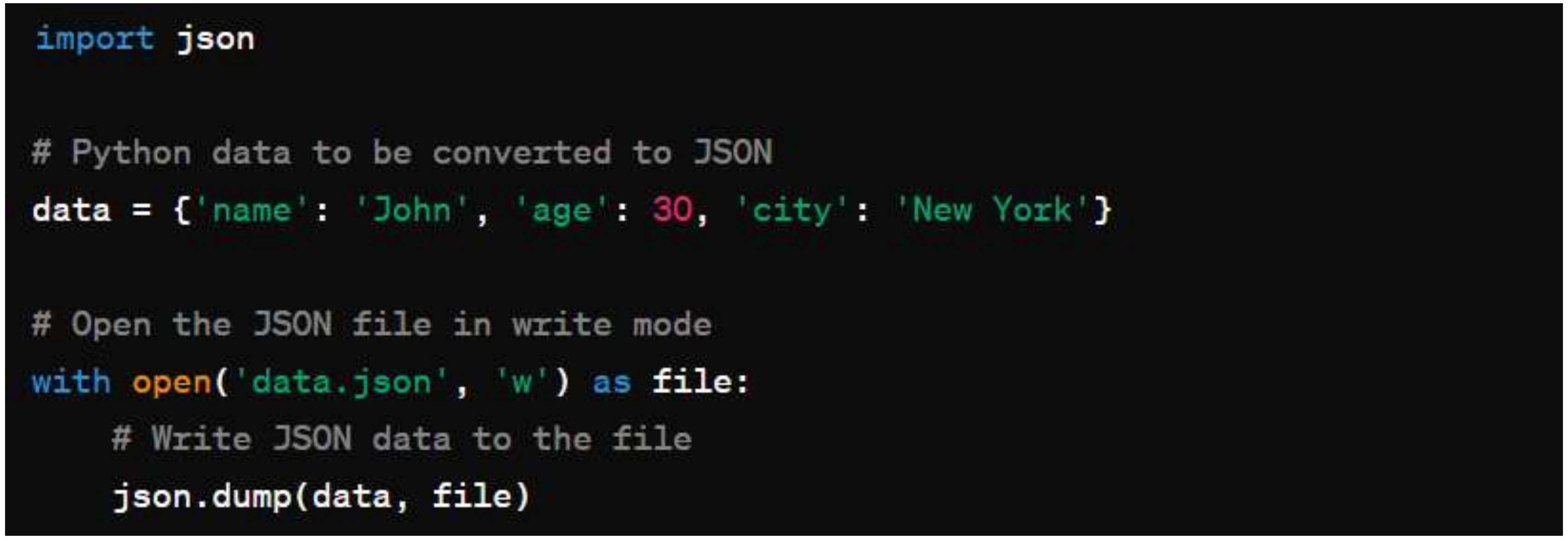


```python
import json

# Open the JSON file in read mode
with open('data.json', 'r') as file:
    # Load JSON data from the file
    data = json.load(file)
    print(data)
```

JSON Module and I/O Functions

3. **Writing to JSON**:

Use **json.dump(data, file)** or **json.dumps(data)** functions to write Python data to a file object as JSON or to return a JSON-formatted string, respectively.



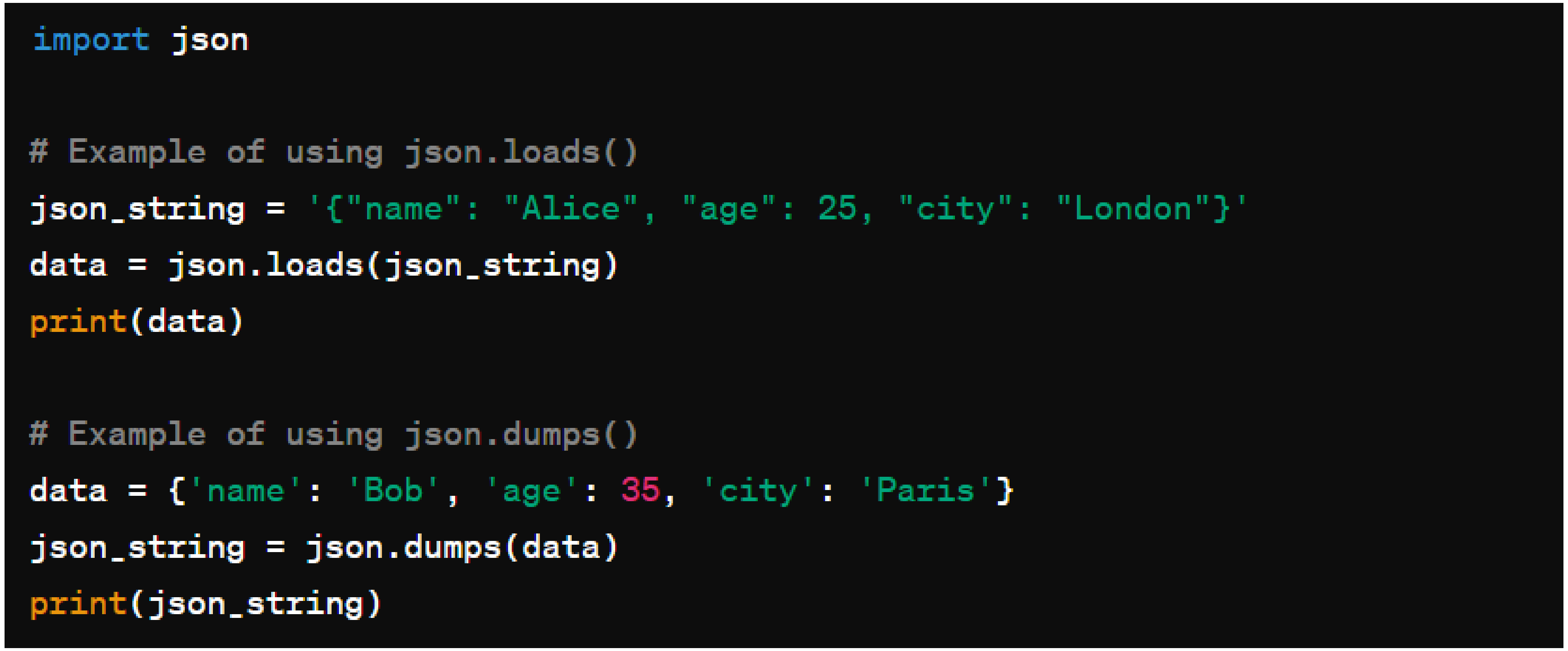
```python
import json

# Python data to be converted to JSON
data = {'name': 'John', 'age': 30, 'city': 'New York'}

# Open the JSON file in write mode
with open('data.json', 'w') as file:
    # Write JSON data to the file
    json.dump(data, file)
```

JSON Module and I/O Functions

4. **I/O Functions**:

   - **json.load(file)**: Loads JSON data from a file object.
   - **json.loads(json_string)**: Loads JSON data from a JSON-formatted string.
   - **json.dump(data, file)**: Writes JSON data to a file object.
   - **json.dumps(data)**: Returns a JSON-formatted string representing the Python data.

JSON Module and I/O Functions



```python
import json

# Example of using json.loads()
json_string = '{"name": "Alice", "age": 25, "city": "London"}'
data = json.loads(json_string)
print(data)

# Example of using json.dumps()
data = {'name': 'Bob', 'age': 35, 'city': 'Paris'}
json_string = json.dumps(data)
print(json_string)
```

JSON Module and I/O Functions

- **json.loads()** function parses a JSON-formatted string and returns a Python data structure.
- **json.dumps()** function serializes a Python data structure into a JSON-formatted string.

The JSON module in Python provides a convenient way to work with JSON data, which is commonly used for data interchange between web servers and clients, as well as for storing and sharing structured data.