Module 7 –
Libraries in Python

Intro to Libraries in Python

Libraries in Python are collections of pre-written code modules that provide a wide range of functionalities to simplify development tasks.

These libraries are developed and maintained by the Python community and cover various domains such as data manipulation, scientific computing, web development, machine learning, natural language processing, and more.

Here's a breakdown of what libraries are, why we use them, and where to use them:

What are Libraries?

Libraries in Python are essentially collections of modules containing reusable functions, classes, and constants.

They are designed to address common programming tasks and provide efficient solutions to complex problems.

Libraries encapsulate code written by other developers, allowing you to leverage their expertise and save time by not having to reinvent the wheel.

Why we use them?

Productivity: Libraries enable developers to write less code by providing pre-built solutions for common tasks. This boosts productivity and reduces development time.

Efficiency: Libraries are typically optimized for performance and reliability, allowing developers to focus on solving higher-level problems without worrying about low-level implementation details.

Why we use them?

Standardization: Libraries often establish best practices and standardize approaches to common problems, promoting consistency and maintainability in codebases.

Specialized Functionality: Many libraries provide specialized functionality that would be difficult or time-consuming to implement from scratch, such as machine learning algorithms, data visualization tools, or web development frameworks.

Where to use them?

Data Science and Analysis: Libraries like NumPy, Pandas, and Matplotlib are commonly used for data manipulation, analysis, and visualization tasks.

Machine Learning and Artificial Intelligence: Libraries such as TensorFlow, PyTorch, and Scikit-learn are used for building and training machine learning models.

Web Development: Frameworks like Django and Flask are popular for building web applications, while libraries like Requests and Beautiful Soup are used for web scraping and interacting with web services.

Where to use them?

Natural Language Processing: Libraries like NLTK and spaCy are used for processing and analyzing human language data.

Game Development: Pygame is a popular library for creating games and multimedia applications.

Computer Vision: OpenCV is widely used for tasks such as image processing, object detection, and face recognition.

Math Library

The **math** library in Python provides a set of mathematical functions and constants for performing mathematical operations. It's part of Python's standard library, so it comes pre-installed with Python and doesn't require any additional installation.

Features and Functions:

Basic Mathematical Functions: The **math** library provides functions for common mathematical operations such as **sqrt()** for square root, **sin()** and **cos()** for trigonometric functions, **log()** for logarithms, etc.

Math Library

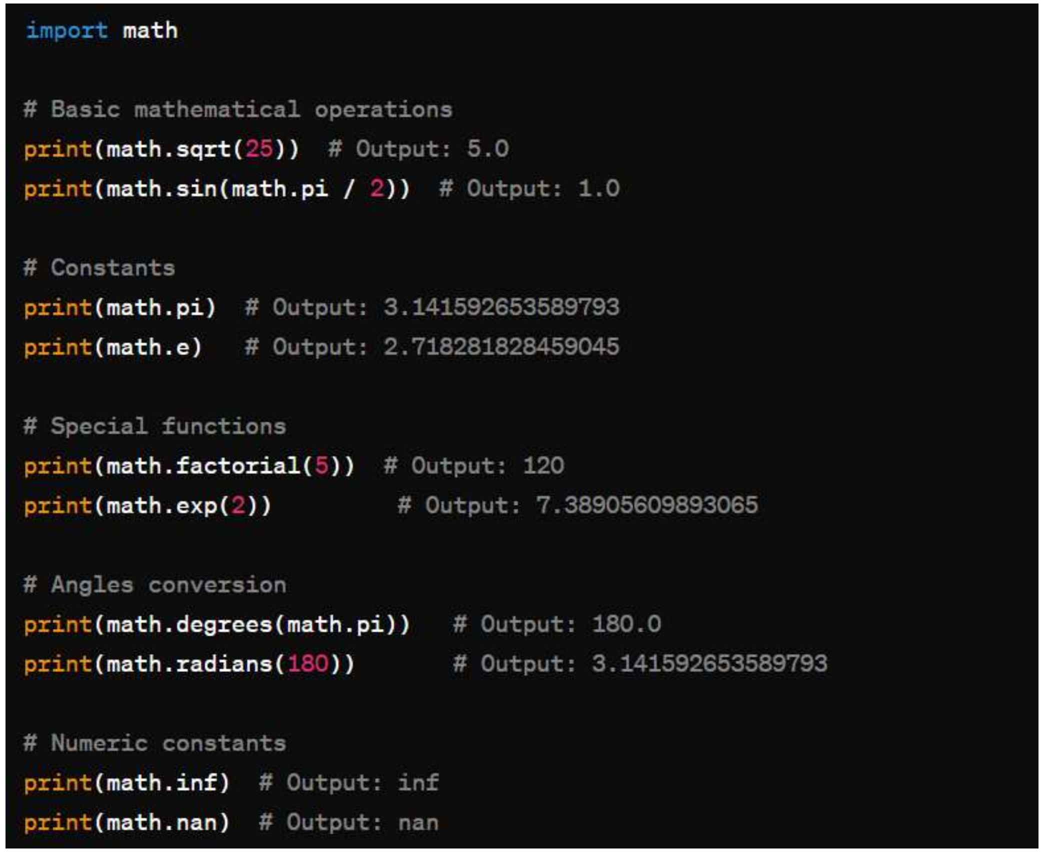
Constants: It also includes important mathematical constants like **pi** (π) and **e** (the base of natural logarithm).

Special Functions: The library includes functions for special mathematical computations like factorials (**factorial()**), exponential and logarithmic functions (**exp()**, **log()**), and more.

Angles Conversion: Functions for converting angles between degrees and radians (**degrees()** and **radians()**).

Numeric Constants: In addition to mathematical functions, the **math** library also defines some numeric constants like **inf** for positive infinity and **nan** for a NaN (Not a Number) value.

Math Library



```python
import math

# Basic mathematical operations
print(math.sqrt(25))  # Output: 5.0
print(math.sin(math.pi / 2))  # Output: 1.0

# Constants
print(math.pi)  # Output: 3.141592653589793
print(math.e)   # Output: 2.718281828459045

# Special functions
print(math.factorial(5))  # Output: 120
print(math.exp(2))         # Output: 7.38905609893065

# Angles conversion
print(math.degrees(math.pi))   # Output: 180.0
print(math.radians(180))       # Output: 3.141592653589793

# Numeric constants
print(math.inf)  # Output: inf
print(math.nan)  # Output: nan
```

Numpy Library

NumPy is a powerful library for numerical computing in Python. It provides support for multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays efficiently. NumPy is not part of the standard library and needs to be installed separately, commonly done using tools like pip.

Features and Functions:

Arrays: NumPy's main object is the **ndarray** (n-dimensional array), which is a homogeneous collection of elements with fixed size.

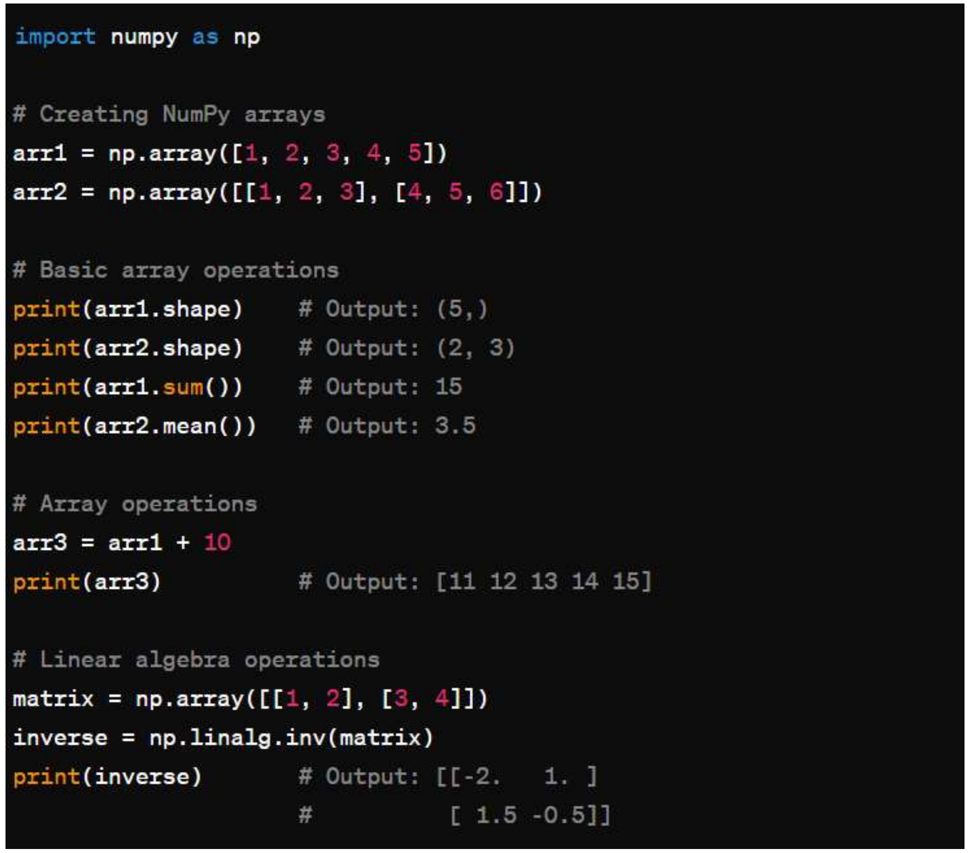
Numpy Library

Array Operations: NumPy provides a wide range of functions for performing operations on arrays, including element-wise operations, linear algebra operations, Fourier transforms, random number generation, and more.

Efficiency: NumPy arrays are significantly more efficient than Python lists for numerical operations due to their fixed size and homogeneous data types. They also allow vectorized operations, which can be much faster than using loops in Python.

Integration: NumPy integrates seamlessly with other libraries in the scientific Python ecosystem, such as SciPy (scientific computing), Matplotlib (data visualization), and scikit-learn (machine learning).

Numpy Library



```python
import numpy as np

# Creating NumPy arrays
arr1 = np.array([1, 2, 3, 4, 5])
arr2 = np.array([[1, 2, 3], [4, 5, 6]])

# Basic array operations
print(arr1.shape)     # Output: (5,)
print(arr2.shape)     # Output: (2, 3)
print(arr1.sum())     # Output: 15
print(arr2.mean())    # Output: 3.5

# Array operations
arr3 = arr1 + 10
print(arr3)           # Output: [11 12 13 14 15]

# Linear algebra operations
matrix = np.array([[1, 2], [3, 4]])
inverse = np.linalg.inv(matrix)
print(inverse)        # Output: [[-2.   1. ]
                      #          [ 1.5 -0.5]]
```

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.

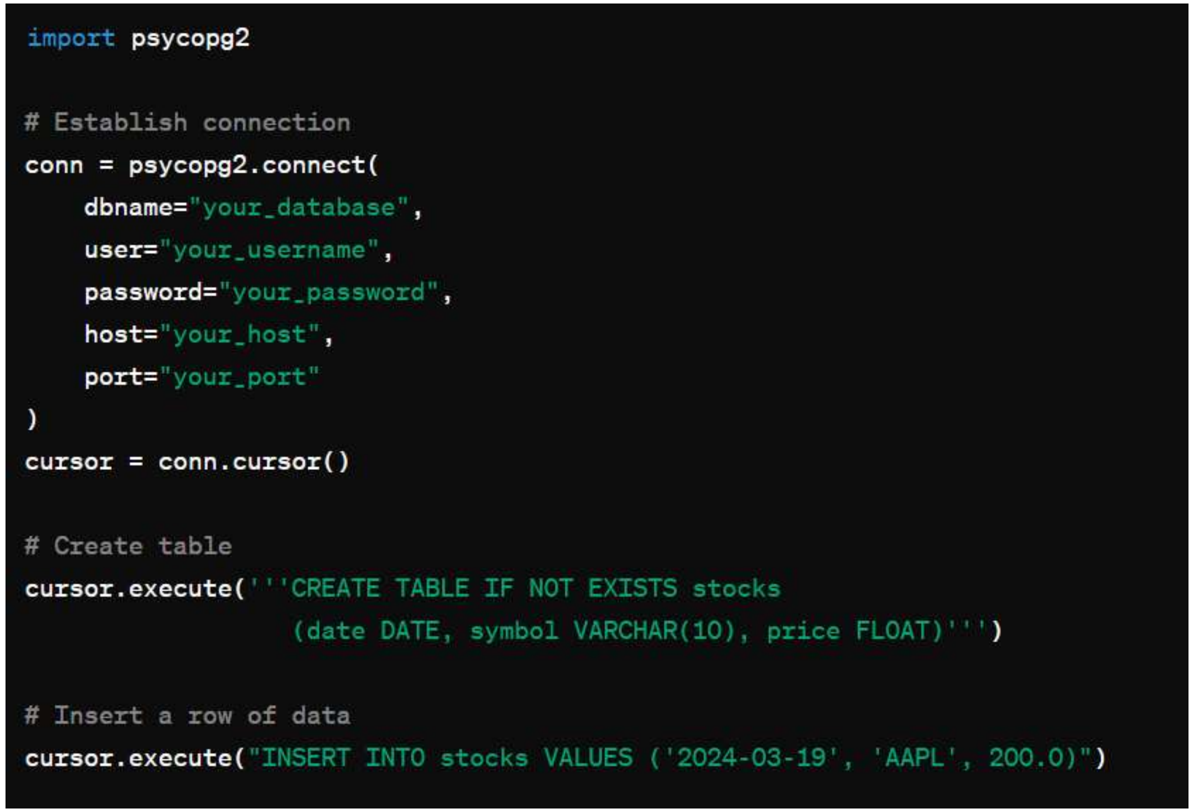
Close Connection: Once you're done with the database operations, close the connection to release resources and avoid memory leaks.

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



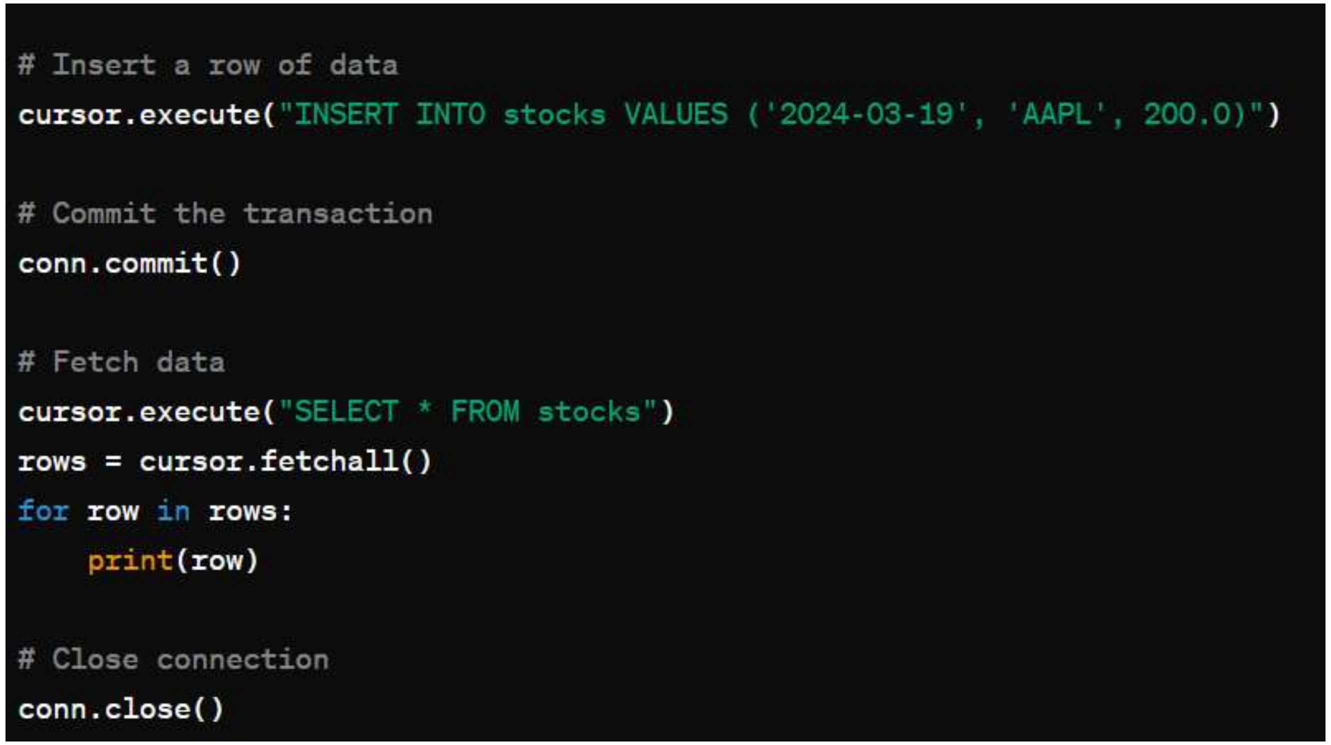
```python
import psycopg2

# 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()
```

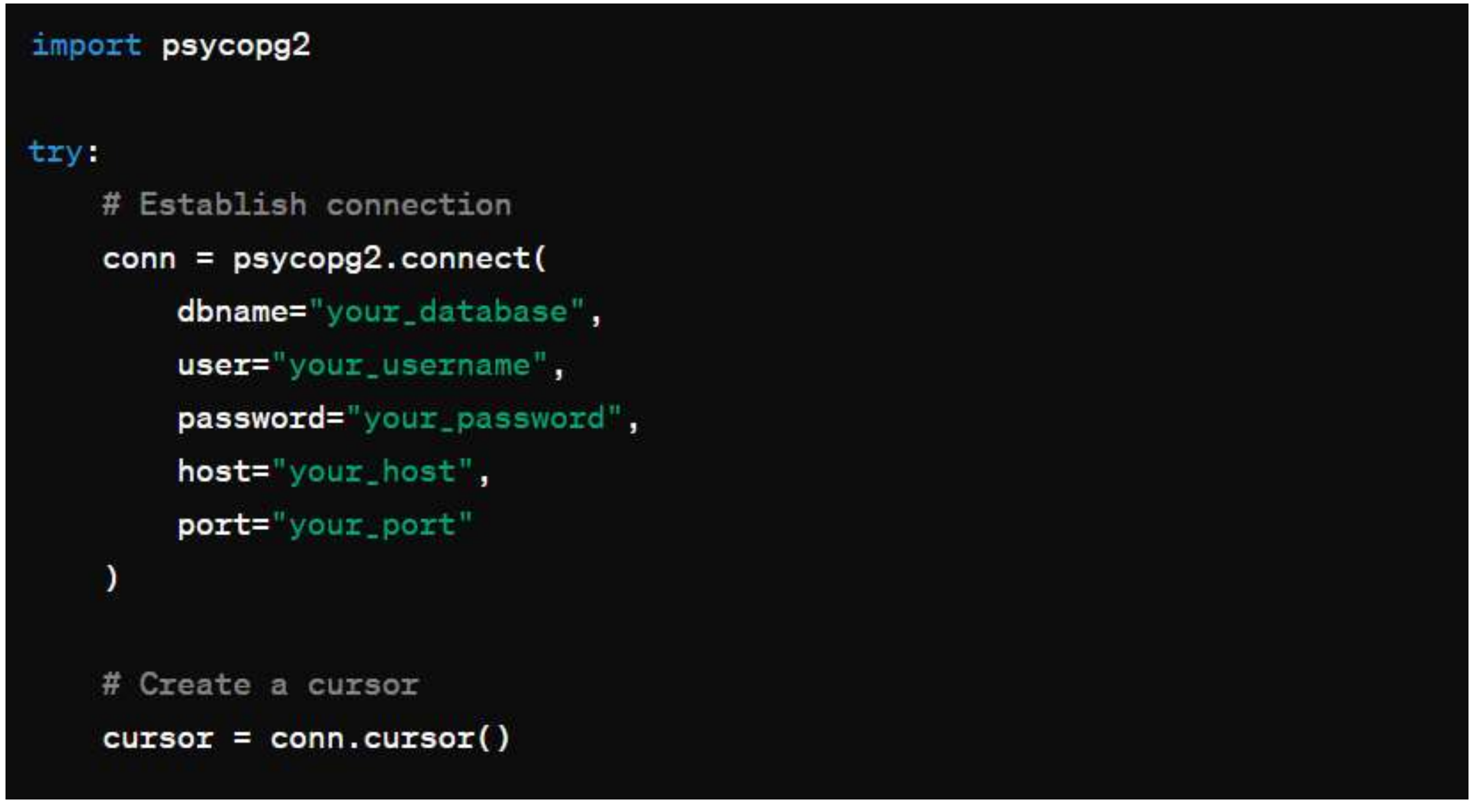
Interact with Postgres

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

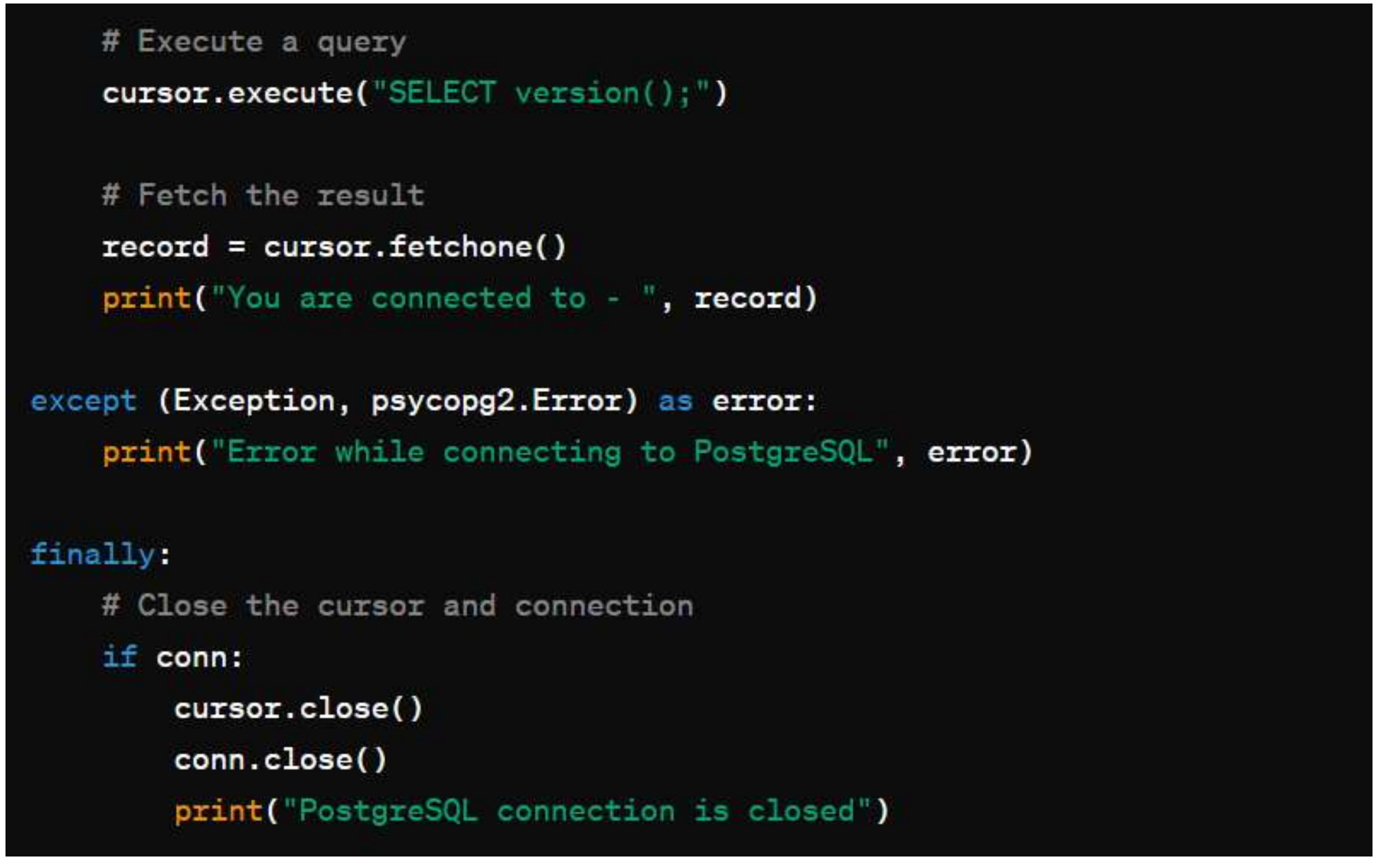


```
import psycopg2

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.

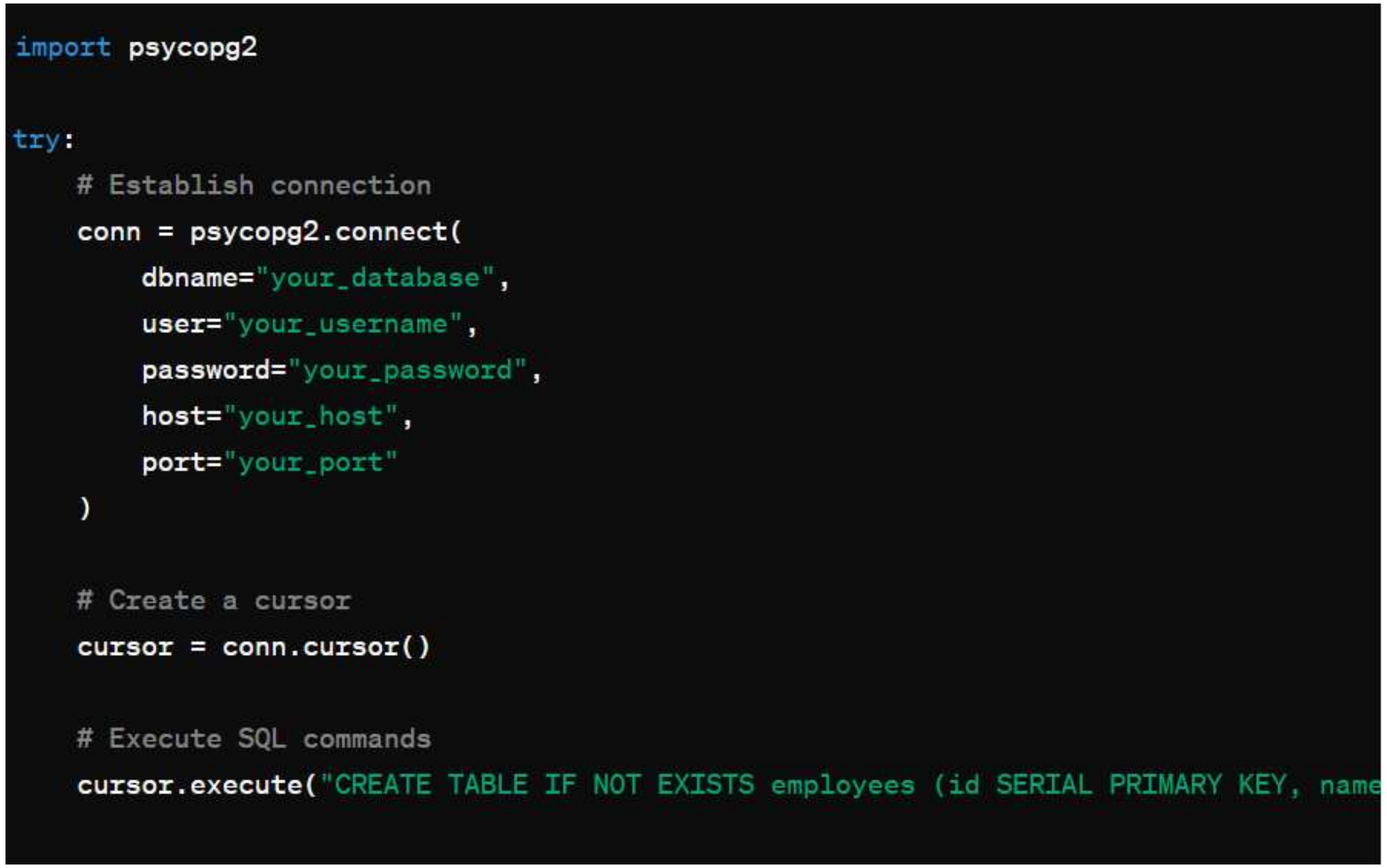
In the **finally** block, we ensure that the cursor and connection are properly closed using **cursor.close()** and **conn.close()** respectively.

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



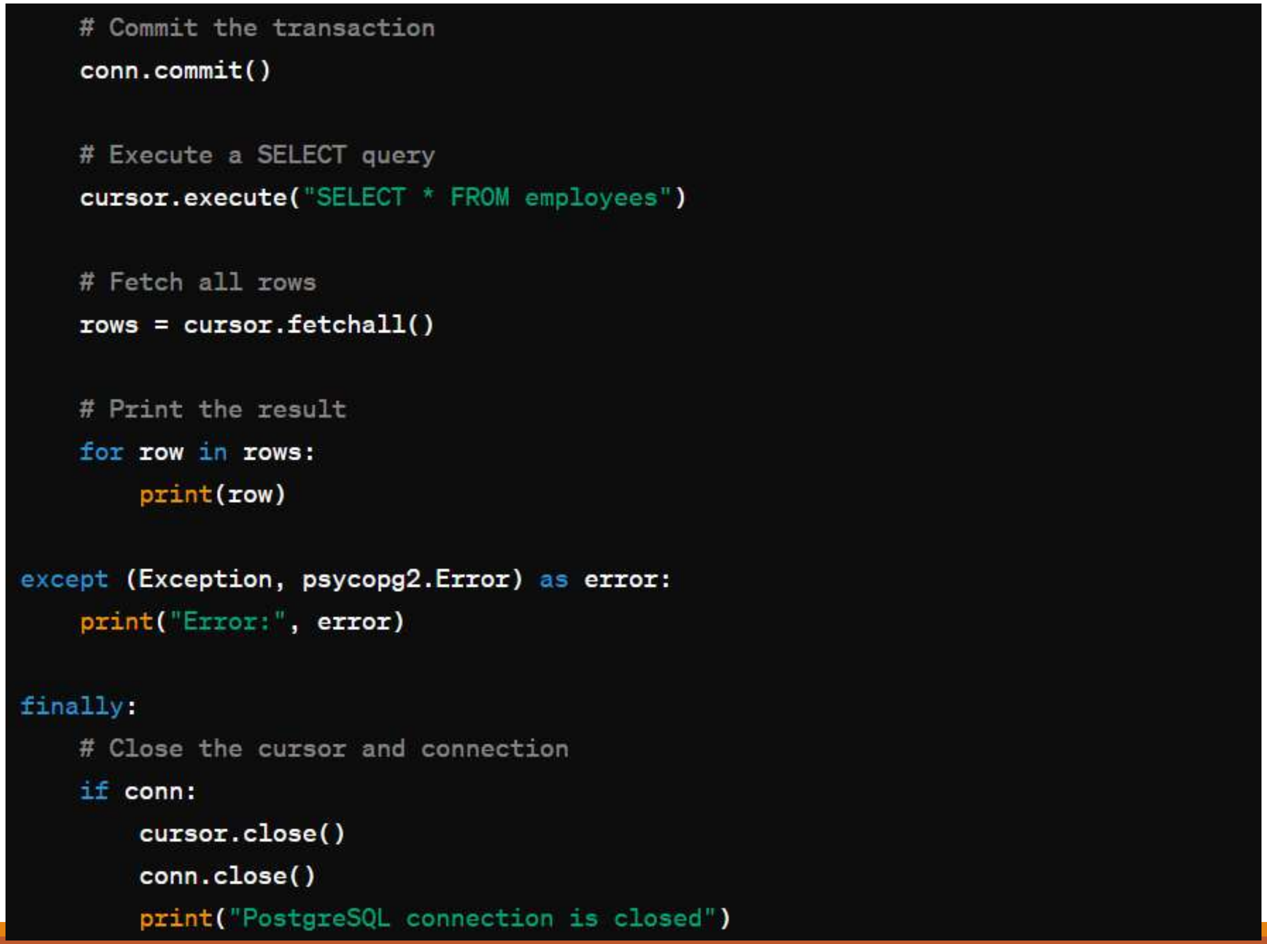
```
import psycopg2

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.

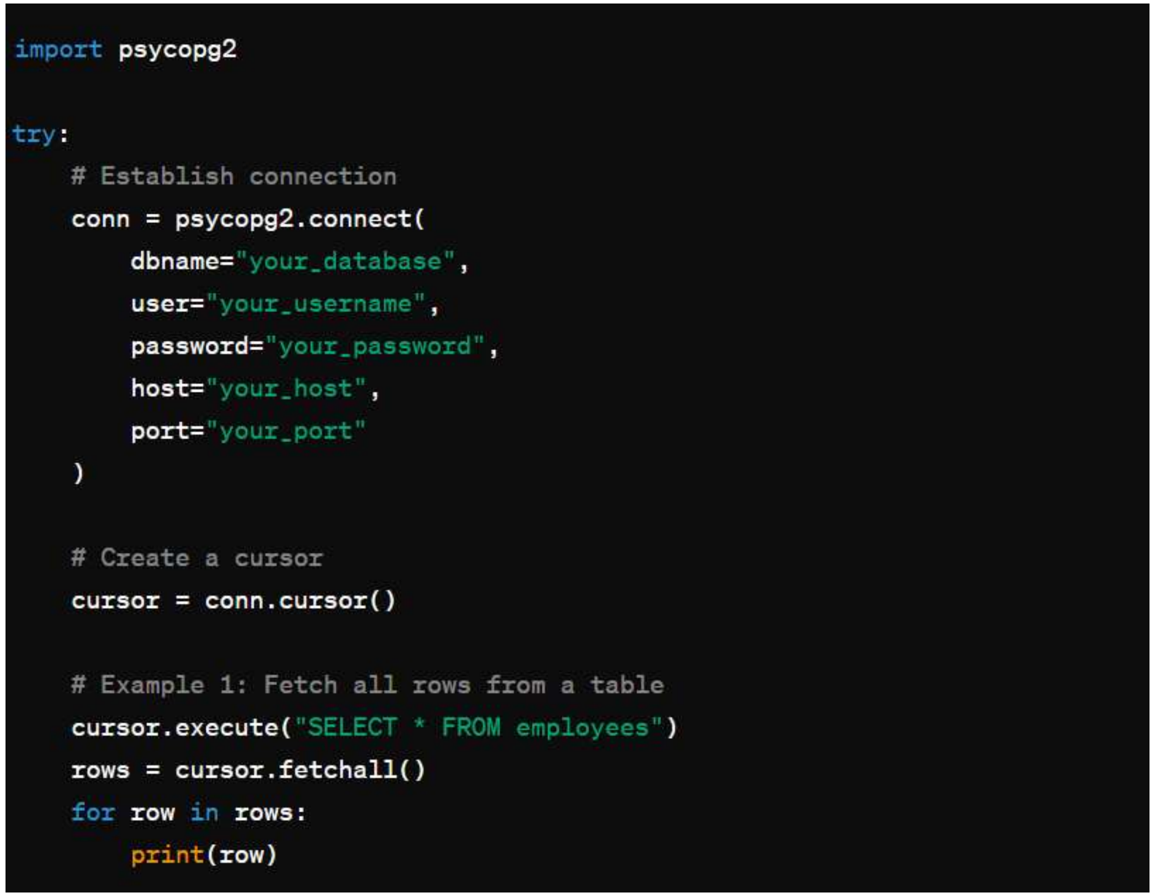
Execute Query: Execute your SQL query using the cursor's **execute()** method.

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



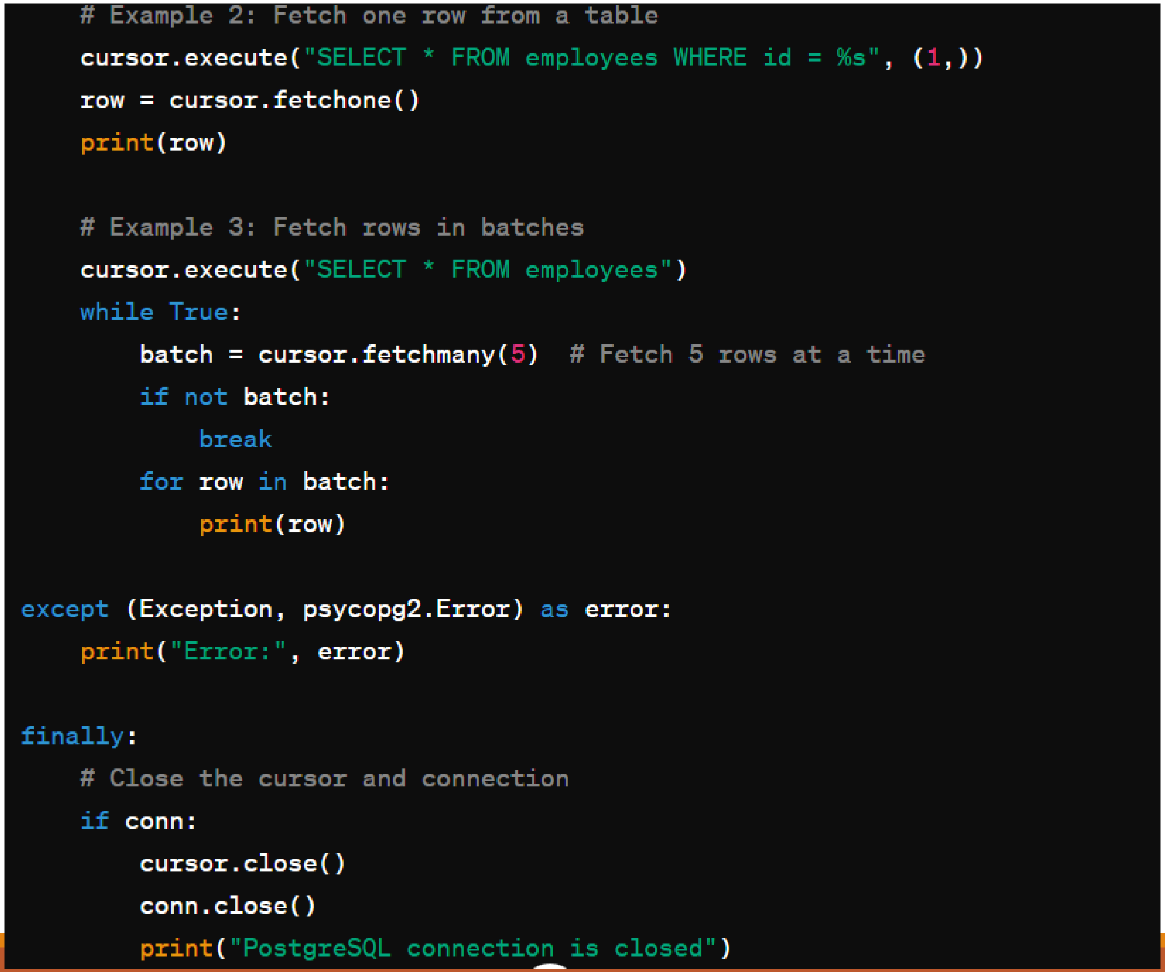
```
import psycopg2

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.

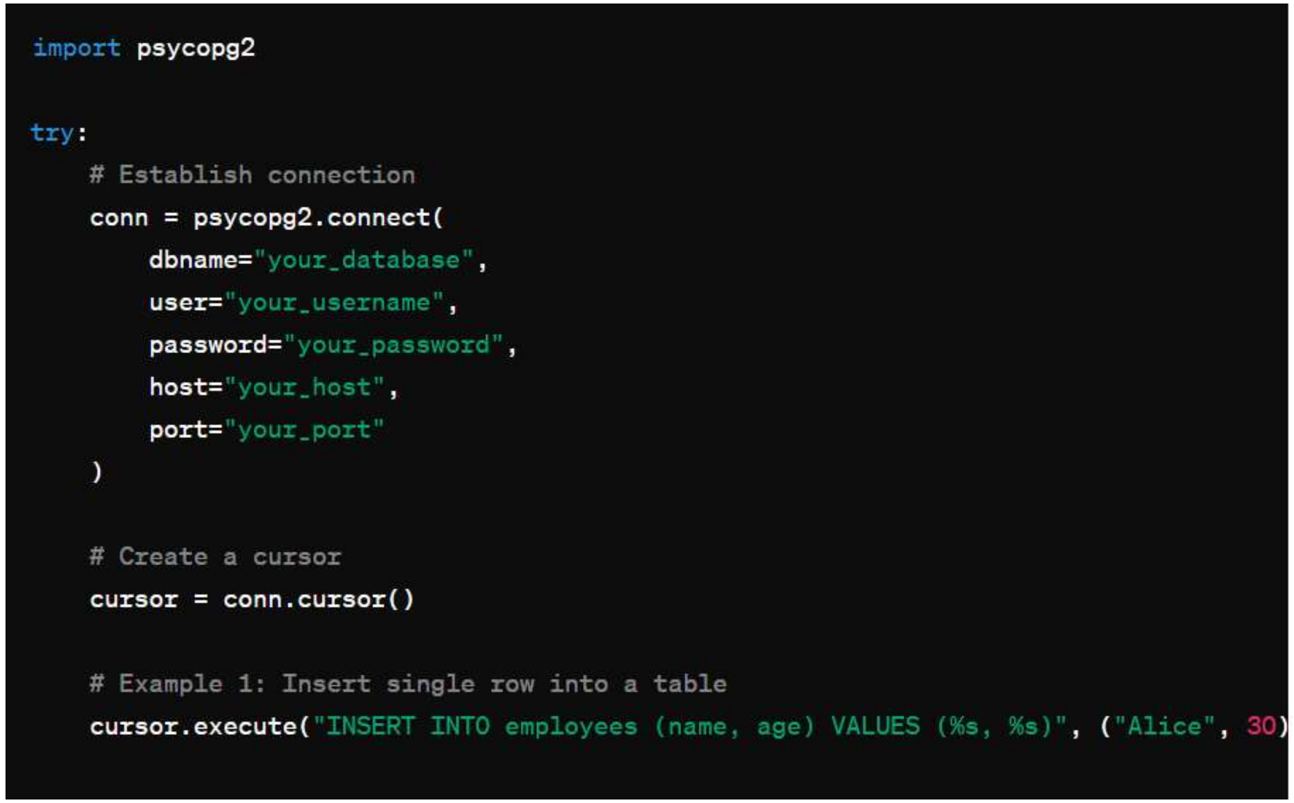
Execute INSERT Queries: Execute **INSERT** queries to add data into the tables.

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



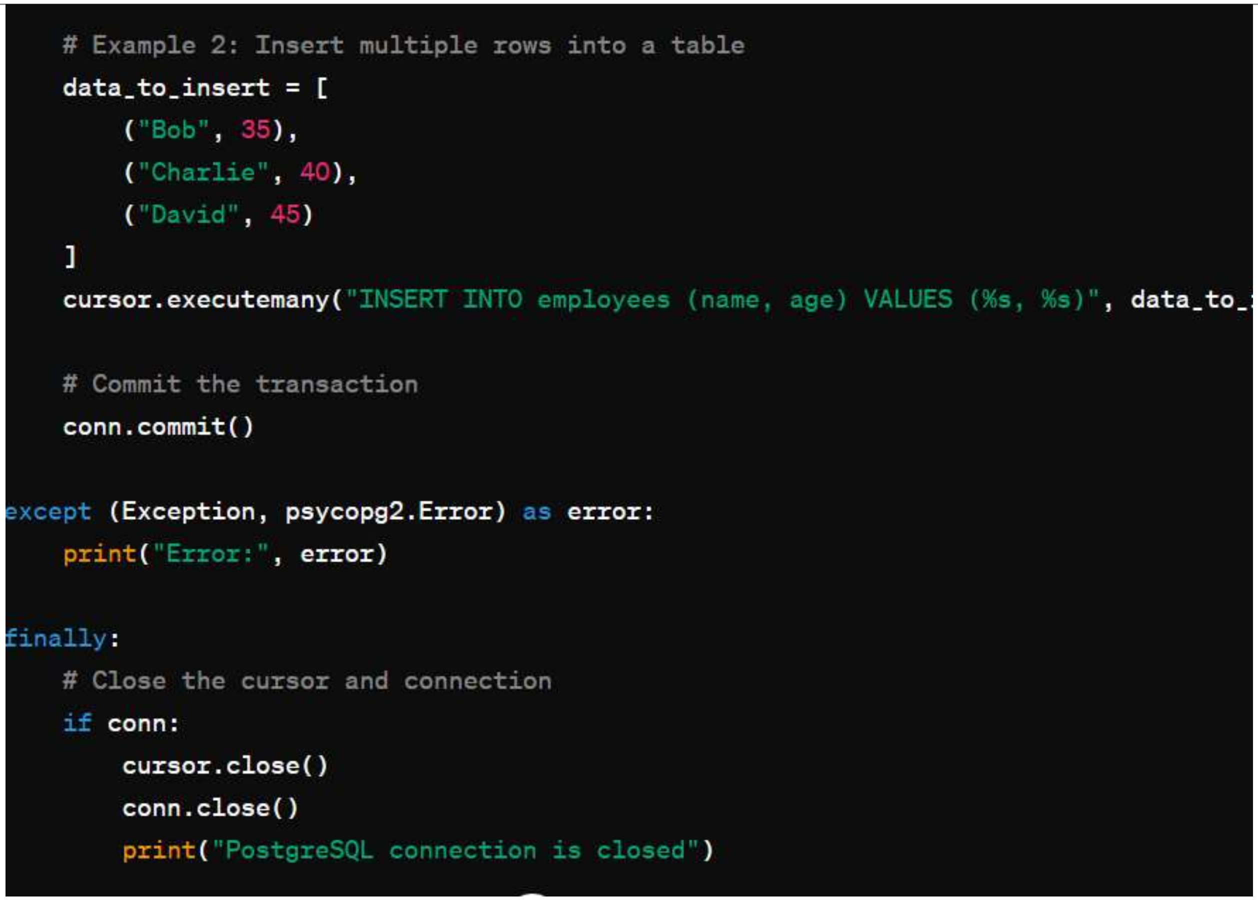
```
import psycopg2

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_

    # 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.