Module 12 - Building Database Apps with PostgreSQL & Python

Welcome to Module 12! This module will introduce you to the world of databases, specifically PostgreSQL, and teach you how to build robust applications by connecting your Python code to a powerful relational database. Data is at the heart of most modern applications and understanding how to manage it is a critical skill.

Chapter 1: Introduction to Data

Before we dive into databases, let's clarify what we mean by "data."

1.1 What is Data?

- **Definition:** Data refers to raw facts, figures, statistics, or items of information that, in their initial form, may not convey much meaning on their own.
- **Examples:** A single number like `25`, a word like `Hyderabad`, a date like `2025-07-02`, or a name like `Alice`.
- **Data vs. Information:** When data is processed, organized, structured, or presented in a given context to make it meaningful or useful, it becomes **information**.
    - **Data:** `Alice`, `30`, `Hyderabad`
    - **Information:** "Alice is 30 years old and lives in Hyderabad."

1.2 Types of Data

Data can broadly be categorized into:

1. **Structured Data:**
    - Highly organized and formatted in a way that makes it easily searchable and manageable.
    - Fits neatly into a fixed schema (like rows and columns in a table).
    - **Examples:** Data in relational databases (SQL databases), spreadsheets (Excel), CSV files.
    - **Characteristics:** Defined data types, clear relationships.
2. **Unstructured Data:**
    - Lacks a predefined format or organization. It cannot be easily put into a traditional row-and-column database.
    - **Examples:** Text documents, emails, social media posts, audio files, video files, images.
    - **Characteristics:** Difficult to search, requires advanced analytics tools.
3. **Semi-structured Data:**
    - Has some organizational properties, but it's not strictly rigid like structured data. It contains tags or other markers to identify distinct elements and impose hierarchy.
    - **Examples:** XML, JSON files, NoSQL databases (like MongoDB documents).

    - **Characteristics:** Flexible schema, easier to process than unstructured data, but less rigid than structured.

1.3 Importance of Data Management

In today's digital world, data is constantly being generated. Effective data management is crucial because it ensures:

- **Data Integrity:** Accuracy, consistency, and reliability of data.
- **Data Security:** Protecting data from unauthorized access or corruption.
- **Efficient Retrieval:** Quickly finding the specific information you need.
- **Scalability:** Ability to handle growing volumes of data.
- **Decision Making:** Providing accurate information for business and analytical insights.

Chapter 2: Introduction to Database

Managing large amounts of data manually (e.g., in text files or simple spreadsheets) quickly becomes impractical. This is where databases come in.

2.1 What is a Database?

- A **database** is an organized collection of data, typically stored and accessed electronically from a computer system.
- More precisely, it's a collection of related tables, queries, reports, and other objects that are created and managed by a **Database Management System (DBMS)**.
- The DBMS is the software that interacts with the user, applications, and the database itself to capture and analyse data.

2.2 Purpose of Databases

The primary purposes of a database are:

- **Storage:** To store large volumes of data persistently.
- **Retrieval:** To allow efficient searching and retrieval of specific data.
- **Management:** To organize, update, delete, and control access to data.
- **Integrity:** To enforce rules and constraints to ensure data consistency and accuracy.
- **Concurrency:** To allow multiple users or applications to access and modify data simultaneously without conflicts.

2.3 Types of Databases

While there are many types, the most common distinction is between:

1. **Relational Databases (SQL Databases - RDBMS):**
    - Data is organized into one or more tables (relations), with rows and columns.
    - Tables are related to each other based on common fields.
    - Use **SQL (Structured Query Language)** for defining and manipulating data.

    - Adhere to **ACID properties** (Atomicity, Consistency, Isolation, Durability) for transaction reliability.
    - **Examples:** PostgreSQL, MySQL, Oracle, SQL Server, SQLite.
2. **NoSQL Databases (Non-relational Databases):**
    - Designed for specific data models and have flexible schemas for handling modern applications' varying data access patterns.
    - Do not primarily use SQL.
    - **Examples:** Document databases (MongoDB), Key-value stores (Redis), Column-family stores (Cassandra), Graph databases (Neo4j).

In this module, we will focus on Relational Databases, specifically PostgreSQL.

2.4 Key Concepts in Relational Databases (RDBMS)

- **Table (Relation):** A collection of related data organized in rows and columns. Similar to a spreadsheet. Each table has a unique name.
    - **Example:** A `Students` table, a `Courses` table.
- **Row (Record / Tuple):** A single entry in a table, representing a single set of related data.
    - **Example:** In a `Students` table, one row would represent "Alice, 20, Computer Science."
- **Column (Field / Attribute):** A vertical set of data that contains values of a single data type for all records in the table. Each column has a unique name.
    - **Example:** In a `Students` table, `Name`, `Age`, `Major` would be columns.
- **Primary Key (PK):**
    - A column (or set of columns) that uniquely identifies each row in a table.
    - No two rows can have the same primary key value.
    - Cannot contain `NULL` values.
    - **Example:** `student_id` in a `Students` table.
- **Foreign Key (FK):**
    - A column (or set of columns) in one table that refers to the primary key in another table.
    - Establishes a link or relationship between two tables.
    - **Example:** `course_id` in a `Students` table that refers to `course_id` in a `Courses` table, indicating which course a student is enrolled in.

Chapter 3: Introduction to PostgreSQL

Now that we understand databases, let's look at the specific RDBMS we'll be using: PostgreSQL.

3.1 What is PostgreSQL?

- **PostgreSQL** (often pronounced "Post-Gres-Q-L" or just "Postgres") is a powerful, open-source object-relational database system (ORDBMS).
- It is known for its strong reputation for reliability, feature robustness, and performance.

- It has been under active development for more than 35 years and is backed by a large, dedicated community of developers.

3.2 Key Features and Advantages

- **Open Source:** Free to use, modify, and distribute, making it very popular for startups and large enterprises alike.
- **Robustness & Reliability:** Adheres strictly to ACID (Atomicity, Consistency, Isolation, Durability) properties, ensuring data integrity even in the event of system failures.
- **Extensibility:** Highly extensible; you can define your own data types, functions, operators, and even programming languages.
- **Advanced Features:** Supports complex queries, full-text search, JSON, XML, geographic data (PostGIS), and more.
- **Concurrency Control:** Uses Multi-Version Concurrency Control (MVCC) to allow many users to work with the database concurrently without interfering with each other.
- **Community Support:** A large and active community provides extensive documentation, forums, and support.
- **Cross-Platform:** Runs on all major operating systems, including Linux, Windows, and macOS.

3.3 Use Cases

PostgreSQL is a versatile database used in a wide range of applications:

- **Web Applications:** Backend for dynamic websites and web services (often with frameworks like Django, Flask).
- **Data Warehousing & Analytics:** Its robust features make it suitable for large-scale data storage and complex analytical queries.
- **Geographic Information Systems (GIS):** With PostGIS extension, it's a leading database for spatial data.
- **Financial Applications:** Its strong ACID compliance is critical for transactional systems.
- **Scientific Data:** Used for managing complex datasets in research.

Chapter 4: Installing PostgreSQL

Before we can interact with PostgreSQL, we need to install it on your system. The installation process varies slightly depending on your operating system.

4.1 General Installation Steps (Conceptual)

While exact steps differ, the general process involves:

1. **Download:** Go to the official PostgreSQL website (www.postgresql.org/download/) and select your operating system.

2. **Installer:** Download the "EDB Interactive installer" (or similar graphical installer for your OS).
3. **Run Installer:**
   - **Installation Directory:** Choose where to install PostgreSQL.
   - **Select Components:** Make sure to select **"PostgreSQL Server"** and **"pgAdmin 4"** (the graphical administration tool). You might also want "Command Line Tools."
   - **Data Directory:** Choose where your database files will be stored.
   - **Password for `postgres` user:** This is crucial! You will be asked to set a password for the default `postgres` superuser. **Remember this password!**
   - **Port:** The default port is `5432`. You can usually leave this as default unless it conflicts with other software.
   - **Locale:** Choose your preferred language/region settings.
4. **Finish Installation:** The installer will complete the setup and start the PostgreSQL server service.

4.2 Verifying Installation

After installation, you can verify if PostgreSQL is running and accessible:

1. **Check Service Status:**
   - **Windows:** Open "Services" (search for it in the Start Menu) and look for a service named `postgresql-x64-xx` (where xx is the version number). Ensure it's "Running."
   - **macOS/Linux:** Commands like `sudo service postgresql status` or `pg_isready` (might vary by distribution).
2. **Use `psql` (Command Line Tool):**
   - `psql` is the command-line interface for PostgreSQL.
   - Open your terminal/command prompt.
   - Try to connect as the `postgres` user (the default superuser):

     Bash

     ```
     psql -U postgres
     ```

   - It will ask for the password you set during installation.
   - If successful, you'll see a `postgres=#` prompt. This means you are connected to the `postgres` database as the `postgres` user.
   - Type `\q` and press Enter to exit `psql`.
3. **Use `pgAdmin 4` (Graphical Tool):**
   - Launch `pgAdmin 4` (usually available in your applications menu).
   - It will likely prompt you to set a master password for `pgAdmin` itself.
   - You should see a "Servers" section. Expand it, and you'll likely see "PostgreSQL x.x". Double-click it.
   - It will ask for the `postgres` user's password. Enter it.
   - If successful, you can browse your databases, tables, etc., graphically.

Chapter 5: Creating a Database (SQL)

Once PostgreSQL is installed, your first step is to create a new database where your application's data will reside. We'll do this using SQL commands via the `psql` command-line tool.

5.1 Connecting via `psql`

To execute SQL commands, you first need to connect to the PostgreSQL server.

- Open your terminal or command prompt.
- Connect as the `postgres` superuser (or any other user with database creation privileges):

  Bash

  ```
  psql -U postgres -h localhost -p 5432
  # -U: specifies username (postgres)
  # -h: specifies host (localhost for your machine)
  # -p: specifies port (default 5432)
  ```

- Enter the `postgres` user's password when prompted.
- You should now be at the `postgres=#` prompt.

5.2 SQL Command: `CREATE DATABASE`

This command is used to create a new, empty database.

- **Syntax:**

  SQL

  ```
  CREATE DATABASE database_name;
  ```

  - Database names are typically lowercase and can contain letters, numbers, and underscores.
  - SQL commands usually end with a semicolon `;`.
- **Example:** Let's create a database for our Python application, named `myapp_db`.

  SQL

  ```
  CREATE DATABASE myapp_db;
  ```

  You should see `CREATE DATABASE` as output if successful.

5.3 Listing Databases

To confirm your database has been created, you can list all databases on the server.

- **`psql` Meta-command:**

SQL

```
\l
# or
\list
```

This will display a list of all databases, including `myapp_db`.

5.4 Connecting to a Database

Once a database is created, you usually want to connect to it specifically to work with its tables.

- **`psql` Meta-command:**

  SQL

  ```
  \c myapp_db
  # or
  \connect myapp_db
  ```

  You should see output like `You are now connected to database "myapp_db" as user "postgres".` Your prompt will change to `myapp_db=#`, indicating you're now working within that database.

- **Exit `psql`:**

  SQL

  ```
  \q
  ```

Chapter 6: Deleting a Database (SQL)

Sometimes, you might need to remove an existing database. This is done using the `DROP DATABASE` command.

6.1 SQL Command: `DROP DATABASE`

This command permanently deletes a database and all its contents. Use with extreme caution!

- **Syntax:**

  SQL

  ```
  DROP DATABASE database_name;
  ```

- **Example:** Let's assume you created a test database called `test_db` and now want to remove it.

- First, connect to a *different* database (e.g., `postgres` or any other database) because you cannot drop a database you are currently connected to.

Bash

```
psql -U postgres
# password prompt
# Now you are connected to postgres=#
```

- Execute the `DROP DATABASE` command:

SQL

```
DROP DATABASE test_db;
```

You should see `DROP DATABASE` as output if successful.

6.2 Important Warning

- **You CANNOT drop a database if you are currently connected to it.** If you try, you'll get an error like "ERROR: cannot drop the currently open database".
- Always connect to a different database (like the default `postgres` database) before attempting to drop the one you intend to remove.
- Dropping a database is irreversible and will delete all tables, data, and other objects within it.

Chapter 7: Creating Table and Adding Data (SQL)

Now that we have a database (`myapp_db`), let's create a table within it and populate it with some initial data.

7.1 Connecting to the Target Database

Before creating tables, ensure you are connected to the correct database.

- Open `psql` and connect to `myapp_db`:

  Bash

  ```
  psql -U postgres -h localhost -p 5432 myapp_db
  # password prompt
  # You should be at myapp_db=#
  ```

7.2 SQL Command: `CREATE TABLE`

This command defines the structure of your table, including column names, data types, and constraints.

- **Common SQL Data Types for PostgreSQL:**

    - `INT`: Integer numbers.
    - `SERIAL`: Auto-incrementing integer, commonly used for primary keys.
    - `VARCHAR(n)`: Variable-length string, `n` is the maximum length.
    - `TEXT`: Variable-length string, no explicit length limit.
    - `BOOLEAN`: `TRUE` or `FALSE`.
    - `DATE`: Date (e.g., 'YYYY-MM-DD').
    - `TIMESTAMP`: Date and time.
    - `NUMERIC(p, s)` or `DECIMAL(p, s)`: Exact numeric, `p` is precision (total digits), `s` is scale (digits after decimal point).
- **Common Constraints:**
    - `PRIMARY KEY`: Uniquely identifies each row; implies `NOT NULL` and `UNIQUE`.
    - `NOT NULL`: Ensures a column cannot have `NULL` values.
    - `UNIQUE`: Ensures all values in a column are different.
    - `DEFAULT value`: Provides a default value if none is specified during insertion.
- **Example: Creating a `Users` table** Let's create a table called `Users` to store user information.

  SQL

  ```
  CREATE TABLE Users (
      id SERIAL PRIMARY KEY,            -- Auto-incrementing unique ID
  (Primary Key)
      username VARCHAR(50) UNIQUE NOT NULL, -- Unique username, cannot
  be empty
      email VARCHAR(100) UNIQUE NOT NULL, -- Unique email, cannot be
  empty
      age INT,                          -- Integer for age (can be
  NULL)
      created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP -- Automatically
  set creation time
  );
  ```

  You should see `CREATE TABLE` as output.

- **Verify Table Creation:**

  SQL

  ```
  \dt
  ```

  This meta-command lists tables in the current database. You should see `users`. To see the table structure: `\d users`

7.3 SQL Command: `INSERT INTO`

This command is used to add new rows (records) into a table.

- **Syntax for single row:**

  SQL

  ```
  INSERT INTO table_name (column1, column2, ...)
  ```

```
VALUES (value1, value2, ...);
```

  - You can omit the column list if you provide values for all columns in the exact order they appear in the table definition.
- **Example: Adding data to `Users` table**

  SQL

  ```
  INSERT INTO Users (username, email, age)
  VALUES ('alice_smith', 'alice@example.com', 30);
  ```

  SQL

  ```
  INSERT INTO Users (username, email, age)
  VALUES ('bob_jones', 'bob@example.com', 25);
  ```

  SQL

  ```
  -- If omitting column names, provide values for all columns in order
  (id, username, email, age, created_at)
  -- This works because 'id' is SERIAL (auto-generated) and
  'created_at' has a DEFAULT
  INSERT INTO Users VALUES (DEFAULT, 'charlie_brown',
  'charlie@example.com', 35, DEFAULT);
  ```

  You should see `INSERT 0 1` as output for each successful insertion.

Chapter 8: Retrieving Data from Database and Deleting Contents in the Table (SQL)

Now that we have data, let's learn how to view it and how to remove it.

8.1 SQL Command: `SELECT`

The `SELECT` statement is used to retrieve data from one or more tables. It's the most frequently used SQL command.

- **Basic Selection: Select All Columns**

  SQL

  ```
  SELECT * FROM Users;
  ```

  This will retrieve all columns (`*`) for all rows from the `Users` table.

- **Select Specific Columns**

  SQL

```
SELECT username, email FROM Users;
```

This will retrieve only the `username` and `email` columns for all rows.

- **Filtering Data with `WHERE` Clause** The `WHERE` clause is used to specify a condition to filter the rows returned by the `SELECT` statement.

  SQL

  ```
  SELECT * FROM Users WHERE age > 28;
  ```

  This will retrieve all columns for users whose age is greater than 28.

  SQL

  ```
  SELECT username, email FROM Users WHERE username = 'alice_smith';
  ```

  This retrieves username and email for the user with that specific username.

8.2 SQL Command: `DELETE FROM`

The `DELETE FROM` statement is used to remove existing rows from a table.

- **Deleting Specific Rows:** To delete specific rows, you must use a `WHERE` clause. Without it, you will delete *all* rows!

  SQL

  ```
  DELETE FROM Users WHERE username = 'bob_jones';
  ```

  This will delete the row where the `username` is 'bob_jones'. You'll see `DELETE 1` if one row was deleted.

  Verify Deletion:

  SQL

  ```
  SELECT * FROM Users;
  ```

  You'll notice 'bob_jones' is no longer there.

- **Deleting All Contents (but keeping table structure):** To remove all data from a table without deleting the table itself, you use `DELETE FROM` without a `WHERE` clause.

  SQL

  ```
  DELETE FROM Users;
  ```

  This will delete all rows from the `Users` table. You'll see `DELETE X` where X is the number of rows deleted.

Warning: This is a powerful command. Be absolutely sure before executing it!

Alternative (faster for full table clear): `TRUNCATE TABLE` If you want to quickly remove all rows from a table and reset auto-incrementing counters, `TRUNCATE TABLE` is often faster than `DELETE FROM` without a `WHERE` clause. It does not allow a `WHERE` clause.

SQL

```
TRUNCATE TABLE Users;
```

- **Exit `psql`:**

  SQL

  ```
  \q
  ```

Chapter 9: Setting up Virtualenv

Before we install Python libraries for database interaction, it's a best practice to set up a virtual environment.

9.1 Why Virtual Environments?

- **Isolation:** A virtual environment creates an isolated Python environment for your project. This means that any libraries you install for this project will only reside within this environment and won't interfere with other Python projects or your system's global Python installation.
- **Dependency Management:** Different projects might require different versions of the same library. Virtual environments prevent conflicts. For example, Project A might need `library_x v1.0` and Project B needs `library_x v2.0`. With virtual environments, both can coexist peacefully.
- **Cleanliness:** Keeps your global Python installation clean and free of project-specific packages.
- **Portability:** Makes it easier to share your project, as others can recreate the exact environment.

9.2 Installation (`virtualenv` tool)

`venv` is now built into Python 3.3 and later, so you usually don't need to install `virtualenv` as a separate package. However, if you are on an older Python 3 version or prefer the `virtualenv` package for its extra features, you can install it globally:

Bash

```
pip install virtualenv
```

(If you have Python 3.3+, you can skip this step and use `python -m venv` directly)

9.3 Creating a Virtual Environment

Navigate to your project directory (or create a new one for this module).

Bash

```
# Navigate to your project folder
cd C:\Users\YourUser\PythonProjects\DatabaseApp
# Or create a new one:
mkdir DatabaseApp
cd DatabaseApp
```

Now, create the virtual environment. Let's call it `venv` (a common convention).

Bash

```
python -m venv venv
```

- This command creates a directory named `venv` inside your current directory. This directory contains a minimal Python installation and a `pip` installer just for this environment.

9.4 Activating the Virtual Environment

Before you can install packages into your virtual environment, you must *activate* it.

- **On Windows:**

  Bash

  ```
  .\venv\Scripts\activate
  ```

  or

  Bash

  ```
  venv\Scripts\activate.bat
  ```

- **On macOS / Linux:**

  Bash

  ```
  source venv/bin/activate
  ```

After activation, your terminal prompt will change to include the name of your virtual environment (e.g., `(venv) C:\...\DatabaseApp>`). This indicates that you are now operating within the isolated environment. Any `pip install` commands will now install packages into this `venv` directory.

9.5 Deactivating the Virtual Environment

When you're done working on your project, you can exit the virtual environment.

Bash

```
deactivate
```

Your terminal prompt will return to its normal state.

Chapter 10: Installing `psycopg2`

Now that your virtual environment is set up and activated, we can install the necessary Python library to interact with PostgreSQL.

10.1 What is `psycopg2`?

- `psycopg2` is the most popular PostgreSQL adapter for Python.
- It allows your Python programs to connect to a PostgreSQL database, execute SQL commands, and retrieve results.
- It is designed for high concurrency and aims to be thread-safe.

10.2 Installation

Make sure your virtual environment is activated before running the installation command.

Bash

```
(venv) pip install psycopg2-binary
```

- We use `psycopg2-binary` for simpler installation. The regular `psycopg2` might require you to have development headers for PostgreSQL and a C compiler on your system, which can be complex. `psycopg2-binary` provides pre-compiled wheels, making it easier to get started.

10.3 Verification

To confirm `psycopg2` is installed correctly within your virtual environment:

1. Keep your virtual environment activated.
2. Open a Python interpreter:

   Bash

   ```
   (venv) python
   ```

3. Try to import `psycopg2`:

Python

```
>>> import psycopg2
>>>
```

If you don't get an `ImportError`, it means `psycopg2` is successfully installed.

4. Exit the Python interpreter:

Python

```
>>> exit()
```

Chapter 11: Connecting to the Database (Python)

This is where the real integration begins! We'll write Python code to establish a connection to your PostgreSQL database.

11.1 Import `psycopg2`

The first line in your Python script will be to import the library.

Python

```
import psycopg2
```

11.2 Connection Parameters

To connect, you need specific details about your PostgreSQL server:

- `dbname`: The name of the database you want to connect to (e.g., `myapp_db`).
- `user`: The username to connect with (e.g., `postgres`).
- `password`: The password for that user.
- `host`: The database server's address (usually `localhost` if on your machine).
- `port`: The port the database server is listening on (default is `5432`).

11.3 `psycopg2.connect()`

The `connect()` function establishes a connection. It returns a `connection` object.

Python

```
conn = psycopg2.connect(
    dbname="myapp_db",
    user="postgres",
    password="your_postgres_password", # IMPORTANT: Replace with your
actual password
    host="localhost",
    port="5432"
)
```

11.4 Cursor Object

Once you have a `connection` object, you need a `cursor` object. The cursor allows you to execute SQL commands and fetch results from the database.

Python

```
cur = conn.cursor()
```

11.5 Handling Connections (`try...finally` or `with` statement)

It's crucial to always close your database connections and cursors to free up resources. The `try...finally` block or the `with` statement are excellent for this.

Using `try...finally`:

Python

```
try:
    conn = psycopg2.connect(...)
    cur = conn.cursor()
    # Your database operations go here
    print("Database connection successful!")
except psycopg2.Error as e:
    print(f"Error connecting to database: {e}")
finally:
    if 'cur' in locals() and cur: # Check if cursor was created
        cur.close()
    if 'conn' in locals() and conn: # Check if connection was created
        conn.close()
    print("Database connection closed.")
```

Using `with` statement (Recommended for cleaner code):

`psycopg2` connection and cursor objects can be used with Python's `with` statement, which automatically handles closing them even if errors occur.

Python

```
try:
    with psycopg2.connect(dbname="myapp_db", user="postgres",
password="your_postgres_password", host="localhost", port="5432") as conn:
        with conn.cursor() as cur:
            # Your database operations go here
            print("Database connection successful!")
except psycopg2.Error as e:
    print(f"Error connecting to database: {e}")
print("Database connection closed (handled by 'with' statement).")
```

11.6 Example Code: Basic Connection and Disconnection

Let's create a simple Python script to test our database connection.

Python

```
# connect_db.py
import psycopg2
```

```
def test_db_connection():
    conn = None
    cur = None
    try:
        # Replace with your actual PostgreSQL password
        conn = psycopg2.connect(
            dbname="myapp_db",
            user="postgres",
            password="your_postgres_password",
            host="localhost",
            port="5432"
        )
        cur = conn.cursor()
        print("Successfully connected to the database!")
        # You can execute a simple query to confirm
        cur.execute("SELECT version();")
        db_version = cur.fetchone()[0]
        print(f"PostgreSQL version: {db_version}")

    except psycopg2.Error as e:
        print(f"Database connection error: {e}")
    finally:
        if cur:
            cur.close()
        if conn:
            conn.close()
        print("Database connection closed.")

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

To run this:

1. Save the code as `connect_db.py`.
2. Make sure your `myapp_db` exists and PostgreSQL server is running.
3. **Activate your virtual environment.**
4. Run from your terminal: `python connect_db.py`

Expected Output:

```
Successfully connected to the database!
PostgreSQL version: PostgreSQL 16.2 on x86_64-pc-linux-gnu, compiled by...
Database connection closed.
```

(Version number and details may vary)

Chapter 12: Creating Table using Python

Now that we can connect, let's use Python to execute a `CREATE TABLE` SQL command. This is useful for programmatically setting up your database schema.

12.1 SQL Query as a String

You'll define your SQL `CREATE TABLE` statement as a multi-line Python string.

12.2 `cursor.execute()`

The `cursor.execute()` method is used to send SQL commands to the database.

12.3 `conn.commit()`

- For SQL commands that modify the database (like `CREATE TABLE`, `INSERT`, `UPDATE`, `DELETE`), these changes are initially staged in a transaction.
- To make these changes permanent in the database, you **must call `conn.commit()`**. If you don't commit, the changes will not be saved.
- For `SELECT` queries, `commit()` is not strictly necessary, but it doesn't hurt.

12.4 Example Code: Create a `Products` table

Let's create a new table called `Products` with `product_id`, `name`, `price`, and `stock_quantity`.

Python

```python
# create_table.py
import psycopg2

def create_products_table():
    conn = None
    try:
        conn = psycopg2.connect(
            dbname="myapp_db",
            user="postgres",
            password="your_postgres_password", # Replace
            host="localhost",
            port="5432"
        )
        cur = conn.cursor()

        create_table_sql = """
        CREATE TABLE IF NOT EXISTS Products (
            product_id SERIAL PRIMARY KEY,
            name VARCHAR(100) UNIQUE NOT NULL,
            price NUMERIC(10, 2) NOT NULL,
            stock_quantity INT DEFAULT 0
        );
        """
        cur.execute(create_table_sql)
        conn.commit() # Commit the transaction to make changes permanent

        print("Table 'Products' created successfully (or already exists).")

    except psycopg2.Error as e:
        print(f"Error creating table: {e}")
    finally:
        if conn:
            conn.close()

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

To run this:

1. Save the code as `create_table.py`.
2. **Activate your virtual environment.**
3. Run: `python create_table.py`
4. **Verify using `psql`:** Connect to `myapp_db` (`psql -U postgres myapp_db`) and type `\dt` then `\d products`. You should see the new table.

Chapter 13: Inserting the Data using Python

Now, let's insert some data into our `Products` table using Python.

13.1 SQL `INSERT` Query with Placeholders

When inserting data from Python, it's crucial to use **parameterized queries** to prevent SQL Injection vulnerabilities and properly handle data types. `psycopg2` uses `%s` as a placeholder for parameters.

- **Syntax:**

  SQL

  ```
  INSERT INTO table_name (column1, column2) VALUES (%s, %s);
  ```

  You then pass a tuple of values to `cursor.execute()`.

13.2 `cursor.execute(sql, (value1, value2))`

- The second argument to `execute()` is a tuple or list of values corresponding to the `%s` placeholders in your SQL string.
- `psycopg2` handles escaping and data type conversion for you.

13.3 `conn.commit()`

Remember to call `conn.commit()` after `INSERT` statements.

13.4 Inserting Multiple Records (`cursor.executemany()`)

For inserting many rows efficiently, `cursor.executemany()` is highly recommended. It takes the SQL query and a list of tuples (where each tuple represents a row of data).

13.5 Example Code: Insert Sample Product Data

Python

```
# insert_data.py
import psycopg2

def insert_single_product(name, price, quantity):
```

```
    conn = None
    try:
        conn = psycopg2.connect(
            dbname="myapp_db",
            user="postgres",
            password="your_postgres_password", # Replace
            host="localhost",
            port="5432"
        )
        cur = conn.cursor()

        sql_insert = """
        INSERT INTO Products (name, price, stock_quantity)
        VALUES (%s, %s, %s);
        """
        cur.execute(sql_insert, (name, price, quantity))
        conn.commit()
        print(f"Successfully inserted '{name}'.")

    except psycopg2.Error as e:
        print(f"Error inserting single product: {e}")
    finally:
        if conn:
            conn.close()

def insert_multiple_products(products_list):
    conn = None
    try:
        conn = psycopg2.connect(
            dbname="myapp_db",
            user="postgres",
            password="your_postgres_password", # Replace
            host="localhost",
            port="5432"
        )
        cur = conn.cursor()

        sql_insert_many = """
        INSERT INTO Products (name, price, stock_quantity)
        VALUES (%s, %s, %s);
        """
        # products_list is a list of tuples: [('name', price, quantity),
...]
        cur.executemany(sql_insert_many, products_list)
        conn.commit()
        print(f"Successfully inserted {len(products_list)} products.")

    except psycopg2.Error as e:
        print(f"Error inserting multiple products: {e}")
    finally:
        if conn:
            conn.close()

if __name__ == "__main__":
    # Insert a single product
    insert_single_product("Laptop", 1200.00, 10)

    # Insert multiple products
    new_products = [
        ("Mouse", 25.50, 50),
        ("Keyboard", 75.00, 30),
        ("Monitor", 300.00, 15)
    ]
    insert_multiple_products(new_products)

    # Note: If you run this multiple times, and 'name' is UNIQUE,
    # you will get an error for duplicate names.
    # We will learn how to extract data next to verify insertions.
```

To run this:

1. Save the code as `insert_data.py`.
2. **Activate your virtual environment.**
3. Run: `python insert_data.py`
4. You should see success messages.

Chapter 14: Extracting the Data from the Database

Reading data from the database is fundamental. This is done using `SELECT` queries with Python.

14.1 SQL `SELECT` Query

As learned in Chapter 8, you'll form your `SELECT` SQL string.

14.2 `cursor.execute(sql)`

Execute the `SELECT` query using `cursor.execute()`. No `commit()` is needed for `SELECT` operations.

14.3 Fetching Results

After executing a `SELECT` query, the results are stored in the cursor object. You use `fetch` methods to retrieve them:

- **`cursor.fetchone():`** Retrieves the next row of a query result set, returning a single tuple, or `None` when no more data is available.
- **`cursor.fetchall():`** Retrieves all (remaining) rows of a query result set, returning a list of tuples. An empty list is returned if no rows are available.
- **`cursor.fetchmany(size):`** Retrieves the next set of rows (up to `size`) of a query result, returning a list of tuples.

14.4 Iterating over Results

If using `fetchall()`, you typically loop through the list of tuples. If using `fetchone()`, you might loop until `None` is returned.

14.5 Example Code: Select All Products and Specific Products

Python

```
# extract_data.py
import psycopg2

def get_all_products():
    conn = None
    try:
        conn = psycopg2.connect(
            dbname="myapp_db",
            user="postgres",
            password="your_postgres_password", # Replace
            host="localhost",
            port="5432"
        )
        cur = conn.cursor()

        cur.execute("SELECT product_id, name, price, stock_quantity FROM
Products ORDER BY product_id;")
        products = cur.fetchall() # Fetch all results as a list of tuples

        if products:
            print("\n--- All Products ---")
            for product in products:
                print(f"ID: {product[0]}, Name: {product[1]}, Price:
{product[2]}, Stock: {product[3]}")
        else:
            print("\nNo products found.")

    except psycopg2.Error as e:
        print(f"Error retrieving all products: {e}")
    finally:
        if conn:
            conn.close()

def get_product_by_name(product_name):
    conn = None
    try:
        conn = psycopg2.connect(
            dbname="myapp_db",
            user="postgres",
            password="your_postgres_password", # Replace
            host="localhost",
            port="5432"
        )
        cur = conn.cursor()

        sql_select_by_name = "SELECT product_id, name, price,
stock_quantity FROM Products WHERE name = %s;"
        cur.execute(sql_select_by_name, (product_name,)) # Pass tuple for
single parameter

        product = cur.fetchone() # Fetch only one result

        if product:
            print(f"\n--- Product '{product_name}' Found ---")
            print(f"ID: {product[0]}, Name: {product[1]}, Price:
{product[2]}, Stock: {product[3]}")
        else:
            print(f"\nProduct '{product_name}' not found.")

    except psycopg2.Error as e:
        print(f"Error retrieving product by name: {e}")
    finally:
        if conn:
            conn.close()

if __name__ == "__main__":
    get_all_products()
    get_product_by_name("Keyboard")
    get_product_by_name("Webcam") # This product does not exist
```

To run this:

1. Save the code as `extract_data.py`.
2. **Activate your virtual environment.**
3. Run: `python extract_data.py`

Expected Output (similar to):

```
--- All Products ---
ID: 1, Name: Laptop, Price: 1200.00, Stock: 10
ID: 2, Name: Mouse, Price: 25.50, Stock: 50
ID: 3, Name: Keyboard, Price: 75.00, Stock: 30
ID: 4, Name: Monitor, Price: 300.00, Stock: 15

--- Product 'Keyboard' Found ---
ID: 3, Name: Keyboard, Price: 75.00, Stock: 30

Product 'Webcam' not found.
```

Chapter 15: Adding the Input from the User

Finally, let's combine user input with our database operations, allowing users to interact with your data.

15.1 Combining with User Input

You can use Python's built-in `input()` function for simple command-line user interaction, or integrate it with a Tkinter GUI for more complex applications (as you learned in Module 9). For simplicity, we'll use `input()` here.

15.2 Parameterized Queries (Revisit Importance)

It's paramount to always use **parameterized queries** (`%s` placeholders) when inserting or updating data based on user input. **NEVER use f-strings or direct string concatenation to build SQL queries with user input.** This prevents a major security vulnerability known as **SQL Injection**.

15.3 Example Code: Add/Search/Delete Product based on User Input

Python

```
# user_input_db_app.py
import psycopg2

def get_db_connection():
    """Helper function to create and return a database connection."""
    return psycopg2.connect(
        dbname="myapp_db",
        user="postgres",
        password="your_postgres_password", # Replace
        host="localhost",
        port="5432"
    )

def add_new_product():
    name = input("Enter product name: ")
    try:
        price = float(input("Enter price: "))
        quantity = int(input("Enter stock quantity: "))
    except ValueError:
        print("Invalid price or quantity. Please enter numbers.")
        return

    conn = None
    try:
        conn = get_db_connection()
        cur = conn.cursor()
        sql = "INSERT INTO Products (name, price, stock_quantity) VALUES
(%s, %s, %s);"
        cur.execute(sql, (name, price, quantity))
        conn.commit()
        print(f"Product '{name}' added successfully!")
    except psycopg2.Error as e:
        print(f"Error adding product: {e}")
    finally:
        if conn:
            conn.close()

def find_product_by_name():
    name = input("Enter product name to search: ")
    conn = None
    try:
        conn = get_db_connection()
        cur = conn.cursor()
        sql = "SELECT product_id, name, price, stock_quantity FROM Products
WHERE name = %s;"
        cur.execute(sql, (name,))
        product = cur.fetchone()

        if product:
            print(f"\n--- Product Details ---")
            print(f"ID: {product[0]}, Name: {product[1]}, Price:
{product[2]}, Stock: {product[3]}")
        else:
            print(f"Product '{name}' not found.")
    except psycopg2.Error as e:
        print(f"Error searching for product: {e}")
    finally:
        if conn:
            conn.close()

def delete_product_by_name():
    name = input("Enter product name to delete: ")
    conn = None
    try:
        conn = get_db_connection()
        cur = conn.cursor()
        sql = "DELETE FROM Products WHERE name = %s;"
        cur.execute(sql, (name,))
        conn.commit()
        if cur.rowcount > 0: # Check if any rows were affected
            print(f"Product '{name}' deleted successfully!")
        else:
            print(f"Product '{name}' not found for deletion.")
    except psycopg2.Error as e:
        print(f"Error deleting product: {e}")
    finally:
        if conn:
            conn.close()

def view_all_products():
    conn = None
    try:
        conn = get_db_connection()
        cur = conn.cursor()
        cur.execute("SELECT product_id, name, price, stock_quantity FROM
Products ORDER BY product_id;")
        products = cur.fetchall()

        if products:
            print("\n--- All Products in Database ---")
            for p in products:
                print(f"ID: {p[0]}, Name: {p[1]}, Price: {p[2]}, Stock:
{p[3]}")
        else:
            print("\nNo products found in the database.")
    except psycopg2.Error as e:
        print(f"Error viewing products: {e}")
    finally:
        if conn:
            conn.close()


def main_menu():
    while True:
        print("\n--- Product Management System ---")
        print("1. Add New Product")
        print("2. Search Product by Name")
        print("3. Delete Product by Name")
        print("4. View All Products")
        print("5. Exit")
        choice = input("Enter your choice: ")

        if choice == '1':
            add_new_product()
        elif choice == '2':
            find_product_by_name()
        elif choice == '3':
            delete_product_by_name()
        elif choice == '4':
            view_all_products()
        elif choice == '5':
            print("Exiting application. Goodbye!")
            break
        else:
            print("Invalid choice. Please try again.")
```

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

To run this:

1. Save the code as `user_input_db_app.py`.
2. **Activate your virtual environment.**
3. Run: `python user_input_db_app.py`

Expected Interaction: The script will present a menu. You can choose options to add products (it will prompt for name, price, quantity), search for products, delete products, or view all existing products in your `myapp_db`.