

Introduction to Python SQL Libraries

by [Usman Malik](#)  [12 Comments](#)

 [basics](#) [databases](#) [tools](#)

Mark as Completed



 [Tweet](#)

 [Share](#)

 [Email](#)

Table of Contents

- [Understanding the Database Schema](#)
- [Using Python SQL Libraries to Connect to a Database](#)
 - [SQLite](#)
 - [MySQL](#)
 - [PostgreSQL](#)
- [Creating Tables](#)
 - [SQLite](#)
 - [MySQL](#)
 - [PostgreSQL](#)
- [Inserting Records](#)
 - [SQLite](#)
 - [MySQL](#)
 - [PostgreSQL](#)
- [Selecting Records](#)
 - [SQLite](#)
 - [MySQL](#)
 - [PostgreSQL](#)
- [Updating Table Records](#)
 - [SQLite](#)
 - [MySQL](#)
 - [PostgreSQL](#)
- [Deleting Table Records](#)
 - [SQLite](#)
 - [MySQL](#)
 - [PostgreSQL](#)
- [Conclusion](#)

All software applications interact with **data**, most commonly through a [database management system \(DBMS\)](#). Some programming languages come with modules that you can use to interact with a DBMS, while others require the use of third-party packages. In this tutorial, you’ll explore the different **Python SQL libraries** that you can use. You’ll develop a straightforward application to interact with SQLite, MySQL, and PostgreSQL databases.

In this tutorial, you’ll learn how to:

- **Connect** to different database management systems with Python SQL libraries
- **Interact** with SQLite, MySQL, and PostgreSQL databases
- **Perform** common database queries using a Python application
- **Develop** applications across different databases using a Python script

To get the most out of this tutorial, you should have knowledge of basic Python, SQL, and working with database management systems. You should also be able to download and [import](#) packages in Python and know how to install and run different database servers locally or remotely.

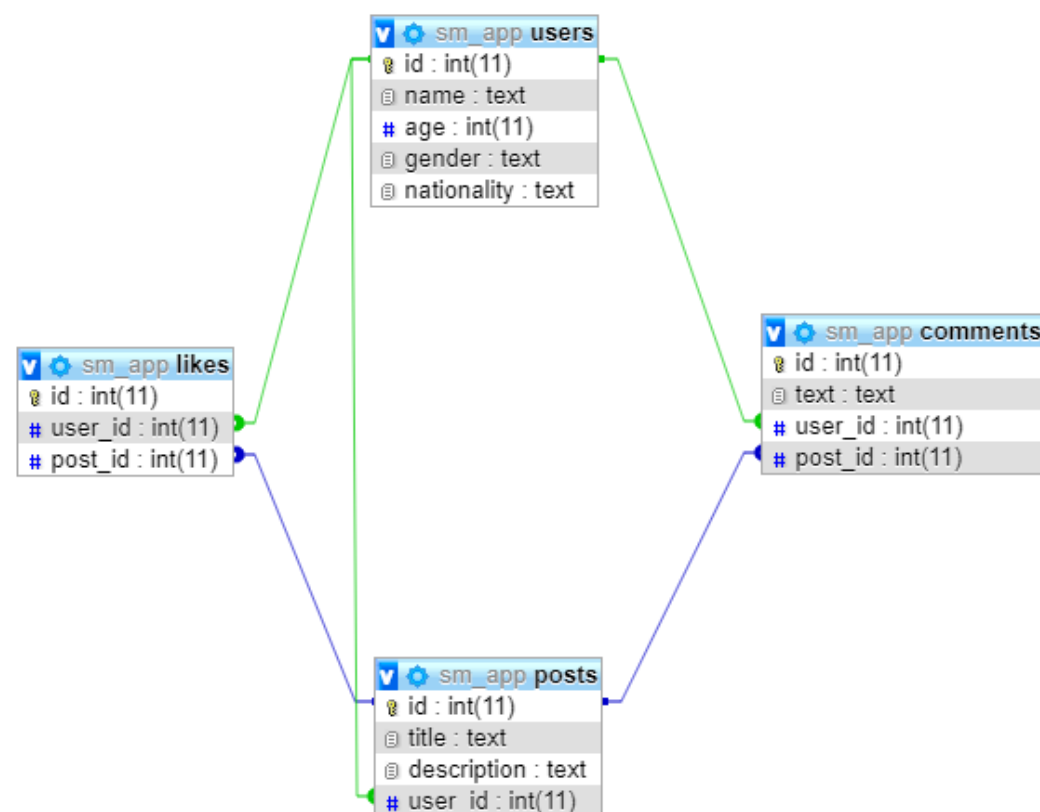
Free PDF Download: [Python 3 Cheat Sheet](#)

Understanding the Database Schema

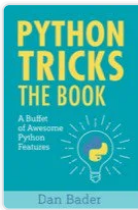
In this tutorial, you’ll develop a very small database for a social media application. The database will consist of four tables:

1. users
2. posts
3. comments
4. likes

A high-level diagram of the database schema is shown below:



Both **users** and **posts** will have a [one-to-many relationship](#) since one user can like many posts. Similarly, one user can post many comments, and one post can also have multiple comments. So, both **users** and **posts** will also have one-to-many relationships with the **comments** table. This also applies to the **likes** table, so both **users** and **posts** will have a one-to-many relationship with the **likes** table.



“I wished I had access to a book like this when I started learning Python many years ago”

— Mariatta Wijaya, CPython Core Developer

[Learn More »](#)

[Remove ads](#)

Using Python SQL Libraries to Connect to a Database

Before you interact with any database through a Python SQL Library, you have to **connect** to that database. In this section, you’ll see how to connect to [SQLite](#), [MySQL](#), and [PostgreSQL](#) databases from within a Python application.

Note: You’ll need [MySQL](#) and [PostgreSQL](#) servers up and running before you execute the scripts in the MySQL and PostgreSQL database sections. For a quick intro on how to start a MySQL server, check out the MySQL section of [Starting a Django Project](#). To learn how to create a database in PostgreSQL, check out the Setting Up a Database section of [Preventing SQL Injection Attacks With Python](#).

It’s recommended that you create three different Python files, so you have one for each of the three databases. You’ll execute the script for each database in its corresponding file.

SQLite

[SQLite](#) is probably the most straightforward database to connect to with a Python application since you don’t need to install any external Python SQL modules to do so. By default, your [Python installation](#) contains a Python SQL library named [sqlite3](#) that you can use to interact with an SQLite database.

What’s more, SQLite databases are **serverless** and **self-contained**, since they read and write data to a file. This means that, unlike with MySQL and PostgreSQL, you don’t even need to install and run an SQLite server to perform database operations!

Here’s how you use `sqlite3` to connect to an SQLite database in Python:

Python

```
1 import sqlite3
2 from sqlite3 import Error
3
4 def create_connection(path):
5     connection = None
6     try:
7         connection = sqlite3.connect(path)
8         print("Connection to SQLite DB successful")
9     except Error as e:
10        print(f"The error '{e}' occurred")
11
12    return connection
```

Here’s how this code works:

- **Lines 1 and 2** import `sqlite3` and the module’s `Error` class.
- **Line 4** defines a function `.create_connection()` that accepts the path to the SQLite database.
- **Line 7** uses `.connect()` from the `sqlite3` module and takes the SQLite database path as a parameter. If the database exists at the specified location, then a connection to the database is established. Otherwise, a new database is created at the specified location, and a connection is established.
- **Line 8** prints the status of the successful database connection.
- **Line 9** catches any [exception](#) that might be thrown if `.connect()` fails to establish a connection.
- **Line 10** displays the error message in the console.

`sqlite3.connect(path)` returns a connection object, which is in turn returned by `create_connection()`. This connection object can be used to execute queries on an SQLite database. The following script creates a connection to the SQLite database:

Python

```
connection = create_connection("E:\\sm_app.sqlite")
```

Once you execute the above script, you'll see that a database file `sm_app.sqlite` is created in the root directory. Note that you can change the location to match your setup.

MySQL

Unlike SQLite, there's no default Python SQL module that you can use to connect to a MySQL database. Instead, you'll need to install a **Python SQL driver** for MySQL in order to interact with a MySQL database from within a Python application. One such driver is `mysql-connector-python`. You can download this Python SQL module with [pip](#):

Shell

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

Note that MySQL is a **server-based** database management system. One MySQL server can have multiple databases. Unlike SQLite, where creating a connection is tantamount to creating a database, a MySQL database has a two-step process for database creation:

1. **Make a connection** to a MySQL server.
2. **Execute a separate query** to create the database.

Define a function that connects to the MySQL database server and returns the connection object:

Python

```
1 import mysql.connector
2 from mysql.connector import Error
3
4 def create_connection(host_name, user_name, user_password):
5     connection = None
6     try:
7         connection = mysql.connector.connect(
8             host=host_name,
9             user=user_name,
10            passwd=user_password
11        )
12        print("Connection to MySQL DB successful")
13    except Error as e:
14        print(f"The error '{e}' occurred")
15
16    return connection
17
18 connection = create_connection("localhost", "root", "")
```

In the above script, you define a function `create_connection()` that accepts three parameters:

1. `host_name`
2. `user_name`
3. `user_password`

The `mysql.connector` Python SQL module contains a method `.connect()` that you use in line 7 to connect to a MySQL database server. Once the connection is established, the `connection` object is returned to the calling function. Finally, in line 18 you call `create_connection()` with the host name, username, and password.

So far, you've only established the connection. The database is not yet created. To do this, you'll define another function `create_database()` that accepts two parameters:

1. **connection** is the connection object to the database server that you want to interact with.
2. **query** is the query that creates the database.

Here's what this function looks like:

Python

```
def create_database(connection, query):
    cursor = connection.cursor()
    try:
        cursor.execute(query)
        print("Database created successfully")
    except Error as e:
        print(f"The error '{e}' occurred")
```

To execute queries, you use the cursor object. The query to be executed is passed to `cursor.execute()` in [string](#) format.

Create a database named `sm_app` for your social media app in the MySQL database server:

Python

```
create_database_query = "CREATE DATABASE sm_app"
create_database(connection, create_database_query)
```

Now you've created a database `sm_app` on the database server. However, the `connection` object returned by the `create_connection()` is connected to the MySQL database server. You need to connect to the `sm_app` database. To do so, you can modify `create_connection()` as follows:

Python

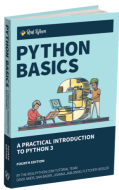
```
1 def create_connection(host_name, user_name, user_password, db_name):
2     connection = None
3     try:
4         connection = mysql.connector.connect(
5             host=host_name,
6             user=user_name,
7             passwd=user_password,
8             database=db_name
9         )
10        print("Connection to MySQL DB successful")
11    except Error as e:
12        print(f"The error '{e}' occurred")
13
14    return connection
```

You can see in line 8 that `create_connection()` now accepts an additional parameter called `db_name`. This parameter specifies the name of the database that you want to connect to. You can pass in the name of the database you want to connect to when you call this function:

Python

```
connection = create_connection("localhost", "root", "", "sm_app")
```

The above script successfully calls `create_connection()` and connects to the `sm_app` database.



[Your Practical Introduction to Python 3 »](#)

 [Remove ads](#)

PostgreSQL

Like MySQL, there's no default Python SQL library that you can use to interact with a PostgreSQL database. Instead, you need to install a **third-party Python SQL driver** to interact with PostgreSQL. One such Python SQL driver for PostgreSQL is `psycopg2`. Execute the following command on your [terminal](#) to install the `psycopg2` Python SQL module:

Shell

```
$ pip install psycopg2
```

Like with the SQLite and MySQL databases, you'll define `create_connection()` to make a connection with your PostgreSQL database:

Python

```
import psycopg2
from psycopg2 import OperationalError

def create_connection(db_name, db_user, db_password, db_host, db_port):
    connection = None
    try:
        connection = psycopg2.connect(
            database=db_name,
            user=db_user,
            password=db_password,
            host=db_host,
            port=db_port,
        )
        print("Connection to PostgreSQL DB successful")
    except OperationalError as e:
        print(f"The error '{e}' occurred")
    return connection
```

You use `psycopg2.connect()` to connect to a PostgreSQL server from within your Python application.

You can then use `create_connection()` to create a connection to a PostgreSQL database. First, you'll make a connection with the default database `postgres` by using the following string:

Python

```
connection = create_connection(
    "postgres", "postgres", "abc123", "127.0.0.1", "5432"
)
```

Next, you have to create the database `sm_app` inside the default `postgres` database. You can define a function to execute any SQL query in PostgreSQL. Below, you define `create_database()` to create a new database in the PostgreSQL database server:

Python

```
def create_database(connection, query):
    connection.autocommit = True
    cursor = connection.cursor()
    try:
        cursor.execute(query)
        print("Query executed successfully")
    except OperationalError as e:
        print(f"The error '{e}' occurred")

create_database_query = "CREATE DATABASE sm_app"
create_database(connection, create_database_query)
```

Once you run the script above, you'll see the `sm_app` database in your PostgreSQL database server.

Before you execute queries on the `sm_app` database, you need to connect to it:

Python

```
connection = create_connection(
    "sm_app", "postgres", "abc123", "127.0.0.1", "5432"
)
```

Once you execute the above script, a connection will be established with the `sm_app` database located in the `postgres` database server. Here, `127.0.0.1` refers to the database server host IP address, and `5432` refers to the port number of the database server.

Creating Tables

In the previous section, you saw how to connect to SQLite, MySQL, and PostgreSQL database servers using different Python SQL libraries. You created the `sm_app` database on all three database servers. In this section, you'll see how to **create tables** inside these three databases.

As discussed earlier, you'll create four tables:

1. `users`
2. `posts`
3. `comments`
4. `likes`

You'll start with SQLite.

SQLite

To execute queries in SQLite, use `cursor.execute()`. In this section, you'll define a function `execute_query()` that uses this method. Your function will accept the `connection` object and a query string, which you'll pass to `cursor.execute()`.

`.execute()` can execute any query passed to it in the form of string. You'll use this method to create tables in this section. In the upcoming sections, you'll use this same method to execute update and delete queries as well.

Note: This script should be executed in the same file where you created the connection for your SQLite database.

Here's your function definition:

Python

```
def execute_query(connection, query):
    cursor = connection.cursor()
    try:
        cursor.execute(query)
        connection.commit()
        print("Query executed successfully")
    except Error as e:
        print(f"The error '{e}' occurred")
```

This code tries to execute the given query and prints an error message if necessary.

Next, write your **query**:

Python

```
create_users_table = """
CREATE TABLE IF NOT EXISTS users (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
    age INTEGER,
    gender TEXT,
    nationality TEXT
);
"""
```

This says to create a table `users` with the following five columns:

1. `id`
2. `name`
3. `age`
4. `gender`
5. `nationality`

Finally, you'll call `execute_query()` to create the table. You'll pass in the `connection` object that you created in the previous section, along with the `create_users_table` string that contains the create table query:

Python

```
execute_query(connection, create_users_table)
```

The following query is used to create the `posts` table:

Python

```
create_posts_table = """
CREATE TABLE IF NOT EXISTS posts(
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    title TEXT NOT NULL,
    description TEXT NOT NULL,
    user_id INTEGER NOT NULL,
    FOREIGN KEY (user_id) REFERENCES users (id)
);
"""
```

Since there's a one-to-many relationship between `users` and `posts`, you can see a foreign key `user_id` in the `posts` table that references the `id` column in the `users` table. Execute the following script to create the `posts` table:

Python

```
execute_query(connection, create_posts_table)
```

Finally, you can create the `comments` and `likes` tables with the following script:

Python

```
create_comments_table = """
CREATE TABLE IF NOT EXISTS comments (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    text TEXT NOT NULL,
    user_id INTEGER NOT NULL,
    post_id INTEGER NOT NULL,
    FOREIGN KEY (user_id) REFERENCES users (id) FOREIGN KEY (post_id) REFERENCES posts (id)
);
"""

create_likes_table = """
CREATE TABLE IF NOT EXISTS likes (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    user_id INTEGER NOT NULL,
    post_id integer NOT NULL,
    FOREIGN KEY (user_id) REFERENCES users (id) FOREIGN KEY (post_id) REFERENCES posts (id)
);
"""

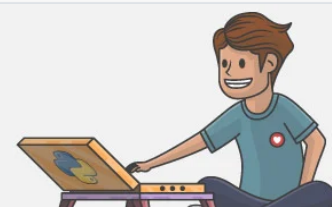
execute_query(connection, create_comments_table)
execute_query(connection, create_likes_table)
```


You can see that **creating tables** in SQLite is very similar to using raw SQL. All you have to do is store the query in a string variable and then pass that variable to `cursor.execute()`.

5 Thoughts on Mastering Python

A free email class for Python developers

realpython.com



 [Remove ads](#)

MySQL

You'll use the `mysql-connector-python` Python SQL module to create tables in MySQL. Just like with SQLite, you need to pass your query to `cursor.execute()`, which is returned by calling `.cursor()` on the `connection` object. You can create another function `execute_query()` that accepts the `connection` and `query` string:

Python

```
1 def execute_query(connection, query):
2     cursor = connection.cursor()
3     try:
4         cursor.execute(query)
5         connection.commit()
6         print("Query executed successfully")
7     except Error as e:
8         print(f"The error '{e}' occurred")
```

In line 4, you pass the query to `cursor.execute()`.

Now you can create your users table using this function:

Python

```
create_users_table = """
CREATE TABLE IF NOT EXISTS users (
    id INT AUTO_INCREMENT,
    name TEXT NOT NULL,
    age INT,
    gender TEXT,
    nationality TEXT,
    PRIMARY KEY (id)
) ENGINE = InnoDB
"""

execute_query(connection, create_users_table)
```

The query for implementing the foreign key relation is slightly different in MySQL as compared to SQLite. What's more, MySQL uses the `AUTO_INCREMENT` keyword (compared to the SQLite `AUTOINCREMENT` keyword) to create columns where the values are **automatically incremented** when new records are inserted.

The following script creates the `posts` table, which contains a foreign key `user_id` that references the `id` column of the `users` table:

Python

```
create_posts_table = """
CREATE TABLE IF NOT EXISTS posts (
    id INT AUTO_INCREMENT,
    title TEXT NOT NULL,
    description TEXT NOT NULL,
    user_id INTEGER NOT NULL,
    FOREIGN KEY fk_user_id (user_id) REFERENCES users(id),
    PRIMARY KEY (id)
) ENGINE = InnoDB
"""

execute_query(connection, create_posts_table)
```

Similarly, to create the `comments` and `likes` tables, you can pass the corresponding `CREATE` queries to `execute_query()`.

PostgreSQL

Like with SQLite and MySQL databases, the `connection` object that's returned by `psycopg2.connect()` contains a `cursor` object. You can use `cursor.execute()` to execute Python SQL queries on your PostgreSQL database.

Define a function `execute_query()`:

Python

```
def execute_query(connection, query):
    connection.autocommit = True
    cursor = connection.cursor()
    try:
        cursor.execute(query)
        print("Query executed successfully")
    except OperationalError as e:
        print(f"The error '{e}' occurred")
```

You can use this function to create tables, insert records, modify records, and delete records in your PostgreSQL database.

Now create the users table inside the sm_app database:

Python

```
create_users_table = """
CREATE TABLE IF NOT EXISTS users (
    id SERIAL PRIMARY KEY,
    name TEXT NOT NULL,
    age INTEGER,
    gender TEXT,
    nationality TEXT
)
"""

execute_query(connection, create_users_table)
```

You can see that the query to create the users table in PostgreSQL is slightly different than SQLite and MySQL. Here, the keyword SERIAL is used to create columns that increment automatically. Recall that MySQL uses the keyword AUTO_INCREMENT.

In addition, foreign key referencing is also specified differently, as shown in the following script that creates the posts table:

Python

```
create_posts_table = """
CREATE TABLE IF NOT EXISTS posts (
    id SERIAL PRIMARY KEY,
    title TEXT NOT NULL,
    description TEXT NOT NULL,
    user_id INTEGER REFERENCES users(id)
)
"""

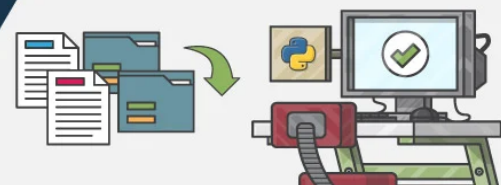
execute_query(connection, create_posts_table)
```

To create the comments table, you'll have to write a CREATE query for the comments table and pass it to execute_query(). The process for creating the likes table is the same. You only have to modify the CREATE query to create the likes table instead of the comments table.

Free PDF Download: Python 3 Cheat Sheet

[Download Now](#)

realpython.com



[Remove ads](#)

Inserting Records

In the previous section, you saw how to create tables in your SQLite, MySQL, and PostgreSQL databases by using different Python SQL modules. In this section, you'll see how to **insert records** into your tables.

SQLite

To insert records into your SQLite database, you can use the same `execute_query()` function that you used to create tables. First, you have to store your `INSERT INTO` query in a string. Then, you can pass the `connection` object and query string to `execute_query()`. Let's insert five records into the `users` table:

Python

```
create_users = """
INSERT INTO
  users (name, age, gender, nationality)
VALUES
  ('James', 25, 'male', 'USA'),
  ('Leila', 32, 'female', 'France'),
  ('Brigitte', 35, 'female', 'England'),
  ('Mike', 40, 'male', 'Denmark'),
  ('Elizabeth', 21, 'female', 'Canada');
"""

execute_query(connection, create_users)
```

Since you set the `id` column to auto-increment, you don't need to specify the value of the `id` column for these users. The `users` table will auto-populate these five records with `id` values from 1 to 5.

Now insert six records into the `posts` table:

Python

```
create_posts = """
INSERT INTO
  posts (title, description, user_id)
VALUES
  ("Happy", "I am feeling very happy today", 1),
  ("Hot Weather", "The weather is very hot today", 2),
  ("Help", "I need some help with my work", 2),
  ("Great News", "I am getting married", 1),
  ("Interesting Game", "It was a fantastic game of tennis", 5),
  ("Party", "Anyone up for a late-night party today?", 3);
"""

execute_query(connection, create_posts)
```

It's important to mention that the `user_id` column of the `posts` table is a **foreign key** that references the `id` column of the `users` table. This means that the `user_id` column must contain a value that **already exists** in the `id` column of the `users` table. If it doesn't exist, then you'll see an error.

Similarly, the following script inserts records into the `comments` and `likes` tables:

Python

```
create_comments = """
INSERT INTO
    comments (text, user_id, post_id)
VALUES
    ('Count me in', 1, 6),
    ('What sort of help?', 5, 3),
    ('Congrats buddy', 2, 4),
    ('I was rooting for Nadal though', 4, 5),
    ('Help with your thesis?', 2, 3),
    ('Many congratulations', 5, 4);
"""

create_likes = """
INSERT INTO
    likes (user_id, post_id)
VALUES
    (1, 6),
    (2, 3),
    (1, 5),
    (5, 4),
    (2, 4),
    (4, 2),
    (3, 6);
"""

execute_query(connection, create_comments)
execute_query(connection, create_likes)
```

In both cases, you store your `INSERT INTO` query as a string and execute it with `execute_query()`.

MySQL

There are two ways to insert records into MySQL databases from a Python application. The first approach is similar to SQLite. You can store the `INSERT INTO` query in a string and then use `cursor.execute()` to insert records.

Earlier, you defined a wrapper function `execute_query()` that you used to insert records. You can use this same function now to insert records into your MySQL table. The following script inserts records into the `users` table using `execute_query()`:

Python

```
create_users = """
INSERT INTO
    `users` (`name`, `age`, `gender`, `nationality`)
VALUES
    ('James', 25, 'male', 'USA'),
    ('Leila', 32, 'female', 'France'),
    ('Brigitte', 35, 'female', 'England'),
    ('Mike', 40, 'male', 'Denmark'),
    ('Elizabeth', 21, 'female', 'Canada');
"""

execute_query(connection, create_users)
```

The second approach uses `cursor.executemany()`, which accepts two parameters:

1. **The query** string containing placeholders for the records to be inserted
2. **The [list](#)** of records that you want to insert

Look at the following example, which inserts two records into the `likes` table:

Python

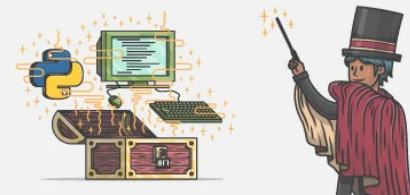
```
sql = "INSERT INTO likes ( user_id, post_id ) VALUES ( %s, %s )"
val = [(4, 5), (3, 4)]

cursor = connection.cursor()
cursor.executemany(sql, val)
connection.commit()
```

It's up to you which approach you choose to insert records into your MySQL table. If you're an expert in SQL, then you can use `.execute()`. If you're not much familiar with SQL, then it may be more straightforward for you to use `.executemany()`. With either of the two approaches, you can successfully insert records into the `posts`, `comments`, and `likes` tables.

Improve Your Python with Python Tricks

realpython.com



 [Remove ads](#)

PostgreSQL

In the previous section, you saw two approaches for inserting records into SQLite database tables. The first uses an SQL string query, and the second uses `.executemany()`. `psycopg2` follows this second approach, though `.execute()` is used to execute a placeholder-based query.

You pass the SQL query with the placeholders and the list of records to `.execute()`. Each record in the list will be a [tuple](#), where tuple values correspond to the column values in the database table. Here's how you can insert user records into the `users` table in a PostgreSQL database:

Python

```
users = [
    ("James", 25, "male", "USA"),
    ("Leila", 32, "female", "France"),
    ("Brigitte", 35, "female", "England"),
    ("Mike", 40, "male", "Denmark"),
    ("Elizabeth", 21, "female", "Canada"),
]

user_records = ", ".join(["%s"] * len(users))

insert_query = (
    f"INSERT INTO users (name, age, gender, nationality) VALUES {user_records}"
)

connection.autocommit = True
cursor = connection.cursor()
cursor.execute(insert_query, users)
```

The script above creates a list `users` that contains five user records in the form of tuples. Next, you create a placeholder string with five placeholder elements (`%s`) that correspond to the five user records. The placeholder string is [concatenated](#) with the query that inserts records into the `users` table. Finally, the query string and the user records are passed to `.execute()`. The above script successfully inserts five records into the `users` table.

Take a look at another example of inserting records into a PostgreSQL table. The following script inserts records into the `posts` table:

Python

```
posts = [
    ("Happy", "I am feeling very happy today", 1),
    ("Hot Weather", "The weather is very hot today", 2),
    ("Help", "I need some help with my work", 2),
    ("Great News", "I am getting married", 1),
    ("Interesting Game", "It was a fantastic game of tennis", 5),
    ("Party", "Anyone up for a late-night party today?", 3),
]

post_records = ", ".join(["%s" % post for post in posts])

insert_query = (
    f"INSERT INTO posts (title, description, user_id) VALUES {post_records}"
)

connection.autocommit = True
cursor = connection.cursor()
cursor.execute(insert_query, posts)
```

You can insert records into the `comments` and `likes` tables with the same approach.

Selecting Records

In this section, you'll see how to select records from database tables using the different Python SQL modules. In particular, you'll see how to perform `SELECT` queries on your SQLite, MySQL, and PostgreSQL databases.

SQLite

To select records using SQLite, you can again use `cursor.execute()`. However, after you've done this, you'll need to call `.fetchall()`. This method returns a list of tuples where each tuple is mapped to the corresponding row in the retrieved records.

To simplify the process, you can create a function `execute_read_query()`:

Python

```
def execute_read_query(connection, query):
    cursor = connection.cursor()
    result = None
    try:
        cursor.execute(query)
        result = cursor.fetchall()
        return result
    except Error as e:
        print(f"The error '{e}' occurred")
```

This function accepts the `connection` object and the `SELECT` query and returns the selected record.

SELECT

Let's now select all the records from the `users` table:

Python

```
select_users = "SELECT * from users"
users = execute_read_query(connection, select_users)

for user in users:
    print(user)
```

In the above script, the `SELECT` query selects all the users from the `users` table. This is passed to the `execute_read_query()`, which returns all the records from the `users` table. The records are then traversed and printed to the console.

Note: It's not recommended to use `SELECT *` on large tables since it can result in a large number of I/O operations that increase the network traffic.

The output of the above query looks like this:

Shell

```
(1, 'James', 25, 'male', 'USA')
(2, 'Leila', 32, 'female', 'France')
(3, 'Brigitte', 35, 'female', 'England')
(4, 'Mike', 40, 'male', 'Denmark')
(5, 'Elizabeth', 21, 'female', 'Canada')
```

In the same way, you can retrieve all the records from the `posts` table with the below script:

Python

```
select_posts = "SELECT * FROM posts"
posts = execute_read_query(connection, select_posts)

for post in posts:
    print(post)
```

The output looks like this:

Shell

```
(1, 'Happy', 'I am feeling very happy today', 1)
(2, 'Hot Weather', 'The weather is very hot today', 2)
(3, 'Help', 'I need some help with my work', 2)
(4, 'Great News', 'I am getting married', 1)
(5, 'Interesting Game', 'It was a fantastic game of tennis', 5)
(6, 'Party', 'Anyone up for a late-night party today?', 3)
```

The result shows all the records in the `posts` table.

JOIN

You can also execute complex queries involving **JOIN operations** to retrieve data from two related tables. For instance, the following script returns the user ids and names, along with the description of the posts that these users posted:

Python

```
select_users_posts = """
SELECT
    users.id,
    users.name,
    posts.description
FROM
    posts
    INNER JOIN users ON users.id = posts.user_id
"""

users_posts = execute_read_query(connection, select_users_posts)

for users_post in users_posts:
    print(users_post)
```

Here's the output:

Shell

```
(1, 'James', 'I am feeling very happy today')
(2, 'Leila', 'The weather is very hot today')
(2, 'Leila', 'I need some help with my work')
(1, 'James', 'I am getting married')
(5, 'Elizabeth', 'It was a fantastic game of tennis')
(3, 'Brigitte', 'Anyone up for a late night party today?')
```

You can also select data from three related tables by implementing **multiple JOIN operators**. The following script returns all posts, along with the comments on the posts and the names of the users who posted the comments:

Python

```
select_posts_comments_users = """
SELECT
    posts.description as post,
    text as comment,
    name
FROM
    posts
    INNER JOIN comments ON posts.id = comments.post_id
    INNER JOIN users ON users.id = comments.user_id
"""

posts_comments_users = execute_read_query(
    connection, select_posts_comments_users
)

for posts_comments_user in posts_comments_users:
    print(posts_comments_user)
```

The output looks like this:

Shell

```
('Anyone up for a late night party today?', 'Count me in', 'James')
('I need some help with my work', 'What sort of help?', 'Elizabeth')
('I am getting married', 'Congrats buddy', 'Leila')
('It was a fantastic game of tennis', 'I was rooting for Nadal though', 'Mike')
('I need some help with my work', 'Help with your thesis?', 'Leila')
('I am getting married', 'Many congratulations', 'Elizabeth')
```

You can see from the output that the column names are not being returned by `.fetchall()`. To return column names, you can use the `.description` attribute of the cursor object. For instance, the following list returns all the column names for the above query:

Python

```
cursor = connection.cursor()
cursor.execute(select_posts_comments_users)
cursor.fetchall()

column_names = [description[0] for description in cursor.description]
print(column_names)
```

The output looks like this:

Shell

```
['post', 'comment', 'name']
```

You can see the names of the columns for the given query.

WHERE

Now you'll execute a `SELECT` query that returns the post, along with the total number of likes that the post received:

Python

```
select_post_likes = """
SELECT
    description as Post,
    COUNT(likes.id) as Likes
FROM
    likes,
    posts
WHERE
    posts.id = likes.post_id
GROUP BY
    likes.post_id
"""

post_likes = execute_read_query(connection, select_post_likes)

for post_like in post_likes:
    print(post_like)
```

The output is as follows:

Shell

```
('The weather is very hot today', 1)
('I need some help with my work', 1)
('I am getting married', 2)
('It was a fantastic game of tennis', 1)
('Anyone up for a late night party today?', 2)
```

By using a `WHERE` clause, you're able to return more specific results.

Python Dependency Management Pitfalls

A free email class

realpython.com



 [Remove ads](#)

MySQL

The process of selecting records in MySQL is absolutely identical to selecting records in SQLite. You can use `cursor.execute()` followed by `.fetchall()`. The following script creates a wrapper function `execute_read_query()` that you can use to select records:

Python

```
def execute_read_query(connection, query):
    cursor = connection.cursor()
    result = None
    try:
        cursor.execute(query)
        result = cursor.fetchall()
        return result
    except Error as e:
        print(f"The error '{e}' occurred")
```

Now select all the records from the `users` table:

Python

```
select_users = "SELECT * FROM users"
users = execute_read_query(connection, select_users)

for user in users:
    print(user)
```

The output will be similar to what you saw with SQLite.

PostgreSQL

The process of selecting records from a PostgreSQL table with the psycopg2 Python SQL module is similar to what you did with SQLite and MySQL. Again, you'll use `cursor.execute()` followed by `.fetchall()` to select records from your PostgreSQL table. The following script selects all the records from the `users` table and prints them to the console:

Python

```
def execute_read_query(connection, query):
    cursor = connection.cursor()
    result = None
    try:
        cursor.execute(query)
        result = cursor.fetchall()
        return result
    except OperationalError as e:
        print(f"The error '{e}' occurred")

select_users = "SELECT * FROM users"
users = execute_read_query(connection, select_users)

for user in users:
    print(user)
```

Again, the output will be similar to what you've seen before.

Updating Table Records

In the last section, you saw how to select records from SQLite, MySQL, and PostgreSQL databases. In this section, you'll cover the process for **updating records** using the Python SQL libraries for SQLite, PostgreSQL, and MySQL.

SQLite

Updating records in SQLite is pretty straightforward. You can again make use of `execute_query()`. As an example, you can update the description of the post with an `id` of 2. First, `SELECT` the description of this post:

Python

```
select_post_description = "SELECT description FROM posts WHERE id = 2"

post_description = execute_read_query(connection, select_post_description)

for description in post_description:
    print(description)
```

You should see the following output:

Shell

```
('The weather is very hot today',)
```

The following script updates the description:

Python

```
update_post_description = """
UPDATE
  posts
SET
  description = "The weather has become pleasant now"
WHERE
  id = 2
"""

execute_query(connection, update_post_description)
```

Now, if you execute the `SELECT` query again, you should see the following result:

Shell

```
('The weather has become pleasant now',)
```

The output has been updated.

Write Cleaner & More Pythonic Code

realpython.com



 [Remove ads](#)

MySQL

The process of updating records in MySQL with `mysql-connector-python` is also a carbon copy of the `sqlite3` Python SQL module. You need to pass the string query to `cursor.execute()`. For example, the following script updates the description of the post with an `id` of 2:

Python

```
update_post_description = """
UPDATE
  posts
SET
  description = "The weather has become pleasant now"
WHERE
  id = 2
"""

execute_query(connection, update_post_description)
```

Again, you've used your wrapper function `execute_query()` to update the post description.

PostgreSQL

The update query for PostgreSQL is similar to what you've seen with SQLite and MySQL. You can use the above scripts to update records in your PostgreSQL table.

Deleting Table Records

In this section, you'll see how to delete table records using the Python SQL modules for SQLite, MySQL, and PostgreSQL databases. The process of deleting records is uniform for all three databases since the **DELETE** query for the three databases is the same.

SQLite

You can again use `execute_query()` to delete records from YOUR SQLite database. All you have to do is pass the `connection` object and the string query for the record you want to delete to `execute_query()`. Then, `execute_query()` will create a `cursor` object using the `connection` and pass the string query to `cursor.execute()`, which will delete the records.

As an example, try to delete the comment with an `id` of 5:

Python

```
delete_comment = "DELETE FROM comments WHERE id = 5"
execute_query(connection, delete_comment)
```

Now, if you select all the records from the `comments` table, you'll see that the fifth comment has been deleted.

MySQL

The process for deletion in MySQL is also similar to SQLite, as shown in the following example:

Python

```
delete_comment = "DELETE FROM comments WHERE id = 2"
execute_query(connection, delete_comment)
```

Here, you delete the second comment from the `sm_app` database’s `comments` table in your MySQL database server.

PostgreSQL

The delete query for PostgreSQL is also similar to SQLite and MySQL. You can write a delete query string by using the `DELETE` keyword and then passing the query and the connection object to `execute_query()`. This will delete the specified records from your PostgreSQL database.

Conclusion

In this tutorial, you’ve learned how to use three common Python SQL libraries. `sqlite3`, `mysql-connector-python`, and `psycopg2` allow you to connect a Python application to SQLite, MySQL, and PostgreSQL databases, respectively.

Now you can:

- **Interact** with SQLite, MySQL, or PostgreSQL databases
- **Use** three different Python SQL modules
- **Execute** SQL queries on various databases from within a Python application

However, this is just the tip of the iceberg! There are also Python SQL libraries for **object-relational mapping**, such as `SQLAlchemy` and `Django ORM`, that automate the task of database interaction in Python. You’ll learn more about these libraries in other tutorials in our [Python databases](#) section.

Mark as Completed

Python Tricks

Get a short & sweet **Python Trick** delivered to your inbox every couple of days. No spam ever. Unsubscribe any time. Curated by the Real Python team.

Email Address

Send Me Python Tricks »

```
1 # How to merge two dicts
2 # in Python 3.5+
3
4 >>> x = {'a': 1, 'b': 2}
5 >>> y = {'b': 3, 'c': 4}
6
7 >>> z = {**x, **y}
8
9 >>> z
10 {'c': 4, 'a': 1, 'b': 3}
```

About **Usman Malik**



Usman is an avid Pythonista and writes for Real Python.

» [More about Usman](#)

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:



[Aldren](#)



[Geir Arne](#)



[Jaya](#)

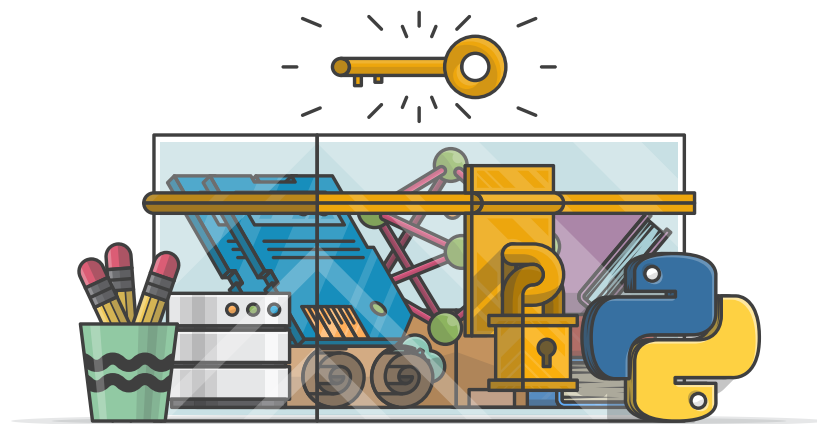


[Joanna](#)



[Mike](#)

Master Real-World Python Skills With Unlimited Access to Real Python



**Join us and get access to thousands of tutorials,
hands-on video courses, and a community of expert
Pythonistas:**

[Level Up Your Python Skills »](#)

What Do You Think?

Rate this article:



[Tweet](#)

[Share](#)

[Share](#)

[Email](#)

What's your #1 takeaway or favorite thing you learned? How are you going to put your newfound skills to use? Leave a comment below and let us know.

Commenting Tips: The most useful comments are those written with the goal of learning from or helping out other students. [Get tips for asking good questions](#) and [get answers to common questions in our support portal](#).

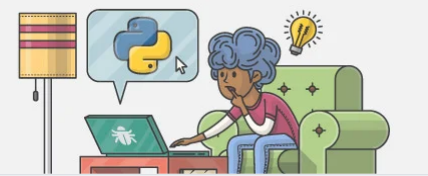
Looking for a real-time conversation? Visit the [Real Python Community Chat](#) or join the next [“Office Hours” Live Q&A Session](#). Happy Pythoning!

Keep Learning

Related Tutorial Categories: [basics](#) [databases](#) [tools](#)

Learn Python Programming, By Example

[realpython.com](#)



 [Remove ads](#)

© 2012–2023 Real Python · [Newsletter](#) · [Podcast](#) · [YouTube](#) · [Twitter](#) · [Facebook](#) · [Instagram](#) ·
[Python Tutorials](#) · [Search](#) · [Privacy Policy](#) · [Energy Policy](#) · [Advertise](#) · [Contact](#)

❤ Happy Pythoning!