Week 6 - Databases in python

Content of practical work:

1. Database Schema
2. Connecting to databases
3. Creating tables
4. Inserting Records
5. Retrieving Records
6. Content update
7. Deleting table entries

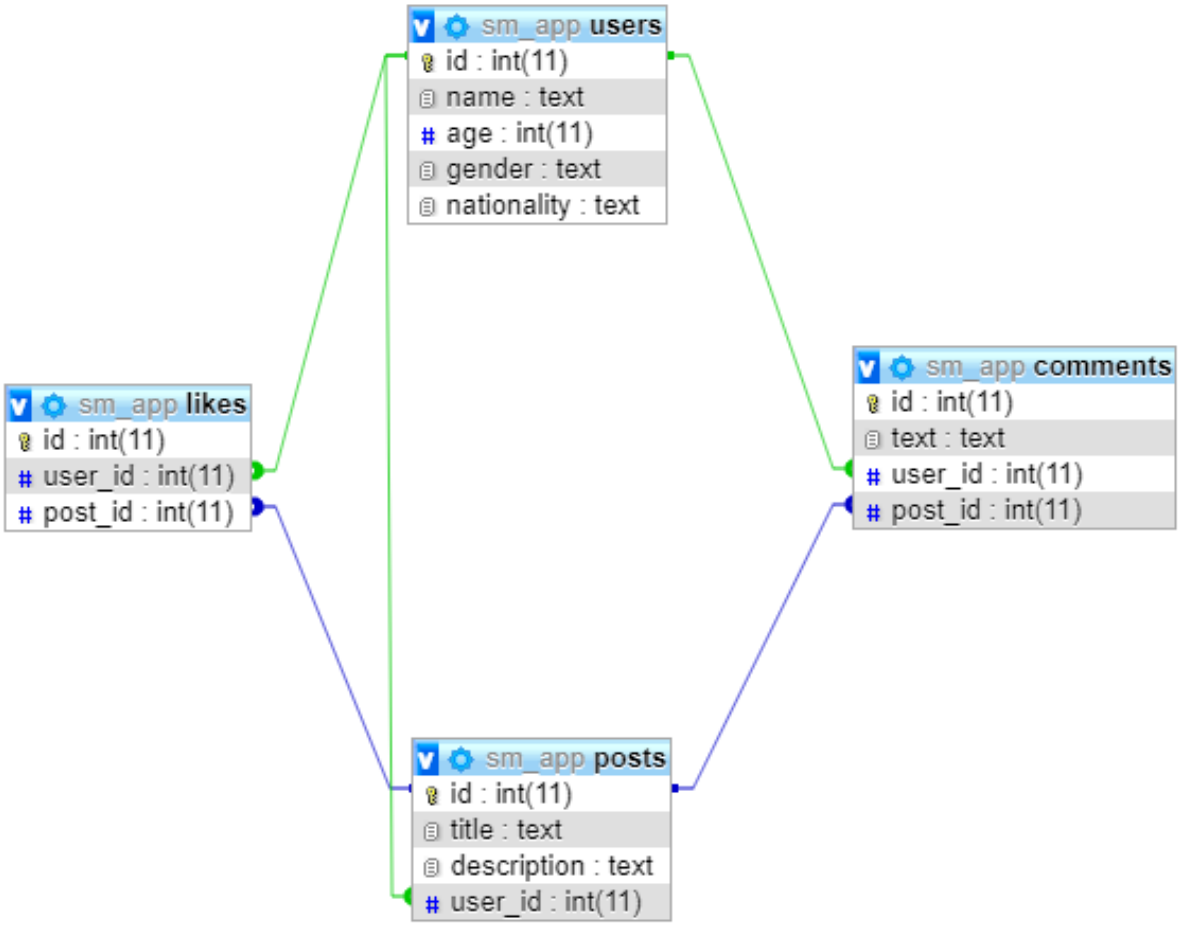
Each section has three subsections: SQLite, MySQL, and PostgreSQL.

1. Database schema for training

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

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

The database schema is shown in the figure below.



Users (users) and publications (posts) will be of the type of relationship one-to-many: one reader can like several posts. Similarly, the same user can leave many comments (comments), and one

post can have several comments. Thus both users and posts have the same relationship type with respect to comments. And likes are identical to comments in this regard.

2. Connecting to databases

Before interacting with any database through the SQL library, it must be contacted. In this section, we'll look at how to connect from a Python application to databases.SQLite,MySQLandPostgreSQL. We recommend making your own .py file for each of the three databases.
Note. To complete the sections onMySQLandPostgreSQLyou need to start the corresponding servers yourself. For a quick guide on how to start a MySQL server, check out the MySQL section in the post.Starting a Django Project(English). To learn how to create a database in PostgreSQL, go to Setting Up a Database in the postPreventing SQL injection attacks with Python(English).

SQLite

SQLiteis probably the easiest database to connect to with Python as it doesn't require you to install any external modules. By default, the Python standard library already contains the modulesqlite3.
Moreover, the SQLite database does not require a server and is self-contained, that is, it simply reads and writes data to a file. Connect using sqlite3 to the database:

```
importsqlite3
fromsqlite3importerror

def create_connection(path):
connection=None
 try:
connection = sqlite3.connect(path)
print("Connection to SQLite DB successful")
 excepterrorase:
print(f"The error '{e}' occurred")

 returnconnections
```

Here is how this code works:
Lines 1 and 2 import sqlite3 and the Error class.
Line 4 defines the create_connection() function, which accepts a path to a SQLite database.
Line 7 uses the connect() method and takes the path to the SQLite database as a parameter. If a database exists at the specified location, the connection will be established. Otherwise, a new database will be created at the specified path and a connection will be established in the same way.
Line 8 displays the status of a successful connection to the database.
Line 9 catches any exception that might be thrown if the .connect() method fails to establish a connection.
Line 10 displays an error message in the console.
sqlite3.connect(path) returns a connection object. This object can be used to query a SQLite database. The following script forms a connection to a SQLite database:

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

After executing the above script, you will see the sm_app.sqlite database file appear in the root directory of drive E. Of course, you can change the location according to your interests.

MySQL

Unlike SQLite, there is no default module in Python that can be used to connect to a MySQL database. To do this, you need to install the Python driver for MySQL. One such driver is mysql-connector-python. You can download this Python SQL module using pip:

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

Note that MySQL is a server-side database management system. One MySQL server can store multiple databases. Unlike SQLite, where a connection is equivalent to creating a database, creating a MySQL database consists of two stages:

- Establishing a connection to the MySQL server.
- Execute a query to create a database.

Let's define a function that will connect to the MySQL server and return a connection object:

```
importmysql.connector
frommysql.connectorimporterror

def create_connection(host_name, user_name, user_password):
connection=None
 try:
connection = mysql.connector.connect(
host=host_name,
user=user_name,
passwd=user_password
)
print("Connection to MySQL DB successful")
 excepterrorase:
print(f"The error '{e}' occurred")

 returnconnections

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

In the code above, we have defined a new create_connection() function that takes three parameters:

1. host_name
2. username
3. user_password

The mysql.connector module defines the connect() method used in the seventh line to connect to the MySQL server. Once the connection is established, the connection object is returned to the calling function. On the last line, the create_connection() function is called with the hostname, username, and password.

So far we have only established a connection. There is no base yet. To do this, we will define another function, create_database() , which takes two parameters:

connection object;

query is a string query to create a database.

Here's what that function looks like:

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

The cursor object is used to execute queries.
Let's create the sm_app database for our application on the MySQL server:

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

We now have a database on the server. However, the connection object returned by the create_connection() function is connected to the MySQL server. And we need to connect to the sm_app database. To do this, you need to change create_connection() as follows:

```
def create_connection(host_name, user_name, user_password, db_name):
connection=None
 try:
connection = mysql.connector.connect(
host=host_name,
user=user_name,
passwd=user_password,
database=db_name
)
print("Connection to MySQL DB successful")
 excepterrorase:
print(f"The error '{e}' occurred")

 returnconnections
```

The create_connection() function now accepts an additional parameter called db_name. This parameter specifies the name of the database we want to connect to. The name can now be passed when calling a function:

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

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

PostgreSQL

As with MySQL, there is no module in the Python standard library for PostgreSQL to interact with the database. But for this task there is a solution - the psycopg2 module:

```
pip install psycopg2
```

Let's define the create_connection() function to connect to a PostgreSQL database:

```
frompsycopg2importOperationalError

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")
 exceptOperationalErrorase:
print(f"The error '{e}' occurred")
 returnconnections
```

The connection is made through the psycopg2.connect() interface. Next, we use the function we wrote:

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

Now, inside the default postgres database, we need to create the sm_app database. The corresponding create_database() function is defined below:

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

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

By running the above script, we will see the sm_app database on our PostgreSQL server. Let's connect to it:

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

Here 127.0.0.1 and 5432 are respectively the IP address and port of the server host.

3. Create tables

In the previous section, we saw how to connect to SQLite, MySQL, and PostgreSQL database servers using different Python libraries. We have created the sm_app database on all three DB servers. In this section, we'll look at how to create tables within these three databases.

As discussed earlier, we need to fetch and link four tables:

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

SQLite

SQLite uses the cursor.execute() method to execute queries. In this section, we will define the execute_query() function that uses this method. The function will take a connection object and a query string. Next, the query string will be passed to the execute( ) method. In this section, we will use it to generate tables, and in the following sections, we will use it to perform update and delete queries.

Note. The script described below is part of the same file in which we described the connection to the SQLite database.

So let's start by defining the execute_query() function:

```
def execute_query(connection, query):
```

```
cursor = connection.cursor()
 try:
cursor.execute(query)
connection.commit()
print("Query executed successfully")
 excepterrorase:
print(f"The error '{e}' occurred")
```

Now let's write the passed query (query):

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

The query says to create a users table with the following five columns:

- id
- name
- age
- gender
- nationality

Finally, to make the table appear, we call execute_query(). We pass in the connection object we described in the previous section, along with the create_users_table query string we just prepared:

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

The following query is used to create the posts table:

```
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)
);
"""
```

Because there is a one-to-many relationship between users and posts, there is a user_id key in the table that refers to the id column in the users table. Run the following script to build the posts table:

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

Finally, we form the comments and likes tables with the following script:

```
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 may have noticed that creating tables in SQLite is very similar to usingpure SQL. All you have to do is store the request in a string variable and then pass that variable to cursor.execute().

MySQL

Just like with SQLite, to create a table in MySQL, you need to pass a query to cursor.execute(). Let's create a new version of the execute_query() function:

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

Describing the users table:

```
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 a foreign key relationship in MySQL is slightly different from SQLite. Moreover, MySQL uses the AUTO_INCREMENT keyword to specify columns whose values are automatically incremented when new records are inserted.
The following script will create a posts table containing a user_id foreign key that refers to the column id of the users table:

```
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, we pass the corresponding CREATE queries to the execute_query() function.

PostgreSQL

Using the psycopg2 library in execute_query() also involves working with a cursor:

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

We can use this feature to organize tables, insert, modify and delete records in your PostgreSQL database.

Let's create a users table inside the sm_app database:

```
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)
```

The query to create a users table in PostgreSQL is slightly different from SQLite and MySQL. Here, the SERIAL keyword is used to specify auto-increment columns. In addition, the way to specify references to foreign keys is different:

```
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)
```

4. Paste records

In the previous section, we covered how to deploy tables in SQLite, MySQL, and PostgreSQL databases using various Python modules. In this section, we will learn how to insert records.

SQLite

To insert records into a SQLite database, we can use the same execute_query() function we use to create tables. To do this, you first need to store an INSERT INTO query as a string. Then you need to pass the connection object and the string query to execute_query(). For example, let's insert five records into the users table:

```
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)
```

Because we set the id column to auto-increment, we don't need to specify it additionally. The users table will be automatically populated with five records with id values from 1 to 5.
Let's insert six records into the posts table:

```
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 is important to note that the user_id column of the posts table is a foreign key that refers to a 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, we'll get an error message.
The following script inserts records into the comments and likes tables:

```
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
(16),
(2, 3),
(fifteen),
(5, 4),
(2, 4),
(4, 2),
(3, 6);
"""

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

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 an INSERT INTO query in a string and then use cursor.execute() to insert records.
We previously defined the execute_query() wrapper function that we used to insert records. We can use the same function:

```
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 the cursor.executemany() method, which takes two parameters:
A query string containing placeholders for the records to insert.
The list of records we want to insert.
Look at the following example, which inserts two records into the likes table:

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

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

Which approach to choose is up to you. If you're not very familiar with SQL, it's easier to use the executemany() cursor method.

PostgreSQL

In the previous subsection, we saw two approaches for inserting records into MySQL database tables. Psycopg2 uses the second approach: we pass an SQL query with placeholders and a list of records to the execute() method. Each entry in the list must be a tuple whose values correspond to the values of a column in the database table. This is how we can insert user records into the users table:

```
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 users list contains five user entries as tuples. We then create a string with five placeholder elements (%s) corresponding to the five user entries. The placeholder string is combined with a query that inserts records into the users table. Finally, the query string and user entries are passed to the execute() method.
The following script inserts records into the posts table:

```
posts = [
```

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

post_records =", ".join(["%s"] * len(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)
```

Using the same technique, you can insert records into the comments and likes tables.

5. Extracting data from records

SQLite

To select records in SQLite, cursor.execute() can be used again. However, after that, you will need to call the fetchall() cursor method. This method returns a list of tuples where each tuple is mapped to the corresponding row in the retrieved records. To simplify the process, let's write the execute_read_query() function:

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

This function takes a connection object and a SELECT query and returns the selected record.
SELECT
Let's select all records from the users table:

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

foruserinusers:
print(user)
```

In the script above, the SELECT query retrieves all users from the users table. The result is passed to the execute_read_query() function we wrote, which returns all records from the users table.

Note. It is not recommended to use SELECT * on large tables, as this can result in a large number of I/O operations, which increase network traffic.
The result of the above query looks like this:

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

In the same way, you can extract all records from the posts table:

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

forpostinposts:
print(post)
```

The output looks like this:

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

JOIN

You can also run more complex queries that involve JOIN operations to retrieve data from two related tables. For example, the following script returns the user IDs and names, along with a description of the posts posted by those users:

```
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)

forusers_postinusers_posts:
print(users_post)
```

Data output:

```
(one,'James','I am feeling very happy today')
(2,'Leila','The weather is very hot today')
(2,'Leila','I need some help with my work')
(one,'James','I am getting married')
```

```
(5,'Elizabeth','It was a fantastic game of tennis')
(3,'Brigitte','Anyone up for a late night party today?')
```

The following script returns all posts along with the post comments and the names of the users who posted the comments:

```
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
)

forposts_comments_userinposts_comments_users:
print(posts_comments_user)
```

The output looks like this:

```
('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')
```

It is clear from the output that the column names were not returned by the fetchall() method. To return the column names, you need to take the description attribute of the cursor object. For example, the following list returns all the column names for the above query:

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

column_names = [description[0]fordescriptionincursor.description]
print(column_names)
```

The output looks like this:

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

WHERE

Now we will execute a SELECT query that returns the text of the post and the total number of likes it received:

```
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)

forpost_likeinpost_likes:
print(post_like)
```

The output is the following:

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

That is, using a WHERE query, you can return more specific results.

MySQL

The process of selecting records in MySQL is exactly the same as the process of selecting records in SQLite:

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

Now select all records from the users table:

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

foruserinusers:
print(user)
```

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

PostgreSQL

The process of selecting records from a PostgreSQL table using the psycopg2 module is also similar to SQLite and MySQL. We use cursor.execute() again, then the fetchall() method to select records from the table. The following script selects all records from the users table:

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

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

foruserinusers:
print(user)
```

Again, the result will be similar to what we saw before.

6. Update table entries

SQLite

Updating records in SQLite looks pretty easy. Again, execute_query() can be used. As an example, let's update the text of a post with an id of 2. First, let's create a description for the SELECT:

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

post_description = execute_read_query(connection, select_post_description)

fordescriptioninpost_description:
print(description)
```

We will see the following output:

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

The following script will update the description:

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

execute_query(connection, update_post_description)
```

Now, if we execute the SELECT query again, we will see the following result:

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

That is, the record has been updated.

MySQL

The process of updating records in MySQL using the mysql-connector-python module is an exact copy of the sqlite3 module:

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

execute_query(connection, update_post_description)
```

PostgreSQL

PostgreSQL update query is similar to SQLite and MySQL.

7. Deleting table entries

SQLite

As an example, let's remove a comment with an id of 5:

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

Now, if we retrieve all the records from the comments table, we can see that the fifth comment has been removed. The delete process in MySQL and PostgreSQL is identical to SQLite:

Conclusion

In this tutorial, we figured out how to use three common Python libraries to work with relational databases. Having learned to work with one of the sqlite3, mysql-connector-python and psycopg2 modules, you can easily transfer your knowledge to other modules and operate on any of the SQLite, MySQL and PostgreSQL databases.

There are also libraries for working with SQL andobject-relational mappings, such asSQLAlchemyandDjango ORM, which automate the tasks of Python interacting with databases. If you are interested in working with databases using Python, write about it in the comments - we will prepare additional materials.