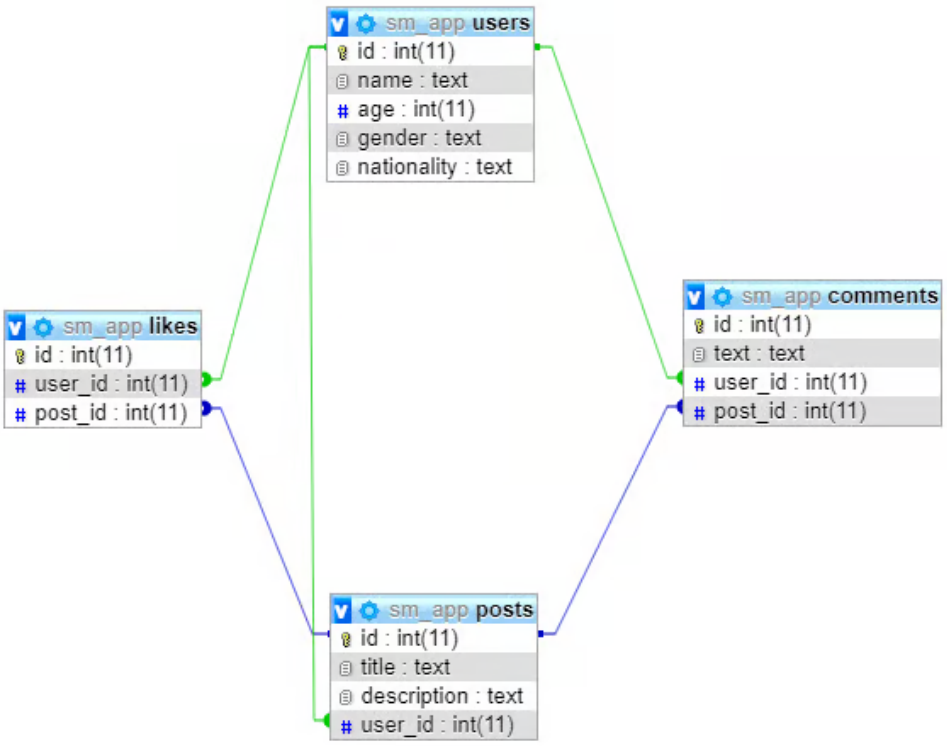
SQLite from python

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.

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.

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:

```
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:

```
connection = create_connection("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.

Creating Tables

As discussed earlier, you'll create four tables:

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

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.

```
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**:

```
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:

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

See the other tables creation in the example code.

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()`.

Inserting Records

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:

```
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)
```

See the other tables creation in the example code.

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.

Selecting Records

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()`:

```
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:

```
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.

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:

```
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)
```

WHERE

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

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

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

Updating Table Records

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:

```
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)
```

The following script updates the description:

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

execute_query(connection, update_post_description)
```

Deleting Table Records

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`:

```
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.

Reference:

https://realpython.com/python-sql-libraries/