SQlite and Python tutorial for:

CREATING AND PROCESSING WEB FORMS WITH FLASK

Mateo Prigl

SQLite

SQLite is a C-language library that implements a small, fast, self-contained, high-reliability, full-featured, SQL database engine. Let's go over some SQL basics needed for this course **"Creating and Processing Web Forms with Flask"**.
SQL is a standard language for storing, manipulating and retrieving data in databases. It is has a similar syntax for every big database engine.

Relational databases store information in tables.
Uppercased words are pure SQL commands. Lowercase words are customizable. We can change those, since they represent things like table and column names.

Here are some basic SQLite queries:

CREATE TABLES

```
CREATE TABLE table_name (column_1_name column_1_type, column_2_name column_2_type...);
```

e.g.

```
CREATE TABLE items (id INTEGER, name TEXT);
```

***items** table*

id	name

INSERT DATA

```
INSERT INTO table_name (column_1_name, column_2_name) VALUES (value_1, value_2);
```

e.g.

```
INSERT INTO items (id, name) VALUES (2, "Book");
```

***items** table*

id	name
2	Book

You can add `PRIMARY KEY` and `AUTOINCREMENT` keywords to the `id` field while creating the table, like so:

```
CREATE TABLE items (id INTEGER PRIMARY KEY AUTOINCREMENT, name TEXT);
```

This will automatically increment id on each insertion, so you don't have to define id when inserting an item into the table. Primary key means that this column is unique and it uniquely identifies each row.

UPDATE DATA

```
UPDATE table_name SET column_n_name = value_1 WHERE column_n_name = value_2;
```

e.g.

```
UPDATE items SET name = "Novel" WHERE id = 2;
```

***items** table*

id	name
2	Novel

SELECT (READ) DATA

```
SELECT column_n_name, column_n_name... FROM table_name;
```

e.g.

```
SELECT id, name FROM items;
```

Gets you back all of the rows:

***items** table*

id	name
2	Novel

The same result would be if you just put a '*' wildcard which represents all columns:

```
SELECT * FROM items;
```

Gets you back all of the rows:

***items** table*

id	name
2	Novel

But you choose which columns you want to select and the result will get back all of the rows with the columns you chose.

You can also use the `WHERE` statement here:

```
SELECT * FROM items WHERE name = "Novel";
```

The `ORDER BY` will sort the result by the column you choose. It can also do it in reverse, for this you can add `ASC` or `DESC` after the column name (ascending or descending).

```
SELECT * FROM items WHERE name = "Novel" ORDER BY id DESC;
```

You can also use logical operators like `AND` and `OR` to connect more statements.

```
SELECT * FROM items WHERE name = "Novel" AND id = 2;
```

DELETE DATA

```
DELETE FROM table_name WHERE column_n_name = value;
```

e.g.

```
DELETE FROM items WHERE id = 2;
```

***items** table*

id	name

DROP (DELETE) TABLE

```
DROP TABLE table_name;
```

e.g.

```
DROP TABLE items;
```

You can also create a condition to only delete the table if it exists.

```
DROP TABLE IF EXISTS items;
```

FOREIGN KEYS

We can also define relationships between more tables. This is the true power of relational databases. Let's say we have an `items` table and each item belongs to some `category`.

***items** table*

id	name	category
1	Apple	Fruit
2	Hammer	Tools

A lot of these items will belong to same categories, so instead of storing the same string multiple times, we can create another database which will store all of the categories.

***categories** table*

id	name
1	Fruit
2	Tools

Now we can just create a reference column inside of the `items` table, instead of storing strings, we can now store `id`'s from the `categories` table.

***items** table*

id	name	category_id
1	Apple	1
2	Hammer	2

We can connect this `category_id` column with the `id` column from the `categories` database with the help of `FOREIGN KEYS`. `FOREIGN KEY` will define the connection. So if we insert some item which has a `category_id` = 4, this will result in an error, because this category does not exist in the `categories` table.

Here is the SQL for creating these tables:

```
CREATE TABLE categories (
                        id INTEGER PRIMARY KEY AUTOINCREMENT,
            name TEXT
);

CREATE TABLE items (
                        id INTEGER PRIMARY KEY AUTOINCREMENT,
            name TEXT,
            category_id INTEGER,
            FOREIGN KEY(category_id) REFERENCES categories(id)
);
```

INNER JOIN

There are different kinds of joins in SQL, but in this course we will be interested just in `INNER JOIN`. Let's say that you want to print out the rows from the `items` table we created in the previous example.

```
SELECT id, name, category_id FROM items;
```

This will print out:
***items** table*

id	name	category_id
1	Apple	1
2	Hammer	2

But this is not enough. We don't want to just see the category id. We want to see the name of the category. `INNER JOIN` will join two tables, you just need to define how to join them. This is done after the `ON` keyword and it is usually the joining of foreign key columns like `items.category_id = categories.id` .

So this...

```
SELECT *
FROM items
INNER JOIN categories ON items.category_id = categories.id;
```

... will join the tables togather on the appropriate columns.

***items** table joined with categories*

id	name	category_id	id	name
1	Apple	1	1	Fruit
2	Hammer	2	2	Tools

Instead of printing all of the columns, you can pick which one to print. Columns with the same name in both tables have to be prefixed with the table name. Since this is the case, it's better to prefix all of the columns when doing `INNER JOIN`, just to be sure.

```
SELECT items.id, items.name, categories.name
FROM items
INNER JOIN categories ON items.category_id = categories.id;
```

***items** table joined with categories*

items.id	items.name	categories.name
1	Apple	Fruit
2	Hammer	Tools

This looks nicer. Now we have the names printed out next to the appropriate items without the need to store the category name strings in the same table.

One more thing. Since these joins can have more tables, the queries can get long and messy. You can use `AS` to create aliases for each table and use those aliases throughout the query.

```
SELECT i.id, i.name, c.name
FROM items AS i
INNER JOIN categories AS c ON i.category_id = c.id;
```

COUNT

You can get the amount of rows which are returned by the query.

```
SELECT COUNT(*) FROM items WHERE name = "Apple"
```

This will give back `1`.

LIKE

The `LIKE` keyword can be used with `WHERE` to find substrings in some column. For example, you want all of the rows which have the name that starts with `App`.

```
SELECT id, name FROM items WHERE name LIKE 'App%';
```

***items** table*

id	name
1	Apple

SQLite and Python

Now that we glanced over the SQLite syntax, we can see how to use it in Python. Python can query the SQLite database with the help of the **sqlite3** package.

```
import sqlite3
```

Frist we need to open the connection to the database.

```
conn = sqlite3.connect("some.db")
```

In this example "some.db" is the name of the database file. SQLite can store the whole database in one *db* file. If the database does not exist, this method will first create it and then establish the connection to it.

We can then take the cursor from the connection.

```
c = conn.cursor()
```

The cursor let's us interact with the database. It allows us to use methods like `execute`, to directly execute SQL queries.

For example we can create a table like this:

```
c.execute("""CREATE TABLE items (
                                                                id INTEGER PRIMARY KEY AUTOINCREMENT,
                                                                title TEXT
)        """)
```

You will probably execute more queries like this inside of the code. Python will sometimes gather all of the queries and execute them all at once, instead of pinging the database for each query. If you want to be sure that the queries are commited at some specific moment, you can commit them manually.

```
conn.commit()
```

When you are done, you should close the connection.

```
conn.close()
```

SELECT (READ) data

Values that will be passed to the query should be inside of the tuple.

```
item_id = 2

c.execute("SELECT * FROM items WHERE id = ?", (item_id,) )
```

Even if you have just one value, this value needs to be inside of the tuple. Inside of the query strings, you should use the question mark placeholders. These placeholders will then be replaced by the passed tuple, in the order of the elements from that tuple. The reason we do this is because of the security. We will go over this in the third module. For now you should know that every database engine has some kind of syntax similar to this one, to prevent the possible SQL injection. Just remember that you should never append the values from the code directly to the query, always use parameterized queries like this.

You can now get the result of the query from the cursor.

```
result = c.fetchone()
# (1, "Title")
```

This will give us back a tuple containing the values from the selected row. Of course, sometimes you'll want to select more rows. In that case you should use the `fetchall` method.

```
result = c.fetchall()
# [ (1, "Title"), (2, "Title 2") ]
```

This will give us back the list of tuples, each tuple representing one row from the database.

If you want to iterate through the list of tuples, you can do it directly, without the `fetchall` method.

```
for row in result:
        print("ID:    ", row[0])
        print("Title: ", row[1])

#       ID:    1
# Title: Title
# ID:    2
# Title: Title 2
```

You can insert data in the same way, just use the parameterized queries. We can use the method `executemany` to execute the same query on more tuples we pass from the list.

```
items = [
        ("Title 3", ),
        ("Title 4", )
]

c.executemany("""INSERT INTO items
                                                                (title)
                                                                VALUES (?)""", items)
conn.commit()
```

We don't need to pass the id, it is incremented automatically.

Scripts from the course

There are two database scripts inside of the db folder. These will be added after the seventh lesson of the first module. The **db_init.py** creates and initializes the database.

I imported the `os` package to get the full path to the **db** folder inside of the project tree.

```
import sqlite3
import os

db_abs_path = os.path.dirname(os.path.realpath(__file__)) + '/globomantics.db'
conn = sqlite3.connect(db_abs_path)
c = conn.cursor()
```

This way we can run the script from any directory inside of the terminal and the database will still be placed inside of the **db** folder. Now that we opened the connection, we can drop the tables if they exist. This will reset the whole database and will always give us the fresh start, if something goes wrong.

```
c.execute("DROP TABLE IF EXISTS items")
c.execute("DROP TABLE IF EXISTS categories")
c.execute("DROP TABLE IF EXISTS subcategories")
c.execute("DROP TABLE IF EXISTS comments")
```

Now we can create the tables.

```
c.execute("""CREATE TABLE categories(
                    id              INTEGER PRIMARY KEY AUTOINCREMENT,
                    name            TEXT
)""")

c.execute("""CREATE TABLE subcategories(
                    id              INTEGER PRIMARY KEY AUTOINCREMENT,
                    name            TEXT,
                    category_id     INTEGER,
                    FOREIGN KEY(category_id) REFERENCES categories(id)
)""")

c.execute("""CREATE TABLE items(
                    id              INTEGER PRIMARY KEY AUTOINCREMENT,
                    title           TEXT,
                    description     TEXT,
                    price           REAL,
                    image           TEXT,
                    category_id     INTEGER,
                    subcategory_id  INTEGER,
                    FOREIGN KEY(category_id) REFERENCES categories(id),
                    FOREIGN KEY(subcategory_id) REFERENCES subcategories(id)
)""")

c.execute("""CREATE TABLE comments(
                    id              INTEGER PRIMARY KEY AUTOINCREMENT,
                    content         TEXT,
                    item_id         INTEGER,
                    FOREIGN KEY(item_id) REFERENCES items(id)
)""")
```

This syntax should be familiar from SQLite.

I referenced four tables here. We will need one table to store the **items**. Each item has an **id**, **title**, **description**, **price**, **image** and the appropriate **category** and **subcategory**. The last two fields will just be a reference to another two tables. One for **categories** and one for **subcategories**. This way we can always add as much categories as we want and the only thing we need to do in the **items** table is to reference the id of the needed category. So these two fields are actually **foreign keys**, pointing to other tables. Same goes for the **subcategory** table, but each subcategory belongs to some category, so we need another foreign key there. Finally we have the **comments** table. This table holds all of the comments and the **item_id** is a foreign key to the item the comment belongs to.

After that I inserted some initial random data to the tables and wrote a message that the database is initialized.

The second script **show_tables.py** will let us list all of the rows from a specified table. We first take in the table choice from the user.

```
print("Options: (items, comments, categories, subcategories, all)")
table = input("Show table: ")
```

We can then execute the appropriate method based on the user input.

```
if table == "items":
    show_items()
elif table == "comments":
    show_comments()
elif table == "categories":
    show_categories()
elif table == "subcategories":
    show_subcategories()
elif table == "all":
    show_items()
    show_comments()
    show_categories()
    show_subcategories()
else:
    print("This option does not exist.")
```

Each method will execute a select query on a given table and print out each row to the terminal.

We can put this query inside of the try block. If something goes wrong (maybe someone forgot to create the database), we will print out the appropriate message and close the connection.

And that's it. Hope this short tutorial got you up to speed.

Happy learning!

Mateo