Name: Natiola, Henry Jay P.
Course and Section: CPE 019 - CPE32S9
Date of Submission: 01/31/24
Instructor: Engr. Roman Richard

# Working with Python and SQLite

Objectives:
Use the sqlite3 module to interact with a SQL database.
Access data stored in SQLite using Python.
Describe the difference in interacting with data stored as a CSV file versus in SQLite.
Describe the benefits of accessing data using a database compared to a CSV file.

## Required Resources

1 PC with Internet access
Jupyter notebook

## SQL refresh

Very brief introduction to relational databases (temporary): http://searchsqlserver.techtarget.com/definition/relational-database More videos on relational databases: https://www.youtube.com/watch?v=jyju2P-7hPA&list=PLAwxTw4SYaPm4R6j_wzVOCV9fJaiQDYx4 Introduction to SQL: http://www.w3schools.com/sql/sql_intro.asp Working with SQLite via the command-line: https://www.sqlite.org/cli.html

# Part 1: Python and SQL

When you open a CSV in python, and assign it to a variable name, you are using your computers memory to save that variable. Accessing data from a database like SQL is not only more efficient, but also it allows you to subset and import only the parts of the data that you need.

**The sqlite3 module**
The sqlite3 module provides a straightforward interface for interacting with SQLite databases. A connection object is created using sqlite3.connect(); the connection must be closed at the end of the session with the .close() command. While the connection is open, any interactions with the database require you to make a cursor object with the .cursor() command. The cursor is then ready to perform all kinds of operations with .execute().

**Step 1: Create a SQL connection to our SQLite database**
Creating a new SQLite database is as simple as creating a connection using the sqlite3 module in the Python standard library. To establish a connection all you need to do is pass a file path to the connect(...) method in the sqlite3 module, and if the database represented by the file does not exists one will be created at that path.

```
import sqlite3
con = sqlite3.connect('sqlite.db')
```

You will find that in everyday database programming you will be constantly creating connections to your database, so it is a good idea to wrap this simple connection statement into a reusable generalized function.

```
import os
import sqlite3

# create a default path to connect to and create (if necessary) a database
# called 'database.sqlite3' in the same directory as this script
DEFAULT_PATH = os.path.join('sqlite.db')

def db_connect(db_path=DEFAULT_PATH):
    con = sqlite3.connect(db_path)
    return con
```

**Step 2: Create a table on the SQLite database**
The code below creates a table on the sqlite.db database. The cursor() command is needed to make a cursor object to interact with the created database. The cursor is then ready to perform all kinds of operations with .execute(). The execute() command performs a query that creates a table using the parameters as shown. The commit() command

In order to create database tables you need to have an idea of the structure of the data you are interested in storing. There are many design considerations that go into defining the tables of a relational database. To aid in the discussion of SQLite database programming with Python, we will be working off the premise that a database needs to be created for a fictitious book store that has the below data already collected on book sales.

| Customer | Date | Product | Price |
|---|---|---|---|
| Allan Turing | 2/22/1944 | Introduction to Combinatorics | 7.99 |
| Donald Knuth | 7/3/1967 | A Guide to Writing Short Stories | 17.99 |
| Donald Knuth | 7/3/1967 | Data Structures and Algorithms | 11.99 |
| Edgar Codd | 1/12/1969 | Advanced Set Theory | 16.99 |

Upon inspecting this data, it is evident that it contains information about customers, products, and orders. A common pattern in database design for transactional systems of this type are to break the orders into two additional tables, orders and line items (sometimes referred to as order details) to achieve greater normalization.

Enter the SQL for creating the customers and products tables follows:

```
con = db_connect() # connect to the database
cur = con.cursor() # instantiate a cursor object
customers_sql = """CREATE TABLE customers(id integer PRIMARY KEY, first_name text NOT NULL, last_name text NOT NULL)"""
cur.execute(customers_sql)
products_sql = """CREATE TABLE products (id integer PRIMARY KEY,name text NOT NULL,price real NOT NULL)"""
cur.execute(products_sql)
```

```
<sqlite3.Cursor at 0x782a83e2d4c0>
```

The above code creates a connection object then uses it to instantiate a cursor object. The cursor object is used to execute SQL statements on the SQLite database.

With the cursor created, we write the SQL to create the customers table, giving it a primary key along with a first and last name text field and assign it to a variable called customers_sql. Then we call the execute(...) method of the cursor object passing it the customers_sql variable. Similar steps were don for the products table.

You can query the sqlite_master table, a built-in SQLite metadata table, to verify that the above commands were successful.

To see all the tables in the currently connected database query the name column of the sqlite_master table where the type is equal to "table".

```
cur.execute("SELECT name FROM sqlite_master WHERE type='table'")

print(cur.fetchall())
```

```
[('customers',), ('products',)]
```

To get a look at the schema of the tables query the sql column of the same table where the type is still "table" and the name is equal to "customers" and/or "products".

```
cur.execute("""SELECT sql FROM sqlite_master WHERE type='table'AND name='customers'""")
print(cur.fetchone()[0])
```

```
CREATE TABLE customers(id integer PRIMARY KEY, first_name text NOT NULL, last_name text NOT NULL)
```

The next table to define will be the orders table which associates customers to orders via a foreign key and the date of their purchase. Since SQLite does not support an actual date/time data type (or data class to be consistent with the SQLite vernacular) all dates will be represented as text values.

```
orders_sql = """
... CREATE TABLE orders (
...     id integer PRIMARY KEY,
...     date text NOT NULL,
...     customer_id integer,
...     FOREIGN KEY (customer_id) REFERENCES customers (id))"""
cur.execute(orders_sql)
```

```
<sqlite3.Cursor at 0x782a83e2d4c0>
```

The final table to define will be the line items table which gives a detailed accounting of the products in each order.

```
lineitems_sql = """
... CREATE TABLE lineitems (
...     id integer PRIMARY KEY,
...     quantity integer NOT NULL,
...     total real NOT NULL,
...     product_id integer,
...     order_id integer,
...     FOREIGN KEY (product_id) REFERENCES products (id),
...     FOREIGN KEY (order_id) REFERENCES orders (id))"""
cur.execute(lineitems_sql)
```

```
<sqlite3.Cursor at 0x782a83e2d4c0>
```

**Step 3: Loading the Data**

In this section we will use INSERT to our sample data into the tables just created. A natural starting place would be to populate the products table first because without products we cannot have a sale and thus would not have the foreign keys to relate to the line items and orders. Looking at the sample data, we see that there are four products:

1. Introduction to Combinatorics - 7.99
2. A Guide to Writing Short Stories -17.99
3. Data Structures and Algorithms - 11.99
4. Advanced Set Theory - 16.99

The workflow for executing INSERT statements is simply:

1. Connect to the database
2. Create a cursor object
3. Write a parameterized insert SQL statement and store as a variable
4. Call the execute method on the cursor object passing it the sql variable and the values, as a tuple, to be inserted into the table

Given this general outline let us write some more code.

```
con = db_connect()
cur = con.cursor()
product_sql = "INSERT INTO products (name, price) VALUES (?, ?)"
cur.execute(product_sql, ('Introduction to Combinatorics', 7.99))
cur.execute(product_sql, ('A Guide to Writing Short Stories', 17.99))
cur.execute(product_sql, ('Data Structures and Algorithms', 11.99))
cur.execute(product_sql, ('Advanced Set Theory', 16.99))
con.commit()
```

The insert statement follows the standard SQL syntax except for the ? bit. The ?'s are actually placeholders in what is known as a "parameterized query".

Parameterized queries are an important feature of essentially all database interfaces to modern high level programming languages such as the sqlite3 module in Python. This type of query serves to improve the efficiency of queries that are repeated several times. Perhaps more important, they also sanitize inputs that take the place of the ? placeholders which are passed in during the call to the execute method of the cursor object to prevent nefarious inputs leading to SQL injection.

To populate the remaining tables we are going to follow a slightly different pattern to change things up a bit. The workflow for each order, identified by a combination of customer first and last name and the purchase date, will be:

1. Insert the new customer into the customers table and retrieve its primary key id
2. Create an order entry based off the customer id and the purchase date then retrieve its primary key id
3. For each product in the order determine its primary key id and create a line item entry associating the order and the product
4. To make things simpler on ourselves let us do a quick look up of all our products. For now do not worry too much about the mechanics of the SELECT SQL statement as we will devote a section to it shortly.

```
cur.execute("SELECT id, name, price FROM products")
formatted_result = [f"{id:<5}{name:<35}{price:>5}" for id, name, price in cur.fetchall()]
id, product, price = "Id", "Product", "Price"
print('\n'.join([f"{id:<5}{product:<35}{price:>5}"] + formatted_result))
```

```
Id   Product                            Price
1    Introduction to Combinatorics       7.99
2    A Guide to Writing Short Stories   17.99
```

```
3    Data Structures and Algorithms     11.99
4    Advanced Set Theory                16.99
```

The first order was placed on Feb 22, 1944 by Alan Turing who purchased Introduction to Combinatorics for $7.99.

Start by making a new customer record for Mr. Turing then determine his primary key id by accessing the lastrowid field of the cursor object.

```
customer_sql = "INSERT INTO customers (first_name, last_name) VALUES (?, ?)"
cur.execute(customer_sql, ('Alan', 'Turing'))
customer_id = cur.lastrowid
print(customer_id)
con.commit()
```

```
1
```

**Task 1: Insert 3 more records on the customers table**

Insert the following records:

1. Donald Knuth
2. Edgar Codd
3. Martin Forest

```
customers1_sql = "INSERT into customers (first_name, last_name) VALUES (?, ?)"
cur.execute(customers1_sql, ('Donald', 'Knuth'))
cur.execute(customers1_sql, ('Edgar', 'Codd'))
cur.execute(customers1_sql, ('Martin', 'Forest'))
```

```
<sqlite3.Cursor at 0x782a83e2e1c0>
```

```
cur.execute("SELECT * FROM customers")
for row in cur:
  print (row[0], row[1], row[2])
```

```
1 Alan Turing
2 Donald Knuth
3 Edgar Codd
4 Martin Forest
```

We can now create an order entry, collect the new order id value and associate it to a line item entry along with the product Mr. Turing ordered.

```
order_sql = "INSERT INTO orders (date, customer_id) VALUES (?, ?)"
date = "1944-02-22" # ISO formatted date
cur.execute(order_sql, (date, customer_id))
order_id = cur.lastrowid
print(order_id)
con.commit()
```

```
1
```

**Task 2: Insert 3 more records on the orders table**

Insert the following records:

1. for Donald Knuth, date is 7/3/1967
2. Edgar Codd, date is 1/12/1969
3. Martin Forest, date is 1/15/2021

```
order_sql = "INSERT INTO orders (date, customer_id) VALUES (?, ?)"
cur.execute(order_sql, ("1967-07-03", 2))
cur.execute(order_sql, ("1969-01-12", 3))
cur.execute(order_sql, ("2021-01-15", 4))
```

```
<sqlite3.Cursor at 0x782a83e2e1c0>
```

```
cur.execute("SELECT * FROM orders")
for row in cur:
  print (row[0], row[1], row[2])
```

```
1 1944-02-22 1
2 1967-07-03 2
3 1969-01-12 3
4 2021-01-15 4
```

Each order can be inserted into the lineitems as shown below.

```
li_sql = """INSERT INTO lineitems
... (order_id, product_id, quantity, total)
... VALUES (?, ?, ?, ?)"""
product_id = 1
cur.execute(li_sql, (order_id, 1, 1, 7.99))
con.commit()
```

```
cur.execute("SELECT * FROM lineitems")
for row in cur:
  print (row[0], row[1], row[2], row[3], row[4])
```

```
1 1 7.99 1 1
```

```
cur.execute("DELETE from lineitems WHERE id = 2")
```

```
<sqlite3.Cursor at 0x782a83e2e1c0>
```

The remaining records are loaded exactly the same except for the order made to Donald Knuth, which will receive two line item entries.

**Task 3: Insert 3 more records on the lineitems**

Insert the following records:

1. for Donald Knuth, insert (order_id, 2, 2, 17.99)
2. Edgar Codd, insert (order_id, 3, 3, 11.99)
3. Martin Forest, insert (order_id, 4, 4, 10.99)

```
li1_sql = """INSERT INTO lineitems
... (order_id, product_id, quantity, total)
... VALUES (?, ?, ?, ?)"""
cur.execute(li1_sql, (2, 2, 2, 17.99))
cur.execute(li1_sql, (3, 3, 3, 11.99))
cur.execute(li1_sql, (4, 4, 4, 10.99))
con.commit()
```

```
cur.execute("SELECT * FROM lineitems")
for row in cur:
  print (row[0], row[1], row[2], row[3], row[4])
```

```
1 1 7.99 1 1
2 2 17.99 2 2
3 3 11.99 3 3
4 4 10.99 4 4
```

**Step 3: Querying the Database**

Generally the most common action performed on a database is a retrieval of some of the data stored in it via a SELECT statement. For this section, we will be demonstrating how to use the sqlite3 interface to perform simple SELECT queries.

To perform a basic multirow query of the customers table you pass a SELECT statement to the execute(...) method of the cursor object. After this you can iterate over the results of the query by calling the fetchall() method of the same cursor object.

```
cur.execute("SELECT * FROM customers")
results = cur.fetchall()
for row in results:
    print(row)
```

```
(1, 'Alan', 'Turing')
(2, 'Donald', 'Knuth')
(3, 'Edgar', 'Codd')
(4, 'Martin', 'Forest')
```

Lets say you would like to instead just retrieve one record from the database. You can do this by writing a more specific query, say for Donald Knuth's id of 2, and following that up by calling fetchone() method of the cursor object.

```
cur.execute("SELECT id, first_name, last_name FROM customers WHERE id = 2")
result = cur.fetchone()
print(result)
```

```
(2, 'Donald', 'Knuth')
```

See how the individual row of each result is in the form of a tuple? Well while tuples are a very useful Pythonic data structure for some programming use cases many people find them a bit hindering when it comes to the task of data retrieval. It just so happens that there is a way to represent the data in a way that is perhaps more flexible to some. All you need to do is set the row_factory method of the connection object to something more suitable such as sqlite3.Row. This will give you the ability to access the individual items of a row by position or keyword value.

```
con.row_factory = sqlite3.Row
cur = con.cursor()
cur.execute("SELECT id, first_name, last_name FROM customers WHERE id = 1")
result = cur.fetchone()
id, first_name, last_name = result['id'], result['first_name'], result['last_name']
print(f"Customer: {first_name} {last_name}'s id is {id}")
```

```
Customer: Alan Turing's id is 1
```

**Supplementary Activity:**

1. Create a database and call it user.db
2. Create a table named "users" and insert the following: (id int, name TEXT, email TEXT)
3. Insert the following data:
   (1, 'Jonathan','jvtaylar@gmail.com'),
   (2, 'John','jonathan@gmail.com'),
   (3,'cpeEncoders','encoders@gmail.com')
4. Select all data from users.
5. Select id = 3 from users.
6. Update user id = 3 name and set it to "James."
7. Insert the following data: (4, 'Cynthia','cynthia@gmail.com')
8. Delete id = 4 from users.
9. Display all contents in a formatted way.

```
#Creating a database and call it  user.db
import os
import sqlite3
DEFAULT_PATH = os.path.join('user.db')

def db_connect(db_path=DEFAULT_PATH):
  con = sqlite3.connect(db_path)
  return con

con = db_connect()
cur = con.cursor()

# Create a table named "users" and insert the following: (id int, nameTEXT, email TEXT).
users_sql = """
CREATE TABLE IF NOT EXISTS users (
id integer PRIMARY KEY,
name text NOT NULL,
email text NOT NULL)"""
cur.execute(users_sql)

# Insert the following data: (1, 'Jonathan','jvtaylar@gmail.com'), (2,'John','jonathan@gmail.com'), (3,'cpeEncoders','encoders@gmail.com')
insertusers_sql = "INSERT INTO users (name, email) VALUES (?, ?)"
cur.execute(insertusers_sql, ('Jonathan', 'jvtaylar@gmail.com'))
cur.execute(insertusers_sql, ('John', 'johnatan@gmail.com'))
cur.execute(insertusers_sql, ('cpeEncoders', 'encoders@gmail.com'))

# Select all data from users.
cur.execute("SELECT * from users")
```

```
# Select id = 3 from users.
cur.execute("SELECT * from users WHERE id = 3")

# Update user id = 3 name and set it to "James."
cur.execute("UPDATE users set name = 'James' WHERE id = 3")

# Insert the following data: (4, 'Cynthia','cynthia@gmail.com')
cur.execute(insertusers_sql, ('Cynthia', 'cynthia@gmail.com'))

# Delete id = 4 from users.
cur.execute("DELETE from users where id = 4")

# Display all contents in a formatted way.
cur.execute("SELECT id, name, email FROM users")
formatted_result = [f"{id:<5}{name:<15}{email:>5}" for id, name, email
in cur.fetchall()]
id, name, email = "ID", "Name", "Email"
print('\n'.join([f"{id:<5}{name:<15}{email:>5}"] + formatted_result))
```

```
ID   Name           Email
1    Jonathan       jvtaylar@gmail.com
2    John           jonathan@gmail.com
3    James          encoders@gmail.com
5    John           johnatan@gmail.com
6    cpeEncoders    encoders@gmail.com
7    Cynthia        cynthia@gmail.com
```

**Conclusions/Observations:**

I conclude that the activity is hard since I forgot the execution for the sql in the database that I have in my previous year. But I was able to accomplish and finish the task that is in the activity. There are error that I encounter during the activity that the database is closing so the solution that I had came out with is by creating or copying a new google colab, then the error that I have is solved by just doing it. So overall the activity that we had is just a recap for the SQL that we already had.