Python SQLite Address Book CLI

Contents

Time required: 180 minutes

- Comment each line of code as show in the tutorials and other code examples.
- Follow all directions carefully and accurately.
- Think of the directions as minimum requirements.

SQL Tutorial

- https://www.w3schools.com/sql/sql_intro.asp
- https://www.w3schools.com/sql/sql_syntax.asp
- https://www.w3schools.com/sql/sql_create_db.asp
- https://www.w3schools.com/sql/sql_create_table.asp
- https://www.w3schools.com/sql/sql_drop_table.asp
- https://www.w3schools.com/sql/sql_insert.asp
- https://www.w3schools.com/sql/sql_update.asp
- https://www.w3schools.com/sql/sql_delete.asp
- https://www.w3schools.com/sql/sql_select.asp

SQLite Address Book Tutorial

Model View Controller (MVC)

This tutorial breaks the program into MVC (Model View Controller)

Model–view–controller (MVC) is a software architectural pattern commonly used for developing user interfaces that divide the related program logic into three interconnected elements. This is done to separate internal representations of information from the ways information is presented to and accepted from the user.

Model

The central component of the pattern. It is the application's dynamic data structure, independent of the user interface. It directly manages the data, logic and rules of the application.

View

Any representation of information such as a chart, diagram, or table. Multiple views of the same information are possible, such as a bar chart for management and a tabular view for accountants.

Controller

Accepts input and converts it to commands for the model or view.[15]

In addition to dividing the application into these components, the model–view–controller design defines the interactions between them.

- The model is responsible for managing the data of the application. It receives user input from the controller.
- The view renders presentation of the model in a particular format.
- The controller responds to the user input and performs interactions on the data model objects. The controller receives the input, optionally validates it and then passes the input to the model.

From Wikipedia.

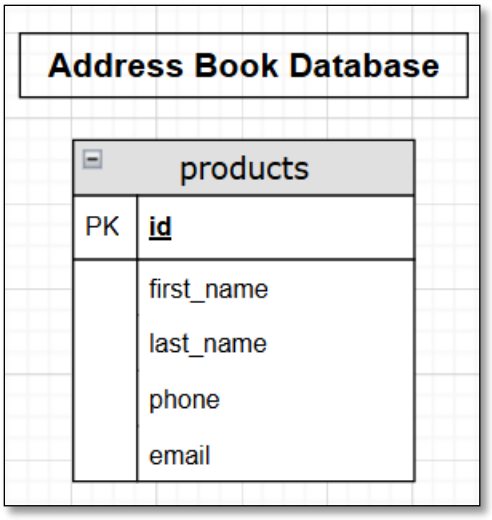
Model: address_book.db (Database)

View: address_book.py (User interface)

Controller: db_operations.py (This contains the SQL code that works with the database.)

Address Book Database ERD

An ERD (Entity Relationship Diagram) is a way to plan your database. It shows the Primary and Foreign Keys and fields in the dictionary along with relationships. This is a single table database, there aren't any relationships or foreign keys.



Data Dictionary

A data dictionary names and gives the data types of all the fields.

Field Name	Data Type	Description
id	int	Primary Key
first_name	text	First Name
last_name	text	
phone	text	
email	Text	

Explanation: How to Create the Database

When the **AddressBook** object is initialized, it creates a **DBOperations** object with a database file name parameter.

```
7  # Import database operations library
8  import db_operations
9
10
11 class AddressBook:
12     def __init__(self):
13         # Create db_operations object
14         # This creates database and table if not exists
15         self.db_op = db_operations.DBOperations("address_book.db")
```

The database is created if one doesn't exist, or a connection is created to the existing database.

```
 8 # Import SQLite library to work with databases
 9 import sqlite3
10
11
12 class DBOperations:
13     def __init__(self, database: str):
14         self.database = database
15         self.debugging = False
```

The following are the fields and data types for our **address_book** table. This tutorial will not implement the **phone_number** field. That is left as an assignment for you at the end of this tutorial.

Field names	**Data types**
id (primary key)	INTEGER
first_name	TEXT
last_name	TEXT
phone_number	TEXT

An SQL statement is a string. We will use a multiline strings to make it easier to read the SQL statement.

If the **address_book** table does not exist, this SQL statement will create the table along with the three data fields.

```
        # Create the address_book table if it doesn't exist
        SQL = """
            CREATE TABLE IF NOT EXISTS tbl_address_book(
            id          INTEGER PRIMARY KEY,
            first_name  TEXT,
            last_name   TEXT,
            phone       TEXT,
            email       TEXT
        )"""
```

Tutorial 1: address_book_cli.py

Let's get started with our first interactive SQL database program.

Create a Python program named **address_book_cli.py**

address_book_cli.py starts by importing the **db_operations** library. We will create the db_operations file in a minute.

```
"""
    Name: address_book_cli_1_create_db.py
    Author: William Loring
    Created: 01/05/22
    Purpose: MVC Address book CLI with SQLite
"""
# Import database operations library
import db_operations


class AddressBook:
    def __init__(self):
        # Create db_operations object
        # This creates database and table if not exists
        self.db_op = db_operations.DBOperations("address_book.db")
        self.db_op.create_table()
        # Print program title
        print("+------------------------------+")
        print("|      Bill's Contact List     |")
        print("+------------------------------+")
```

Tutorial 2: Menu (address_book_cli.py)

We need a menu to test the methods and for the user to use the program. This is only part of the menu. We will add other methods as we go through this tutorial.

```
# ---------------------------- MENU ----------------------------------#
    def menu(self):
        """Program menu"""
        USER_CHOICE = """
(C)reate contact
(R)etrieve contacts
(U)pdate contact
(D)elete contact
(Q)uit
-->> """
        # Get user input
        user_input = input(USER_CHOICE)
        # Convert input into lower case
        user_input = user_input.lower()
        # Menu loop while user does not input q
        while user_input != "q":
            # Get user input
            user_input = input(USER_CHOICE)
            # Convert input into lower case
            user_input = user_input.lower()
```

At the end of the file, we create an address_book object and display the menu.

```
# ---------------------------- START PROGRAM ----------------------------#
address_book = AddressBook()
address_book.menu()
```

Tutorial 3: db_operations.py

This is the controller file. This file makes changes to the database.

Create a Python file named **db_operations.py**

When this file is imported and an object is created from it in the address book program, it creates a reference to the database file and address book table. It then calls the **create_table()** method.

```
1  """
2      Name: db_operations.py
3      Author: William Loring
4      Created: 01/05/22
5      SQLite database query execution
6      Controller
7  """
8  # Import SQLite library to work with databases
9  import sqlite3
10
11
12 class DBOperations:
13     def __init__(self, database: str):
14         self.database = database
15         self.debugging = False
```

Tutorial 4: create_table() (db_operations.py)

The create_table method creates a table if one does not already exist. Comment out Line 21 after you are sure your tables are correct. Otherwise, your table will be deleted and recreated each time you run the program.

```
17 # -------------------- CREATE TABLE ---------------------------------------#
18     def create_table(self):
19         """Create database and table if not exists"""
20         # Drop table if needed to change table structure
21         SQL = "DROP TABLE IF EXISTS tbl_address_book"
22         self.execute_sql(SQL)
23         # Create the address_book table if it doesn't exist
24         SQL = """
25             CREATE TABLE IF NOT EXISTS tbl_address_book(
26             id          INTEGER PRIMARY KEY,
27             first_name  TEXT,
28             last_name   TEXT,
29             phone       TEXT,
30             email       TEXT
31         )"""
32         self.execute_sql(SQL)
```

Tutorial 5: execute_sql (db_operations.py)

This method executes the SQL statements against the database. Some SQL statements will have parameters, some will not.

```python
# -------------------- EXECUTE SQL ----------------------------------------#
    def execute_sql(self, SQL: str, parameters: tuple = None):
        # This is an overloaded method in Python, parameters is optional
        # If everything inside the with sqlite3.connect is successful
        # connect.commit() and connect.close() are automatically called
        # when the with statement exits
        # If DATABASE does not exist, it is created
        try:
            with sqlite3.connect(self.database) as connection:
                # Create cursor to work with SQL
                cursor = connection.cursor()
                if parameters is not None:
                    # Execute SQL with parameters
                    cursor.execute(SQL, parameters)
                else:
                    # Execute SQL without parameters
                    cursor.executescript(SQL)
                # Records are written automatically
                # after the with statement exits
                # All connections are closed
        except Exception as e:
            print(f"There was an SQLite error: {e}")
```

Tutorial 6: insert_record (address_book_cli.py)

1. Get input from the user.
2. Call the controller method **insert_record** with two arguments.

The user interface (view) doesn't do any work, it calls methods in the **db_op** object.

```python
# ------------------------- INSERT RECORD --------------------------------#
    def insert_record(self):
        """Insert new record"""
        # Get input from user
        first_name = input("Enter first name: ")
        last_name = input("Enter last name: ")
        phone = input("Enter phone: ")
        email = input("Enter email: ")
        # Call controller insert record method with arguments
        self.db_op.insert_record(
            first_name,
            last_name,
            phone,
            email
        )
```

Tutorial 7: insert_record() (db_operations.py)

The insert_records() from the address book program passes the arguments to this method and inserts the new record.

```
34 # ------------------------ INSERT RECORD ---------------------------------#
35     def insert_record(
36         self,
37         first_name: str,
38         last_name: str,
39         phone: str,
40         email: str
41     ):
42         """Insert new record"""
43         SQL = """
44             INSERT INTO tbl_address_book
45             VALUES(NULL, ?, ?, ?, ?)
46         """
47         # Parameters are a tuple of variables or values
48         # They are mapped to the ? ? placeholders of the query
49         parameters = (
50             first_name,
51             last_name,
52             phone,
53             email
54         )
55         self.execute_sql(SQL, parameters)
```

Try It Out

At this point, the program should run. You should be able to insert a record. We can't see that this happened as we haven't implemented a way to see the records.

Explanation: Fetch Records

- **SELECT * FROM table** is a query that selects all records and fields from the database.
- The return from the database query is a cursor object which contains a list of tuples as shown in the example below.

Each tuple aligns with a database record.

```
 id, first_name, last_name
(4,  'Laurie',   'Loring')
```

The following is an example of the tuples (records) fetched from a sample database.

```
[
    (4, 'Laurie', 'Loring'),
    (5, 'Fred', 'Flounder'),
    (6, 'Sammy', 'Shark'),
    (7, 'Larry', 'Lungfish'),
    (8, 'Howdy', 'Doody')
]
```

To print out the data nicely, we iterate through the list with the variable record, then use the indexes to get the specific fields from each tuple.

- record[0] is the id
- record[1] is the first_name
- record[2] is the last_name

Tutorial 8: address_book_cli.py (fetch_all_records())

```
"""
    Name: address_book_cli3_fetch.py
    Author: William Loring
    Created: 01/05/22
    Purpose: MVC Address book CLI with SQLite
"""
# pip install tabulate
import tabulate
# Import database operations library
import db_operations
```

Start by adding the method calls to **fetch_all_records()** to the menu.

```
# ---------------------------- MENU ------------------------------------#
    def menu(self):
        """Program menu"""
        USER_CHOICE = """
(C)reate contact
(R)etrieve contacts
(U)pdate contact
(D)elete contact
(Q)uit
-->> """
        # Get user input
        user_input = input(USER_CHOICE)
        # Convert input into lower case
        user_input = user_input.lower()
        # Menu loop while user does not input q
        while user_input != "q":
            # Insert contact record
            if user_input == "c":
                self.insert_record()
                self.fetch_all_records()
            # List all contacts
            elif user_input == "r":
                self.fetch_all_records()
            else:
                print("Unknown option. Please try again")

            # Get user input
            user_input = input(USER_CHOICE)
            # Convert input into lower case
            user_input = user_input.lower()
```

Tutorial 9: fetch_all_records (address_book_cli.py)

This view method calls the **fetch_all_records** from the **dboperations** file and displays the results.

```
71 # ------------------------ FETCH ALL RECORDS ------------------------------#
72     def fetch_all_records(self):
73         """Fetch all records"""
74         # Call controller fetch all records method
75         all_records = self.db_op.fetch_all_records()
76         print("\nBill's Contact List")
77         print()
78         # Records are returned as a list of tuples
79         # Iterate through all records one record namedtuple at a time
80         # Use tablulate library to format the data nicely
81         print(
82             tabulate.tabulate(
83                 all_records,
84                 headers=["id", "First Name", "Last Name", "Phone", "Email"],
85                 tablefmt="psql"  # Table format
86             )
87         )
```

Tutorial 10: fetch_all_records (db_operations.py)

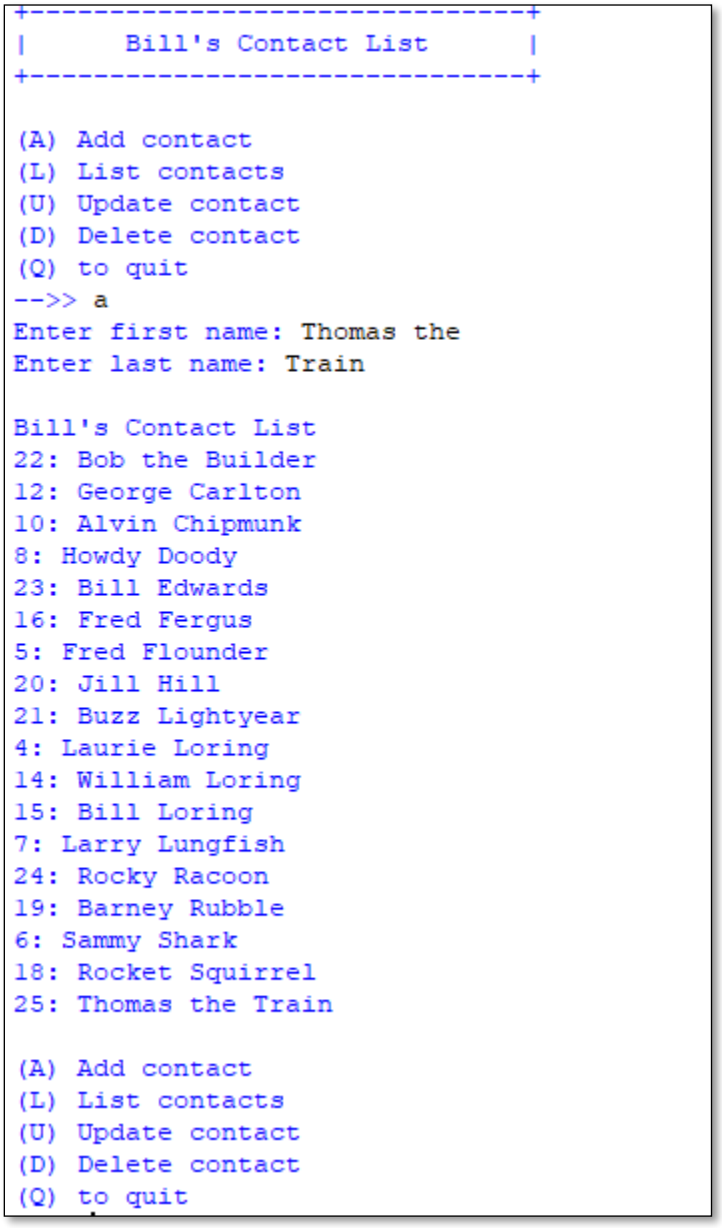
This controller method selects all records from the database and returns the records as a list of tuples to the view.

```
57 # ---------------------- FETCH ALL RECORDS ------------------------------#
58     def fetch_all_records(self):
59         """Fetch all records"""
60         # Query to get all contacts
61         # SELECT * FROM selects all records from a table
62         # sorted by last name
63         # desc (descending) order for GUI Treeview
64         # asc (ascending) order for CLI
65         SQL = """
66             SELECT * FROM tbl_address_book
67             ORDER BY last_name asc
68         """
69         with sqlite3.connect(self.database) as connection:
70             cursor = connection.cursor()
71             # fetchall() fetches the records returned by the SQL
72             # statement as a list of tuples
73             records = cursor.execute(SQL).fetchall()
74         if records:
75             return records
```

Try It Out

We have a functional program. We can add and display records. They are sorted by last name by the controller.

Example run:



Explanation: Update Record

We use a primary key to identify specific records. The primary key is unique and isn't duplicated. A primary key typically has no meaning, other than it is a unique value. Autoincrementing integers are typical primary keys.

- **UPDATE table** identifies the table we want to work with.

- The **id** field (primary key) from the user identifies the specific record we want to update. The **id** field is the third parameter.
- **WHERE id is ?** identifies the specific record.
- **SET first_name = ?** updates the first name field according to the first element of the parameters.

```
UPDATE table SET first_name = ? WHERE id is ?
```

The query and the parameters combine to update a specific record and field.

Tutorial 11: update_record (address_book_cli.py)

Let's update the menu to include the **update_record** method.

```
# ---------------------------- MENU ----------------------------------------#
    def menu(self):
        """Program menu"""
        USER_CHOICE = """
(C)reate contact
(R)etrieve contacts
(U)pdate contact
(D)elete contact
(Q) to quit
-->> """
        # Get user input
        user_input = input(USER_CHOICE)
        # Convert input into lower case
        user_input = user_input.lower()
        # Menu loop while user does not input q
        while user_input != "q":
            # Insert contact record
            if user_input == "c":
                self.insert_record()
                self.fetch_all_records()
            # List all contacts
            elif user_input == "r":
                self.fetch_all_records()
            # Update selected contact
            elif user_input == "u":
                self.fetch_all_records()
                self.update_record()
                self.fetch_all_records()
            else:
                print("Unknown option. Please try again")

            # Get user input
            user_input = input(USER_CHOICE)
            # Convert input into lower case
            user_input = user_input.lower()
```

The **update_record** method sends the updated information as arguments to the controller.

```
94  # -------------------------- UPDATE RECORD --------------------------------#
95      def update_record(self):
96          """Update selected record"""
97          # Get input from user
98          id = int(input("Enter contact id: "))
99          # Get input from user
100         first_name = input("Enter first name: ")
101         last_name = input("Enter last name: ")
102         phone = input("Enter phone: ")
103         email = input("Enter email: ")
104         # Call controller insert record method with arguments
105         self.db_op.update_record(
106             first_name,
107             last_name,
108             phone,
109             email,
110             id
111         )
```

Tutorial 12: update_record (db_operations.py)

This controller method accepts the arguments from the view to update the database.

```python
# ---------------------- UPDATE RECORD ------------------------------------#
    def update_record(
        self,
        first_name: str,
        last_name: str,
        phone: str,
        email: str,
        id: int
    ):
        """Update selected record by id"""
        SQL = """
            UPDATE tbl_address_book
            SET first_name = ?,
            last_name = ?,
            phone = ?,
            email = ?
            WHERE id = ?
        """
        # Parameters are a tuple of variables or values
        # They are mapped to the ? ? ? ? ? in the query
        parameters = (
            first_name,
            last_name,
            phone,
            email,
            id
        )
        self.execute_sql(SQL, parameters)
```

Try It Out

Try out the update record menu item.

Example run:

```
+--------------------------------+
|      Bill's Contact List       |
+--------------------------------+

(A) Add contact
(L) List contacts
(U) Update contact
(D) Delete contact
(Q) to quit
-->> a
Enter first name: Jeff
Enter last name: Airplane

Bill's Contact List
2: Jeff Airplane
1: George Jones

(A) Add contact
(L) List contacts
(U) Update contact
(D) Delete contact
(Q) to quit
-->> u

Bill's Contact List
2: Jeff Airplane
1: George Jones
Enter contact id: 2
Enter first name: Jefferson
Enter last name: Airplane

Bill's Contact List
2: Jefferson Airplane
1: George Jones

(A) Add contact
(L) List contacts
(U) Update contact
(D) Delete contact
(Q) to quit
```

Explanation: Delete Record

The **id** field (primary key) allows us to uniquely identify the record to delete. Notice that as you delete and add records that the **id** value is not reused.

Tutorial 13: delete_record (address_book_cli.py)

This method uses the id field to identify and delete the record.

```
# ---------------------------- MENU ------------------------------------#
    def menu(self):
        """Program menu"""
        USER_CHOICE = """
(C)reate contact
(R)etrieve contacts
(U)pdate contact
(D)elete contact
(B)ackup database to SQL
(Q)uit
-->> """
        # Get user input
        user_input = input(USER_CHOICE)
        # Convert input into lower case
        user_input = user_input.lower()
        # Menu loop while user does not input q
        while user_input != "q":
            # Insert contact record
            if user_input == "c":
                self.insert_record()
                self.fetch_all_records()
            # List all contacts
            elif user_input == "r":
                self.fetch_all_records()
            # Update selected contact
            elif user_input == "u":
                self.fetch_all_records()
                self.update_record()
                self.fetch_all_records()
            elif user_input == "d":
                self.fetch_all_records()
                self.delete_record()
                self.fetch_all_records()
            elif user_input == "b":
                self.db_op.database_dump()
            else:
                print("Unknown option. Please try again")

            # Get user input
            user_input = input(USER_CHOICE)
            # Convert input into lower case
            user_input = user_input.lower()
```

```
# ------------------------ DELETE RECORD ---------------------------------#
    def delete_record(self):
        """Delete selected record"""
        # Get primary key from user
        id = int(input("Enter contact id to delete: "))
        # Call controller delete record method with argument
        self.db_op.delete_record(id)
```

Tutorial 14: delete_record (db_operations.py)

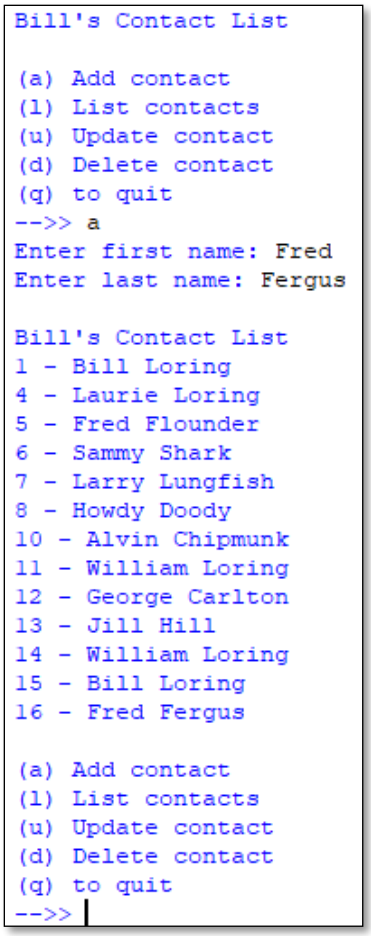
The id field is passed into the delete_record method.

```
106 # ---------------------- DELETE RECORD ----------------------------------#
107     def delete_record(self, id: int):
108         """Delete selected record by id"""
109         SQL = """
110             DELETE FROM tbl_address_book
111             WHERE id = ?
112         """
113         # Parameters are a tuple of variables or values
114         # They are mapped to the ? in the query
115         parameters = (
116             id,
117         )
118         self.execute_sql(SQL, parameters)
119 
120 # -------------------- DATABASE DUMP TO SQL FILE --------------------------#
121     def database_dump(self):
122         try:
123             with sqlite3.connect(self.database) as connection:
124                 # Iterate through database, print SQL
125                 for line in connection.iterdump():
126                     print(line)
127 
128                 # Use with context manager to write and close/save the file
129                 with open("database_dump.sql", "w") as file:
130                     # Iterate through database, write SQL to file
131                     for line in connection.iterdump():
132                         file.write(f"{line}\n")
133                 print("File written to disk.")
134         except Exception as e:
135             print(f"There was an SQLite error: {e}")
```

Test the Complete Program

You can perform CRUD operations on your address book database.

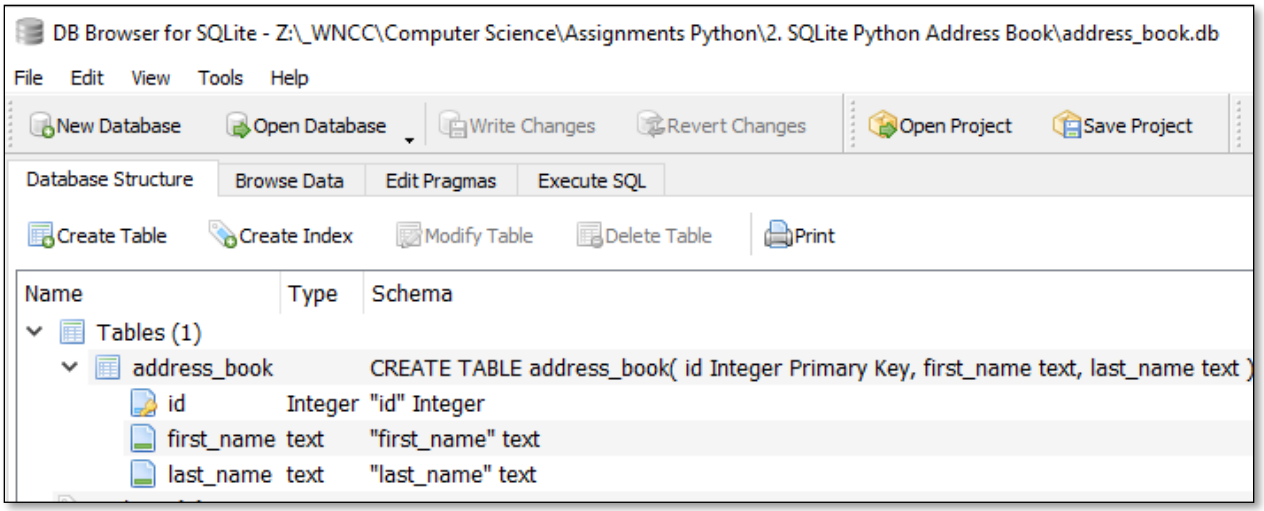
Example run:



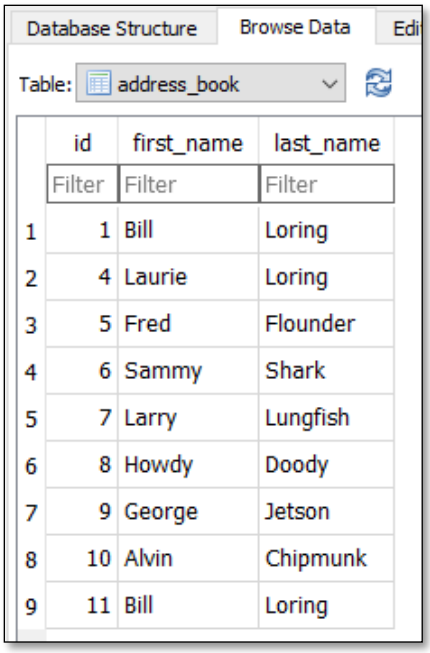
SQLite Database Browser

This is a handy tool to look at, troubleshoot, and manipulate your database.

1. Go to https://sqlitebrowser.org
2. Go to the **Download** tab.
3. Download the **Windows PortableApp**.
4. Unzip the file.
5. In the folder you will find **SQLiteDatabaseBrowserPortable.exe**
6. Double Click the file. Click OK on the warning.
7. Use the **Open Database** button to open your database.



Click the **Browse Data** tab to see your records.



Click the **Close Database** button when you are done.

Assignment: Enhance the Address Book Program

In this and the previous tutorials, we learned the fundaments of SQLite and SQL in Python with a limited data set. Let's expand this program to include one more field. Unless you are going to calculate with a number, use the TEXT data type to store it.

- Another contact field

NOTE: Copy this program to a new folder before you start adding the new field. If you break the database with the additions, you can always go back.

Assignment Submission

1. Attach the pseudocode.
2. Attach the program files.
3. Attach screenshots showing the successful operation of the program.
4. Submit in Blackboard.