COMP2152 LAB MANUAL

OPEN SOURCE DEVELOPMENT

This booklet will help the reader understand the concepts, principles, and implementation of the Python programming language. By the end of the booklet, the reader will be able to code comfortably in Python.

Prepared by Ben Blanc

TABLE OF CONTENTS

CHAPTER 11

DATABASE CONNECTION

A database connection is a facility in computer science that allows client software to communicate with database server software, whether on the same machine or not. Read more

SQLITE

SQLite is a self-contained, high-reliability, embedded, full-featured, public-domain, SQL database engine. SQLite is the most used database engine in the world. Read more

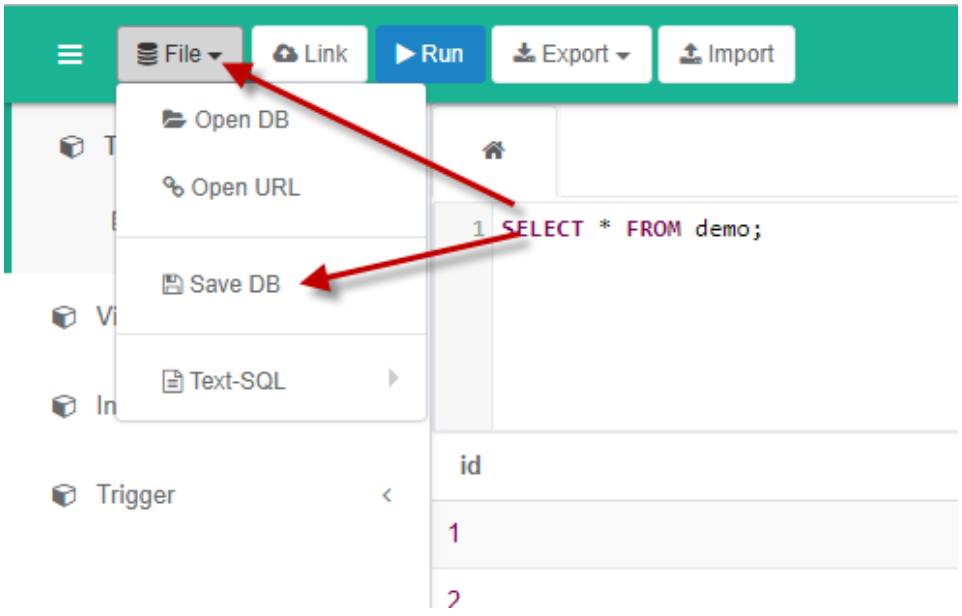
Python 3 has native support for the sqlite3 database engine. You must create a sqlite3 database file. Then python can open and manipulate the database via the file.

CREATE AND SAVE AN SQLITE3.DB FILE

Navigate to https://sqliteonline.com/

Click on the **File Tab**

Click on the **Save DB** option



CHAPTER 11 AT A GLANCE

In this chapter you will learn how to connect to a SQLITE database and execute queries. You will also learn how to catch potential exceptions.

IMPORTING SQLITE FILE TO PYTHON PROJECT

Next, move the downloaded sqlite.db file from your Downloads location to your Python project by dragging on dropping the file.

CONNECTING TO THE SQLITE FILE

To connect to the sqlite file, we use the **connect()** method of sqlite3 module.

```
import sqlite3

db_file = "sqlite.db"

db_con = sqlite3.connect(db_file)
```

Be aware, if the db file doesn't exists, python will create the file but it will not have the appropriate database information.

CLOSE SQLITE CONNECTION

To close the connection, you need to destroy the object by ensuring that all remaining references to it are deleted. You do this by using the close() method of the connection variable.

```
import sqlite3

db_file = "sqlite.db"

db_con = sqlite3.connect(db_file)

db_con.close()
```

SQLITE CURSOR OBJECT

A cursor object is required to execute any operations on the sqlite database. The cursor object can be obtained by using the **cursor()** method of the database connection variable:

```
import sqlite3

db_file = "sqlite.db"

db_con = sqlite3.connect(db_file)

cursor = db_con.cursor()

db_con.close()
```

.

RETRIEVING DATA

After connecting to the database and getting the database cursor object, you will want to retrieve information from the table

SELECT WITHOUT PARAMETER

Below is how you would select **all** the data of the DEMO table:

```
import sqlite3

db_file = "sqlite.db"

db_con = sqlite3.connect(db_file)

cursor = db_con.cursor()

query = "select * from demo"

cursor.execute(query)

all_rows = cursor.fetchall()

for row in all_rows :
    print(row)

db_con.close()
```

You can a query to the **execute()** method of the cursor object. Then you fetch all the results.

To fetch the result of the first row your query matches, use the **fetchone()** method of the cursor object.

```
query = "select * from demo"

cursor.execute(query)

first_match = cursor.fetchone()

print(first_match)
```

You can also fetch a range of rows using the **fetchmany()** method of the cursor object.

```
query = "select * from demo"

cursor.execute(query)

first_three_rows = cursor.fetchmany(3)

for row in first_three_rows :
    print(row)
```

As with file manipulation, after using any fetch method, the cursor stays at the end of that row (after **fetechone()** or **fetchmany()**), or at the end of the file (after **fetchall()**).

DESCRIBING TABLE STRUCTURE

In sqlite, to describe a table, we use the following command:

PRAGMA table_info(table_name)

```
query = "pragma table_info(demo)"

cursor.execute(query)

field_info = cursor.fetchall()
for field in field_info :
    print("field",(field[0] + 1),
          "name =",field[1],
          "data type=",field[2])
```

Now that we know the field names and table types of demo, we can write a select statement with a parameter.

SELECT WITH PARAMETERS

Below is how you would select information with a parameter.

```
import sqlite3

db_file = "sqlite.db"

db_con = sqlite3.connect(db_file)

cursor = db_con.cursor()

query = "select * from demo where id > 2"

cursor.execute(query)

all_rows = cursor.fetchall()

for row in all_rows :
    print(row)

db_con.close()
```

Another example would be

```
query = "select * from demo where id > 2 and id <7"

cursor.execute(query)

all_rows = cursor.fetchall()

for row in all_rows :
    print(row)
```

PREPARED STATEMENTS

A prepared statement is an optimal way of adding or retrieving information from the database, when you would want to pass a variable value as a parameter to a query. For those cases, we use the second argument of the **execute()** method of the cursor object.

execute(sql, parameters)

parameters = a tuple of values

```
import sqlite3

db_file = "sqlite.db"

db_con = sqlite3.connect(db_file)

cursor = db_con.cursor()

value_1 = int(input("Enter first value: "))
value_2 = int(input("Enter second value: "))

query = "select * from demo where id > ? and id < ?"

cursor.execute(query, (value_1,value_2))

all_rows = cursor.fetchall()

for row in all_rows :
    print(row)

db_con.close()
```

In the above code, we put question marks (?) as placeholders for the values, then we place the values in an tuple and pass this tuple to the **execute()** method to run the query.

ROW COUNT

To get the number of rows returned, in a select query use the **len()** method on the fetched data.

```
query = "select * from demo where id > 2"

cursor.execute(query)

all_rows = cursor.fetchall()

num_rows = len(all_rows)

print (num_rows)
```

The **rowcount** property of the cursor object always returns -1 since python doesn't know how many results have been produced.

Getting Column Data

In retrieving information from a database, tuple's or a list of tuple in the return type by default. You can change this behavior by changing the **row_factory** property of the connection variable, to access the returned information by field name.

INDEX BASED INDICES

By default, you can access a field of a query result by indicating its column number.

```
import sqlite3

db_file = "sqlite.db"

db_con = sqlite3.connect(db_file)

query = "select * from demo where id = 7"

cursor = db_con.cursor()

cursor.execute(query)

row = cursor.fetchone()

print("ID=",row[0], "Name=", row[1],
      "Hint=", row[2])
```

Indices are ordered based on your select statement

```
import sqlite3

db_file = "sqlite.db"

db_con = sqlite3.connect(db_file)

query = "select hint,id, name from demo where id = 7"

cursor = db_con.cursor()

cursor.execute(query)

row = cursor.fetchone()

print("ID=",row[1], "Name=", row[2],
      "Hint=", row[0])
```

If more than one row is fetched, it would like the following:

```
import sqlite3

db_file = "sqlite.db"

db_con = sqlite3.connect(db_file)

query = "select * from demo where id in (7,8)"

cursor = db_con.cursor()

cursor.execute(query)

row = cursor.fetchall()

print("ID=",row[0][0], "Name=", row[0][1],
      "Hint=", row[0][2])

print("ID=",row[1][0], "Name=", row[1][1],
      "Hint=", row[1][2])
```

```
import sqlite3

db_file = "sqlite.db"

db_con = sqlite3.connect(db_file)

query = "select hint,id, name from demo where id in (7,8)"

cursor = db_con.cursor()

cursor.execute(query)

row = cursor.fetchall()

print("ID=",row[0][1], "Name=", row[0][2],
      "Hint=", row[0][0])

print("ID=",row[1][1], "Name=", row[1][2],
      "Hint=", row[1][0])
```

NAME BASED INDICES

To access a field by name instead of order, you must change the **row_factory** property of the connection variable

```
db_con.row_factory = sqlite3.Row
```

Below is a full example

```
import sqlite3

db_file = "sqlite.db"

db_con = sqlite3.connect(db_file)

db_con.row_factory = sqlite3.Row

query = "select * from demo where id = 10"

cursor = db_con.cursor()

cursor.execute(query)

row = cursor.fetchone()

print("ID=",row['id'], "Name=", row['name'],
      "Hint=", row['hint'])
```

If more than one row is fetched, it would like the following:

```
import sqlite3

db_file = "sqlite.db"

db_con = sqlite3.connect(db_file)

db_con.row_factory = sqlite3.Row

query = "select * from demo where id in (10, 11)"

cursor = db_con.cursor()

cursor.execute(query)

row = cursor.fetchmany(2)

print("ID=",row[0]['id'], "Name=", row[0]['name'],
      "Hint=", row[0]['hint'])

print("ID=",row[1]['id'], "Name=", row[1]['name'],
      "Hint=", row[1]['hint'])
```

INSERTING DATA

After connecting to database and selecting data, inserting information into the database is the next step you would take.

INSERT STATEMENT

You would insert data into that table with the following statement

```
import sqlite3

db_file = "sqlite.db"

db_con = sqlite3.connect(db_file)

cursor = db_con.cursor()

query = "insert into demo (id,name, hint) values('12', 'new 12th entry', '12th entry hint')"

cursor.execute(query)

db_con.commit()

db_con.close()
```

Notice that we use the **commit()** method of the database connection variable. This method is needed to make any change to a database.

Insert two other entries into your table

```
import sqlite3

db_file = "sqlite.db"

db_con = sqlite3.connect(db_file)

cursor = db_con.cursor()

query = "insert into demo  values('13', 'new 13th entry', '13th entry hint')"

cursor.execute(query)

db_con.commit()

db_con.close()
```

```
import sqlite3

db_file = "sqlite.db"

db_con = sqlite3.connect(db_file)

cursor = db_con.cursor()

query = "insert into demo values( ?, ?, ? )"
params = (14, 'new 14th entry', '14th entry hint')

cursor.execute(query, params)

db_con.commit()

db_con.close()
```

ROW COUNT

To get the number of rows affected by an insert or update query, use the **rowcount** property of the cursor object:

```
import sqlite3

db_file = "sqlite.db"

db_con = sqlite3.connect(db_file)

cursor = db_con.cursor()

query = "insert into demo values( ?, ?, ? )"
params = (15, 'new 15th entry', '15th entry hint')

cursor.execute(query, params)

print(cursor.rowcount)

db_con.commit()

db_con.close()
```

UPDATE DATA

To update a table, use the following code:

UPDATE STATEMENT

```
import sqlite3

db_file = "sqlite.db"

db_con = sqlite3.connect(db_file)

cursor = db_con.cursor()

query = "update demo set name = 'updated value' where id = 5"

cursor.execute(query)

db_con.commit()

db_con.close()
```

Another example of an update statement is:

```
import sqlite3

db_file = "sqlite.db"

db_con = sqlite3.connect(db_file)

cursor = db_con.cursor()

query = "update demo set name = ? , hint = ? where id = ?"

params = ("update part 2", 'hint updated', 7)

cursor.execute(query, params)

db_con.commit()

db_con.close()
```

ROW COUNT

To get the number of rows affected by an update query, use the **rowcount** property of the cursor object:

```
import sqlite3

db_file = "sqlite.db"

db_con = sqlite3.connect(db_file)

cursor = db_con.cursor()

query = "update demo set hint = ?"

params = ("default hint!",)

cursor.execute(query, params)

ans = input(str(cursor.rowcount)+ " affected, are you sure? ")

if ans == 1 or ans.lower()=='y' or ans.lower()=='yes' :
    db_con.commit()

db_con.close()
```

DELETE DATA

To delete data from a table, use the following code:

DELETE STATEMENT

```
import sqlite3

db_file = "sqlite.db"

db_con = sqlite3.connect(db_file)

cursor = db_con.cursor()

query = "delete from demo where id = 5"

cursor.execute(query)

db_con.commit()

db_con.close()
```

Another example of a remove statement is:

```
import sqlite3

db_file = "sqlite.db"

db_con = sqlite3.connect(db_file)

cursor = db_con.cursor()

query = "delete from demo where name = ? and hint = ? and id = ?"

params = ("update part 2", 'hint updated', 7)

cursor.execute(query, params)

db_con.commit()

db_con.close()
```

ROW COUNT

To get the number of rows affected by an remove query, use the **rowcount** property of the cursor object

```
import sqlite3

db_file = "sqlite.db"

db_con = sqlite3.connect(db_file)

cursor = db_con.cursor()

query = "delete from demo where id >  ?"

params = (10,)

cursor.execute(query, params)

ans = input(str(cursor.rowcount)+ " affected, are you sure? ")

if ans == 1 or ans.lower()=='y' or ans.lower()=='yes' :
    db_con.commit()

db_con.close()
```

WITH CLOSING STATEMENT

In the examples above, we have left the cursor object open. However, it is best practice to close the cursor object when we have finished our queries.

We can use the **close()** method of the cursor object

```
import sqlite3

db_file = "sqlite.db"

db_con = sqlite3.connect(db_file)

cursor = db_con.cursor()

query = "select * from demo"

cursor.execute(query)

data = cursor.fetchall()

print(data)

cursor.close()
```

Or we can use the **closing()** method of the contextlib module

```
import sqlite3
from contextlib import closing

db_file = "sqlite.db"

db_con = sqlite3.connect(db_file)

with closing(db_con.cursor()) as cursor :

    query = "select * from demo"

    cursor.execute(query)

    data = cursor.fetchall()

    print(data)
```

ERROR HANDLING

To catch any potential errors, you would use a try-except blocks with attempting to connect to the database and executing queries.

The exception you would catch is the OperationalError of the sqlite3 module.

Reminder, if the database file is not found, python will create an empty file. However, when trying to execute any query of this empty database will result in an OperationalError.

```
import sqlite3
from contextlib import closing

db_file = "sqlite.db"

db_con = sqlite3.connect(db_file)

try :
    with closing(db_con.cursor()) as cursor :

        query = "select * from demoing"

        cursor.execute(query)

        data = cursor.fetchall()

        print(data)

except sqlite3.OperationalError as e :
    print(str(e))
```