Chapter 21 SQLite with Python

Contents

Syllabus Learning Outcomes

3.3 Databases and Data Management

Understand, create and use SQL and NoSQL databases, as well as understand techniques to protect the privacy and integrity of data.

3.3.1 Determine the attributes of a database: table, record and field. 3.3.2 Explain the purpose of and use primary, secondary, composite and foreign keys in tables.
3.3.3 Explain with examples, the concept of data redundancy and data dependency.
3.3.4 Reduce data redundancy to third normal form (3NF).
3.3.5 Draw entity-relationship (ER) diagrams to show the relationship between tables.
3.3.6* Understand how NoSQL database management system addresses the shortcomings of relational database management system (SQL).
3.3.7* Explain the applications of SQL and NoSQL.
3.3.8*Use a programming language to work with both SQL and NoSQL

databases. *Note: NoSQL will be addressed in later chapter

1 Connecting to SQLite database with sqlite3

Python comes with sqlite3 module which allows working with SQLite databases using Python.

Generally, to work with the database,

1. We first **establish connection** to the database with `connect()` method
2. Execute some SQL statements, with `execute`() method
3. **Save** the changes we made to the database with the `commit`() method 4. **Close** the database using the `close`() method. This is similar to how we handle file I/O earlier.

Connect to database

```
1  import sqlite3
2
3  connection = sqlite3.connect("school.db")
4  connection.close()
```

To connect to a database:

Step 1: import sqlite3 module

Step 2: make a connection with database using sqlite3.connect.

Note: If the database of interest, for e.g. school.db, does not exist, an empty school.db database will be created.

Step 3: To ensure the database file is closed properly, use `connection.close().`

2 Execute CRUD operations

After loading an SQLite file and getting a connection, we can execute SQL statements by calling the connection object's `execute()` method with a `str` containing the SQL statement we wish to run. For instance, the following Python program creates a new table named `student` in a new SQLite database file named `school.db`:

Execute SQL statements

```
1  import sqlite3
2
3  connection = sqlite3.connect("school.db")
4  connection.execute("CREATE TABLE student " +
5   "(ExamNo INTEGER PRIMARY KEY," +  " Name TEXT,
6  Class TEXT)")
7  connection.close()
```

After running the program, we can open `school.db` using DB Browser to check that a `student` table was indeed created.

We can also make the statement into a string and parse the string as an argument to be executed.

Parse statement as a string

```
1   import sqlite3
2
3   connection = sqlite3.connect("school.db")
4
5   statement = """CREATE TABLE student (
6    ExamNo INTEGER PRIMARY KEY,
7    Name TEXT NOT NULL,
8    Class TEXT NOT NULL
9    ); """
10
11  connection.execute(statement)
12
13  connection.close()
```

If we try to run the program again, we will get the following error:

```
OperationalError: table student already exists
```


Any errors caused by running the SQL (such as the error telling us the `student` table already exists) are reported as Python exceptions and can be handled as usual.

To avoid this error, we can use “IF NOT EXISTS” when creating tables.

IF NOT EXISTS

```
1   import sqlite3
2
3   connection = sqlite3.connect("school.db")
4
5   statement = """CREATE TABLE IF NOT EXISTS student
6   (   ExamNo INTEGER PRIMARY KEY,
7    Name TEXT NOT NULL,
8    Class TEXT NOT NULL
9    ); """
10
11  connection.execute(statement)
12
13  connection.close()
```

Now, let us try using `INSERT` to put data into the `student` table:

Insert record

```
1   import sqlite3
2
3   connection = sqlite3.connect("school.db")
4
5   statement = "INSERT INTO student(ExamNo, Name, Class)"
6   +  "VALUES(2150601, 'Jack Neo', '21S56')"
7
8   connection.execute(statement)
9
10  connection.close()
11
```

This program runs with no errors. However, if we open `student.db` using DB Browser, we find that the inserted data is missing. What happened to the data? It was discarded because we did not save the changes we made to the database.

We can include “OR IGNORE” or “OR REPLACE” when inserting records to avoid errors associated with inserting duplicate records. For example,

```
INSERT OR IGNORE INTO student(ExamNo, Name, Class)VALUES(2150601, 'Jack Neo', '21S56')
```


3 Commit changes

It is important to run the `commit()` method to save the changes to the database. This is equivalent to the action Write Changes in DB Browser.

Commit method

```
1   import sqlite3
2
3   connection = sqlite3.connect("school.db")
4
5   statement = """INSERT INTO student(ExamNo, Name,
6   Class)  VALUES(2150601, 'Jack Neo', '21S56')"""
7
8   connection.execute(statement)
9
10  connection.commit()
11  connection.close()
```

Adding the `commit()` method in line 10, the data is inserted and saved

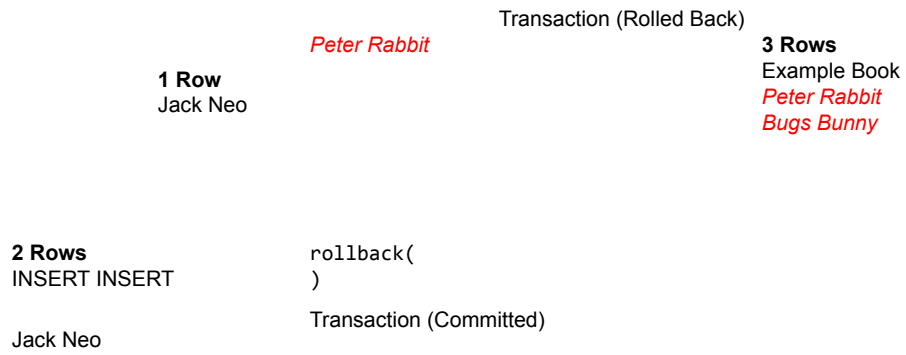
correctly. **4 Revert changes**

We can discard all the changes made since the last commit was executed by calling the connection object's `rollback()` method.

Using rollback to revert changes

```
1   import sqlite3
2
3   connection = sqlite3.connect("school.db")
4
5   connection.execute("INSERT INTO student(ExamNo, Name, Class)"
6   +  "VALUES(2150602, 'Peter Rabbit', '21S56')")
7   connection.execute("INSERT INTO student(ExamNo, Name, Class)"
8   +  "VALUES(2150603, 'Bugs Bunny, '21S56')")
9   connection.rollback()
10
11  connection.execute("INSERT INTO student(ExamNo, Name, Class)"
12  +  "VALUES(2150602, 'Mark Lee', '21S56')")
13  connection.commit()
14
15  connection.close()
```


In the above example, the first two `INSERT` statements are rolled back so they have no effect on the database. On the other hand, the last `INSERT` statement is committed so it *does* affect the database. This process is illustrated by the following diagram:



Jack Neo
Mark Lee

`commit()`

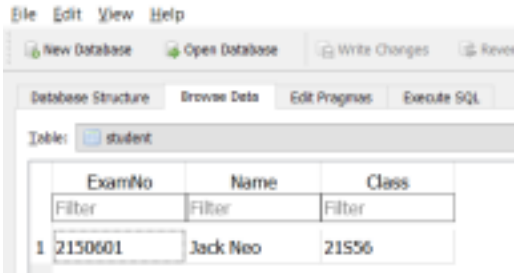
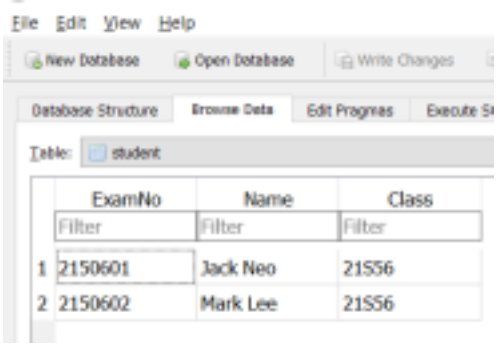
INSERT

1 Row
Jack Neo

2 Rows
Jack Neo
Mark Lee

2 Rows

Comparing the contents of the `student` table before and after running the program shows that the first two `INSERT` statements are indeed ignored.

Before	After
	

only committed transaction is saved

5 Enclosing user input

When generating SQL commands, we often need to include some data that is provided by the user. We may be tempted to use `str` concatenation in order to generate the required SQL command. Unfortunately, this is insecure as special characters or keywords in the user's input are not escaped and malicious users can take advantage

of this to inject his/her own SQL commands.

Instead, we should use *parameter substitution* to safely include data that is provided by the user. We use the question-mark character ? as a placeholder in the SQL for any data that is provided by the user. We then provide a second argument to execute, that is a tuple of values which will replace the placeholders. This ensures that the provided values are escaped properly and cannot be misinterpreted as SQL.

Using ? as placeholder

```
1  import sqlite3
2
3  connection = sqlite3.connect("school.db")
4
5  user_input = input("Enter Student Exam no to delete:
6
7  ") statement = "DELETE FROM student WHERE ExamNo = ?"
8
9  connection.execute(statement, (user_input,))
10
11 connection.commit()
12 connection.close()
```

The program above safely asks for the ExamNo of a student and deletes the corresponding row from the database.

Parameter substitution follows the same order in which the placeholders appear in the SQL. This is illustrated by the following diagram:

```
connection.execute("INSERT INTO student (ExamNo, Name, Class)
      VALUES (?, ?, ?)", (2150605, 'Mickey Mouse', '21S56'))
```


6 Retrieving data

As you have already learned, the SELECT command is used to select data from the

database. When you run a SELECT command in DB Browser, the selected rows are usually displayed in a table below the SQL statement that was executed.

selected rows appear here

Using Python sqlite3 module, we can take 5 approaches to retrieve the data.

Approach 1: Iterate the `Cursor` object, which is also created when we run `execute()` method on `Connection` object,

Approach 2: Use the `fetchone()` method of the `Cursor` object - returns a `tuple` of the columns in the current row.

Approach 3: Use the `fetchall()` method of the `Cursor` object - fetch all the rows at once and keep them in a list. This method returns a `list` of `tuples` with each `tuple` containing the selected columns for a single row.

Approach 4: Cast `Cursor` object as a `list` – converts `cursor` object to a `list` object

Approach 5: Setting the `row factory` attribute of the `Connection` object as `sqlite3.Row` object. This configures the SQLite connection so that each row is retrieved as a `dict,` mapping column names to values.

5 approaches to retrieve data

```
1   cursor = connection.execute("SELECT * FROM student")
2
3   #Approach 1
4   for row in cursor:
5       print(row)
6
7   #Approach 2
8   row = cursor.fetchone()
9   while row is not None:
10   print(row)
11      row = cursor.fetchone()
12
13  #Approach 3
14  rows = cursor.fetchall()
15  print(rows)
16
17  #Approach 4
18  rows = list(cursor)
19  print(rows)
20
21  #Approach 5
22  #setting the `row_factory` attribute of `Connection`
23  object connection.row_factory = sqlite3.Row
24
25  query = "SELECT * FROM student"
26
27  cursor = connection.execute(query)
28
29  rows = cursor.fetchall()
30
31  for row in rows: # row is now a dictionary
32   print(row["ExamNo"])
33   print(row["Name"])
34   print(row["Class"])
```


`sqlite3` Module Summary

Name	Description
`connect(`*`filename`*`)`	Creates a `Connection` object using SQLite file with the given filename

`Row`	Can be used as a `Connection` object's `row_factory` so `fetchone()` returns a `dict` mapping column names to field values instead of returning a `tuple` of values

`Connection` Class Summary

Name	Description
`commit()`	Saves changes to (but does not close) SQLite file
`close()`	Closes (but does not save changes to) SQLite file
`execute(sql)`	Runs the given SQL command on the database and returns a `Cursor` object
`execute(sql, values_tuple)`	Runs the given SQL command (first argument) after substituting question marks with the corresponding values in the given tuple (second argument) and returns a `Cursor` object.
`rollback()`	Undoes any changes made since the last call to `commit()`.
`row_factory`	Can be set to Row so `fetchone()` returns a `dict` mapping column names to field values instead of returning a `tuple` of values

`Cursor` Class Summary

Name	Description
`fetchone()`	Returns a `tuple` of values from next row of the query result or `None` is there are no more values **Note:** This function returns a `dict` mapping column names to field values instead if `row_factory` is set to `Row`
`fetchall()`	Calls `fetchone()` repeatedly until it returns

	None and returns a list of the non-None results

Annex Alternative ways to execute SQL in Python

Create a cursor

```
1   import sqlite3
2
3   connection = sqlite3.connect("school.db")
4   c = connection.cursor()
5   c.execute("CREATE TABLE student " +
6    "(ExamNo INTEGER PRIMARY KEY," +  " Name TEXT,
7   Class TEXT);")
8   c.execute("INSERT INTO student(ExamNo, Name, Class)"
9   +  "VALUES(2150603, 'Peter Rabbit', '21S56');")
10  c.execute("SELECT * FROM student")
11  print(c.fetchone())
12
13  connection.commit()
    connection.close()
```

Using with connection – connection can be used in a with block to auto commit transactions

```
1   import sqlite3
2
3   connection = sqlite3.connect("school.db")
4
5   with connection:
6    connection.execute("CREATE TABLE student " +
7   "(ExamNo INTEGER PRIMARY KEY," +  " Name TEXT,
8   Class TEXT)")
9       connection.execute("INSERT INTO student" +
10       "(ExamNo, Name, Class)" +
11   "VALUES(2150603, 'Peter Rabbit', '21S56')")
12
13  connection.close()
```

Resources:

https://www.tutorialspoint.com/sqlite/sqlite_python.htm

https://stackoverflow.com/questions/10660411/difference-between-cursor-and connection-objects

https://docs.python.org/3/library/sqlite3.html