SQL Databases

Name: _____ ( ) Class: _____ Date: _____

C5c: Using SQLite in Python

Instructional Objectives:

By the end of this task, you should be able to:

- Use a programming language to work with SQL databases
- Use `sqlite3.connect()` to open or create a SQLite file
- Use `sqlite3.Connection.execute()` to run SQL
- Use `sqlite3.Cursor.fetchone()` and `sqlite3.Cursor.fetchall()` to retrieve database rows
- Set `sqlite3.Connection.row_factory` to `sqlite3.Row` in order to simplify the reading of values from retrieved database rows
- Use `sqlite3.Connection.commit()` to save changes and `sqlite3.Connection.close()` to close SQLite files

Python and SQLite

In the previous lessons, you used *DB Browser for SQLite* to create SQLite databases and run SQL queries. This is because DB Browser's graphical user interface makes it easy to experiment with SQL and examine the results.

However, DB Browser is not an appropriate program to use if we want to customise or restrict how the contents of a database are modified or presented. Suppose we have a SQLite database that stores information about the books in a library and we want to let users search the database. We should not use DB Browser for this purpose as malicious users can also use DB Browser to run harmful SQL (e.g., `DROP TABLE`). The interface of DB Browser may also be confusing to users who are not familiar with databases or SQLite.

Instead, developers typically write custom programs to control how users interact with a database. The program may let the user complete a form or choose from a menu to describe what he/she wants to do. Based on the user's input, the program would then generate the appropriate SQL and run it to produce the intended result. This prevents users from modifying the database in ways that are unexpected to the developer.

SQL Databases

In this lesson, we will learn how to write Python programs that can interact with SQLite databases using the built-in `sqlite3` module.

1 A public library uses an SQLite database to store information about its books and the year when each book was published. The library wishes to let users specify a year and query for the titles of all books published before that year.

Which of the following is NOT a valid reason why DB Browser should be avoided for this purpose?

A Users may use DB Browser to insert fake data into the database.

B Users may not know how to perform the query using DB Browser.

C Users may use DB Browser to perform a query that returns nothing.

D Users may use DB Browser to drop tables from the database.

Loading an SQLite Database

To open or create a SQLite database, import `sqlite3` and call `sqlite3.connect()`. This function accepts a `str` argument that contains the path and filename of an SQLite database file and returns a connection object. If no path is included, the SQLite file is assumed to be in the same directory as the Python file. Furthermore, if the specified file does not exist, an empty file will be created with the given filename instead.

After all operations with the database are complete, the `close()` method of the connection object should then be called. This ensures that the database file is closed properly but does not save any modifications that have been made to the data.

For example, the following Python program tries to load an SQLite database named `example.db` in the same directory as the program. If such a file does not exist, an empty file named `example.db` will be created instead:

Program 1: `load_example.py`

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

SQL Databases

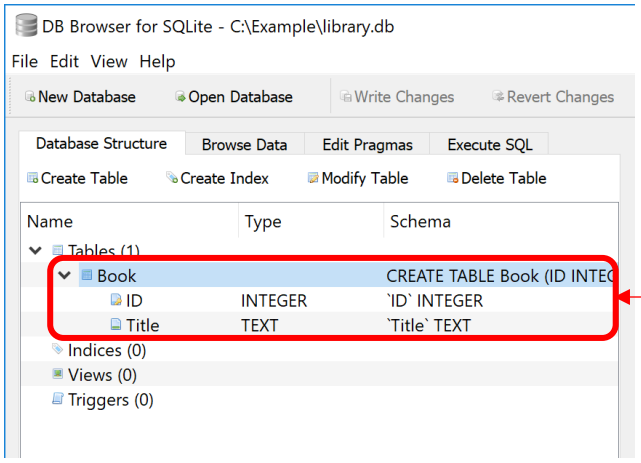
Executing SQL in Python

After loading an SQLite file and getting a connection, we can execute SQL by calling the connection object's `execute()` method with a `str` containing the SQL we wish to run. If any data is modified, we can also save our changes by calling the connection object's `commit()` method. For instance, the following Python program creates a new table named `Book` in a new SQLite file named `library.db` (remove `library.db` from the folder first if it is present):

Program 2: `create_example.py`

```
1  import sqlite3
2
3  connection = sqlite3.connect("library.db")
4  connection.execute("CREATE TABLE Book " +
5                     "(ID INTEGER PRIMARY KEY, Title TEXT)")
6  connection.commit()
7  connection.close()
```

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



new table is created

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

```
Traceback (most recent call last):
  File "create_example.py", line 5, in <module>
    "(ID INTEGER PRIMARY KEY, Title TEXT)")
sqlite3.OperationalError: table Book already exists
```

This demonstrates that calling `execute()` is just like running regular SQL commands in the "Execute SQL" tab of DB Browser. Any errors caused by running the SQL (such as the error telling us the `Book` table already exists) are reported as Python exceptions and can be handled as usual.

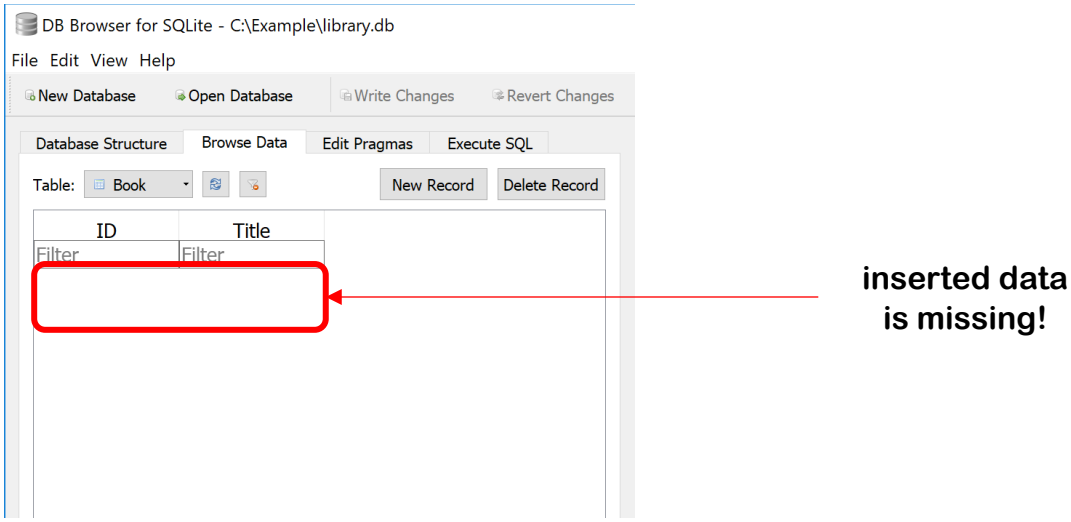
Committing Changes and Rolling Back

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

Program 3: `insert_example_incomplete.py`

```
1  import sqlite3
2
3  connection = sqlite3.connect("library.db")
4  connection.execute("INSERT INTO Book(ID, Title) " +
5                     "VALUES(0, 'Example Book')")
6  connection.close()
```

This program runs with no errors. However, if we open `library.db` using DB Browser, we find that the inserted data is missing:



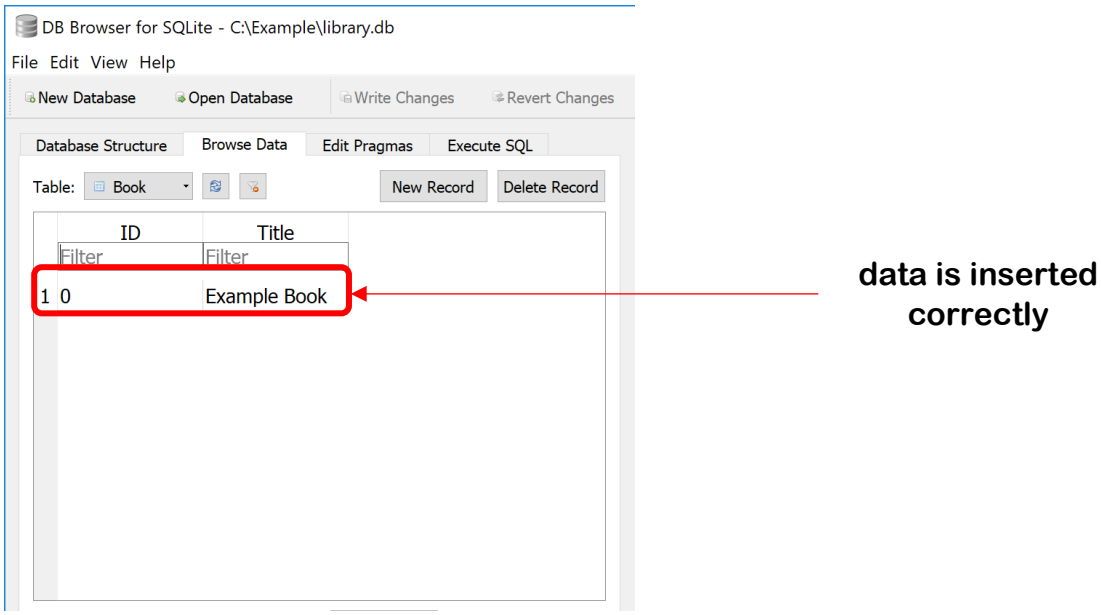
What happened to the data? It was discarded because we did not call `commit()` on the connection object. Using `INSERT`, `UPDATE` or `DELETE` with the `sqlite3` module implicitly opens a *transaction* such that modifications to the data are not saved until the `commit()` method is called:

Program 4: `insert_example.py`

```
1  import sqlite3
2
3  connection = sqlite3.connect("library.db")
4  connection.execute("INSERT INTO Book(ID, Title) " +
5                     "VALUES(0, 'Example Book')")
6  connection.commit()
7  connection.close()
```

SQL Databases

With a call to `commit()` added on line 6, the data is inserted and saved correctly:



This behaviour is useful as sometimes we may wish to discard any modifications to the database's data since the last transaction was opened. For instance, in our library example we may start the process of placing a book on loan but discover partway that the borrower has already reached his/her limit of borrowed books. We can discard all the changes made since the transaction was opened by calling the connection object's `rollback()` method.

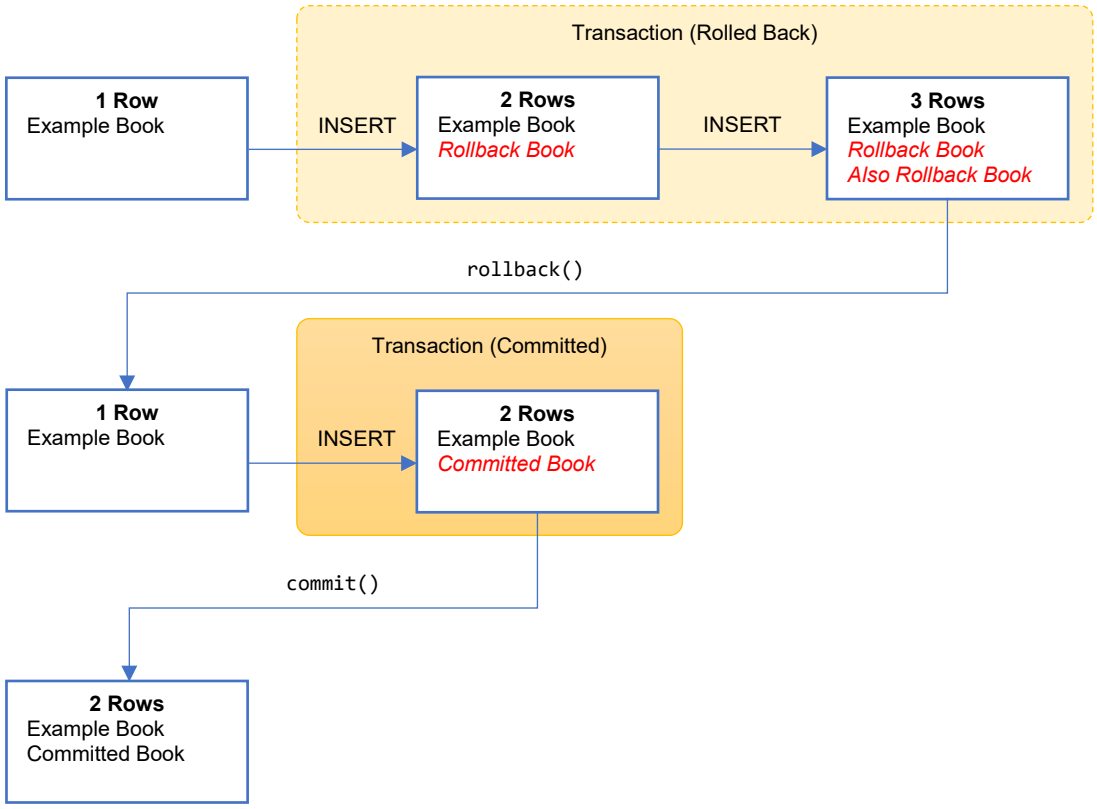
The following example demonstrates how `rollback()` works. In this example, the first two `INSERT` statements are rolled back so they have no effect on the database. On the other hand, the last `INSERT` statement is committed so it *does* affect the database:

Program 5: `rollback_example.py`

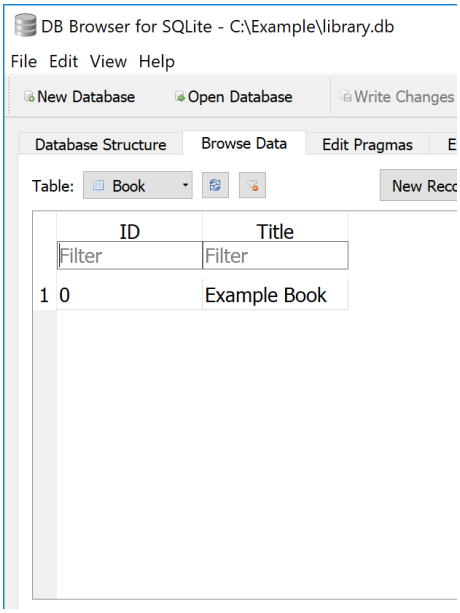
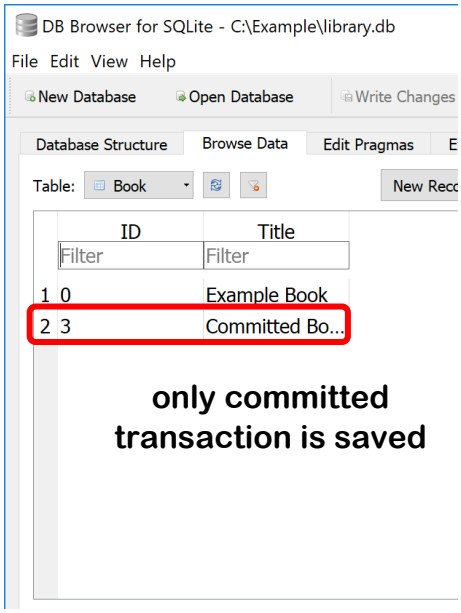
```
1   import sqlite3
2
3   connection = sqlite3.connect("library.db")
4
5   connection.execute("INSERT INTO Book(ID, Title) " +
6                      "VALUES(1, 'Rollback Book')")
7   connection.execute("INSERT INTO Book(ID, Title) " +
8                      "VALUES(2, 'Also Rollback Book')")
9   connection.rollback()
10
11  connection.execute("INSERT INTO Book(ID, Title) " +
12                     "VALUES(3, 'Committed Book')")
13  connection.commit()
14
15  connection.close()
```

SQL Databases

This process is illustrated by the following diagram:



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

Before	After
	

Warning: Starting with Python 3.6, commands that control the database's structure such as `CREATE TABLE` and `DROP TABLE` do not open a transaction and will generally take effect immediately. This means that, by default, it is not possible to roll back such changes automatically.

Enclosing User Input in SQL Safely

When generating SQL commands, we often need to include some data that is provided by the user. For instance, we may want the user to enter the ID of a book to delete from the database. This requires us to generate a `DELETE` command with the entered ID in its `WHERE` clause.

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.

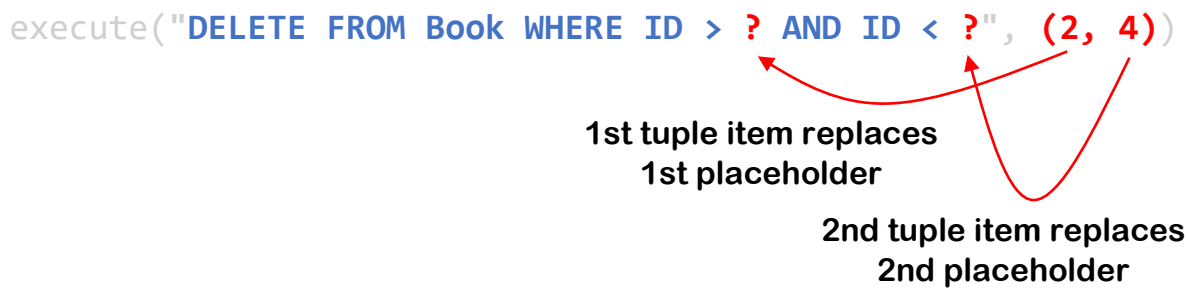
Instead, we should use *parameter substitution* to safely include data that is provided by the user. To do this, we use the question-mark character `?` as a placeholder in the SQL for any data that is provided by the user. We then provide a second argument to `execute()` that is a `tuple` of values that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

For example, the following program safely asks for the ID of a book and deletes the corresponding row from the database:

Program 6: `delete_example.py`

```
1   import sqlite3
2
3   connection = sqlite3.connect("library.db")
4
5   # Insert some rows first so we have something to delete
6   connection.execute("INSERT INTO Book(ID, Title) " +
7                      "VALUES(4, 'Extra Book')")
8   connection.execute("INSERT INTO Book(ID, Title) " +
9                      "VALUES(5, 'Also Extra Book')")
10  connection.commit()
11
12  # Ask for ID and delete the corresponding row
13  book_id = input("Enter Book ID to delete: ")
14  connection.execute("DELETE FROM Book WHERE ID = ?", (book_id,))
15  connection.commit()
16
17  connection.close()
```

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



2 The following is an unsafe program that asks for the ID of a book and deletes the corresponding row from the database:

```
import sqlite3

connection = sqlite3.connect("library.db")
book_id = input("Enter Book ID to delete: ")
connection.execute("DELETE FROM Book WHERE ID = " + book_id)
connection.commit()
connection.close()
```

Can you suggest an input for `book_id` that will delete all the rows in `Book`?

3 You want to write a program that lets a user update the title of a book. To do this, you ask the user for two values: the ID of the book to update (in variable `book_id`) and the new title of the book (in variable `title`). The connection to the SQLite database is stored in variable `connection`.

Which of the following lines of code safely and correctly executes the required SQL to update the book's title?

A
```
connection.execute("UPDATE Book SET Title=? WHERE ID=?",
                   {book_id: title})
```

B
```
connection.execute("UPDATE Book SET Title=? WHERE ID=?",
                   (book_id, title))
```

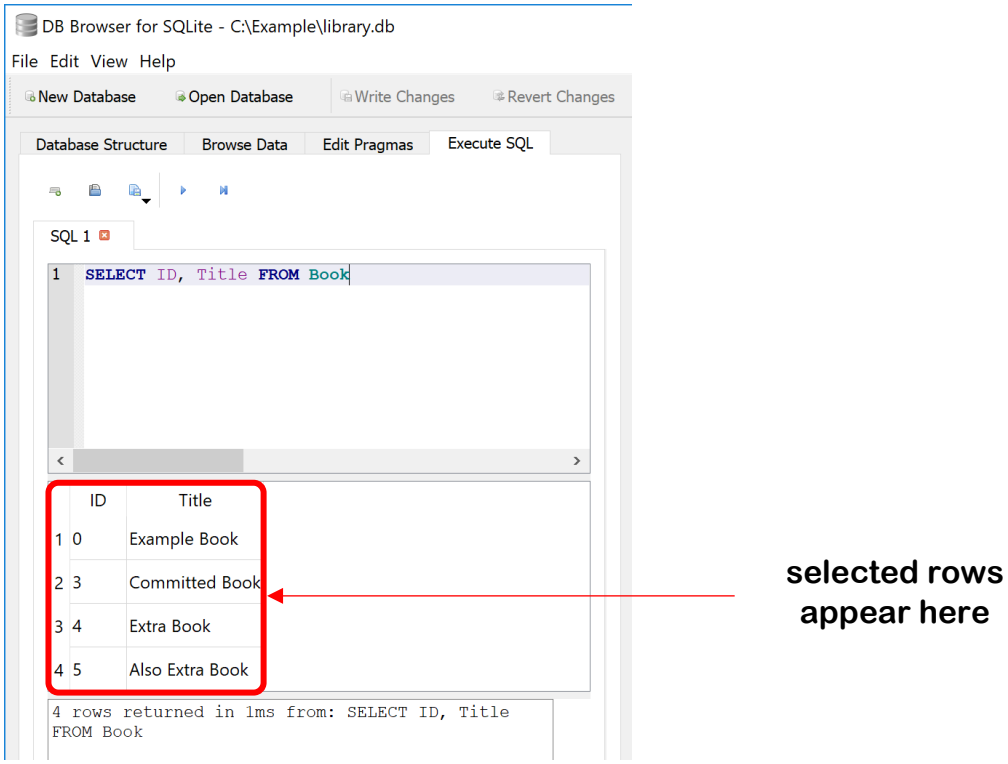
C
```
connection.execute("UPDATE Book SET Title=? WHERE ID=?",
                   title, book_id)
```

D
```
connection.execute("UPDATE Book SET Title=? WHERE ID=?",
                   (title, book_id))
```

SQL Databases

Retrieving Data from Queries

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 that was executed:



In Python, however, you must access the selected rows using a *cursor object* that is returned by the `execute()` method. This cursor object can go through the selected rows, one by one, using either a `for-in` loop or the `fetchone()` method. Each iteration returns a `tuple` of the columns in the current row.

For instance, using the same library example, the following program will print out all the book titles in the `Book` table:

Program 7: `forloop_example.py`

```
1  import sqlite3
2
3  connection = sqlite3.connect("library.db")
4  cursor = connection.execute("SELECT ID, Title FROM Book")
5  for row in cursor:
6      print(row[1])    # Title is second item in the tuple
7  connection.close()
```

For the `fetchone()` method, each call to `fetchone()` will advance the cursor to the next row, so calling `fetchone()` repeatedly will iterate through the selected rows until

the cursor reaches the end and returns None. Hence, the previous program can also be written like this:

Program 8: fetchone_example.py

```
1  import sqlite3
2
3  connection = sqlite3.connect("library.db")
4  cursor = connection.execute("SELECT ID, Title FROM Book")
5  row = cursor.fetchone()
6  while row is not None:
7      print(row[1])    # Title is second item in the tuple
8      row = cursor.fetchone()
9  connection.close()
```

Alternatively, instead of going through the rows one by one, we may wish to fetch all the rows at once and keep them in a list. We can do this calling the `fetchall()` method instead. This method returns a `list` of `tuples` with each `tuple` containing the selected columns for a single row.

For instance, the following program demonstrates yet another way to print out all the book titles in the Book table using `fetchall()`:

Program 9: fetchall_example.py

```
1  import sqlite3
2
3  connection = sqlite3.connect("library.db")
4  cursor = connection.execute("SELECT ID, Title FROM Book")
5  rows = cursor.fetchall()
6  for row in rows:
7      print(row[1])    # Title is second item in the tuple
8  connection.close()
```

In the three previous examples, each row was retrieved as a `tuple` of values based on the order in which the columns were selected. This explains why `row[1]` is used to get the Title as the Title column is the *second* column listed in the `SELECT` statement.

```
cursor = conn.execute("SELECT ID, Title, ID+1 FROM Book")

row = cursor.fetchone()  # (0, "Example Book", 1)
```

Alternatively, we can configure the SQLite connection so that each row is retrieved as a `dict` mapping column names to values instead. To do this, we set the connection

object's `row_factory` attribute to the built-in `sqlite3.Row` class. This lets us change the ordering of columns in our `SELECT` statements without having to modify the code for extracting individual column values.

For example, the following program prints out all the book titles in the `Book` table by retrieving each row as a `dict` of values:

Program 10: `row_factory_example.py`

```
1  import sqlite3
2
3  connection = sqlite3.connect("library.db")
4  connection.row_factory = sqlite3.Row
5  cursor = connection.execute("SELECT ID, Title FROM Book")
6  for row in cursor:
7      print(row["Title"])    # row is now a dictionary
8  connection.close()
```

```
conn.row_factory = sqlite3.Row
cursor = conn.execute("SELECT ID, Title, ID+1 FROM Book")

row = cursor.fetchone()  # { "ID": 0,
                         #   "Title": "Example Book",
                         #   "ID+1": 1 }

print(row["ID"], row["Title"], row["ID+1"])
```

4 Modify Program 9 such that it prints out all the IDs in the `Book` table instead.

5 The SQL on line 5 of Program 10 is replaced with one of the following options. Which option would cause an error on line 7 when the program is run?

A `SELECT * FROM Book`

B `SELECT ID FROM Book`

C `SELECT Title FROM Book`

D `SELECT Title, ID FROM Book`

SQL Databases

6 The following program is supposed to let the user insert data into the Book table. Complete the code on line 12 of the program to perform the insertion.

```
1   import sqlite3
2
3   connection = sqlite3.connect("library.db")
4   while True:
5       try:
6           book_id = int(input("Enter Book ID: "))
7       except ValueError:
8           print("Not a valid ID")
9           continue
10      title = input("Enter Title: ")
11      try:
12          ____________________________________________________
13          connection.commit()
14      except sqlite3.DatabaseError:
15          print("Database error (e.g., duplicate ID)")
16          continue
17      print("Insertion successful!")
18      if input("Quit (Y/N)? ").upper() == "Y":
19          break
20  connection.close()
```

7 Which of the following programs correctly prints out all titles in the Book table in alphabetical order?

A

```
import sqlite3

connection = sqlite3.connect("library.db")
connection.row_factory = sqlite3.Row
cursor = connection.execute("SELECT Title FROM Book " +
                            "WHERE ID=?", (0,))
for row in cursor:
    print(row["Title"])
connection.close()
```

B

```
import sqlite3

connection = sqlite3.connect("library.db")
connection.row_factory = sqlite3.Row
cursor = connection.execute("SELECT Title FROM Book " +
                            "ORDER BY ID")
for row in cursor:
    print(row["Title"])
connection.close()
```

C

```
import sqlite3

connection = sqlite3.connect("library.db")
cursor = connection.execute("SELECT Title FROM Book " +
                            "ORDER BY Title")
for row in cursor:
    print(row["Title"])
connection.close()
```

D

```
import sqlite3

connection = sqlite3.connect("library.db")
cursor = connection.execute("SELECT Title FROM Book " +
                            "ORDER BY Title")
for row in cursor:
    print(row[0])
connection.close()
```

8 Create a new SQLite file using DB Browser named `loans.db` and create three tables using the following SQL:

```
CREATE TABLE Borrower (
    ID INTEGER PRIMARY KEY,
    Name TEXT NOT NULL
);
CREATE TABLE Book (
    ID INTEGER PRIMARY KEY,
    Title TEXT NOT NULL
);
CREATE TABLE Loan (
    ID INTEGER PRIMARY KEY AUTOINCREMENT,
    BorrowerID INTEGER NOT NULL,
    BookID INTEGER NOT NULL,
    FOREIGN KEY (BorrowerID) REFERENCES Borrower(ID),
    FOREIGN KEY (BookID) REFERENCES Book(ID)
);
```

You are to write a Python program to insert loans into this database. The program should do the following for each new loan:

- Ask for the borrower's ID.
- If a borrower with that ID does not exist in the database, ask for the borrower's name and insert it into the `Borrower` table before proceeding.
- Ask for the book's ID.
- If a book with that ID does not exist in the database, ask for the book's title and insert it into the `Book` table before proceeding.

- If the book is already on loan, print an error. Otherwise, insert a new row into the `Loan` table to record that the borrower has loaned the book.

Some parts of the program has been written for you. Complete the rest of the program. (You do not need to perform input validation.)

```
import sqlite3

while True:
    conn = sqlite3.connect("loans.db")

    # Get Borrower ID
    borrower_id = int(input("Enter Borrower ID: "))
    cursor = conn.execute("SELECT COUNT(*) FROM Borrower " +
                          "WHERE ID = ?", (borrower_id,))
    if cursor.fetchone()[0] == 0:
        name = input("Enter Borrower Name: ")
        conn.execute("INSERT INTO Borrower(ID, Name) " +
                     "VALUES(?, ?)", (borrower_id, name))


    # ...fill in the rest of the code below...

    conn.close()
    if input("Quit (Y/N)? ").upper() == "Y":
        break
```

SQL Databases

`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