Connecting to SQL Databases

Introduction

Now that you've gotten a brief introduction to SQL, its time to get some hands-on practice connecting to a database and executing some queries.

Objectives

You will be able to:

- Connect to a SQLite database through the `sqlite3` command-line interface
- Connect to a SQLite database through the `sqlite3` Python library
- View information about database tables and column names
- Retrieve all information from a SQL table

SQLite Interfaces

In this lesson, we will go over a five-step process for retrieving information from a SQLite database:

1. Connecting to the database
2. Viewing the list of tables
3. Selecting all data from the `offices` table
4. Viewing the column names for the `offices` table
5. Disconnecting from the database

This exact same process is achievable using the `sqlite3` terminal interface (in your local terminal, or in a Jupyter Notebook using magic commands), the `sqlite3` Python library alone, or the `sqlite3 + pandas` Python libraries.

In general the `sqlite3 + pandas` library approach is the most succinct and readable, but you might encounter situations where you don't have that particular tooling available, so it's useful to have a basic understanding of all of the various interfaces. If you are coming back to this lesson as reference material, skip down to the `sqlite3 + pandas` section.

Connecting to a Database Using the Terminal

SQLite databases are stored as files on disk. The one we will be using in this lesson is called `data.sqlite`.

One way to interact with a SQLite database is directly in the terminal. If you have the correct `conda` environment installed, you should be able to execute this line of code in the terminal to see the path to the `sqlite3` executable program:

```
which sqlite3
```

The usage of this executable program is

```
sqlite3 name_of_database
```

For example,

```
sqlite3 data.sqlite
```

When you run that code, it **opens up a new terminal inside of your terminal**. This is a SQLite terminal, where any commands you enter will be interpreted as SQLite commands, not regular terminal commands.

SQLite Terminal Code-Along

Most of the time in this course, you will use Python to interact with SQL databases. However it is useful to understand the basics of how to work directly with a SQL terminal. To follow along with this code-along section, **clone this repository locally and run the following commands in the terminal**, not in a Jupyter Notebook. If you can't get it working locally, don't worry too much, just make sure you read through the examples to get a basic understanding.

1. Connect to the database

Run this code from the root of this repository:

```
sqlite3 data.sqlite
```

You should see a print-out with information about the version of SQLite you are using, similar to this:

```
SQLite version 3.35.4 2021-04-02 15:20:15
Enter ".help" for usage hints.
```

You should also see a new command prompt that looks like this:

```
sqlite>
```

2. View the list of tables

Any given database can contain multiple tables. Use the `.tables` SQLite command (then hit Enter) to view the list of tables:

```
sqlite> .tables
```

You should see a print-out with information about the tables in this database, similar to this:

```
customers     offices       orders        productlines
employees     orderdetails  payments      products
```

Note: `.tables` is a SQLite command, *not a SQL query*. This command will only work if you are in the special SQLite terminal, not if you are connecting to the database through Python.

3. Select all data from the `offices` table

Now that we know what tables are available, let's write our first SQL query! We will use `SELECT *` to retrieve all columns from this table, and `FROM offices` to specify which table. Because we are not adding any additional filters, we will automatically get all of the table rows.

Finally, we'll terminate the query with a `;` (semicolon) to indicate that we are finished writing the query. If you forget the semicolon and hit enter, SQLite will assume that you are not finished writing the query and will keep waiting for more instructions. You can type just the semicolon on a row by itself to complete the query.

```
sqlite> SELECT * FROM offices;
```

You should see a print-out of all of the data, with rows separated by newlines and columns separated by `|` (pipe) characters:

```
1|San Francisco|+1 650 219 4782|100 Market Street|Suite 300|CA|USA|94080
|NA
2|Boston|+1 215 837 0825|1550 Court Place|Suite 102|MA|USA|02107|NA
3|NYC|+1 212 555 3000|523 East 53rd Street|apt. 5A|NY|USA|10022|NA
4|Paris|+33 14 723 4404|43 Rue Jouffroy D'abbans|||France|75017|EMEA
5|Tokyo|+81 33 224 5000|4-1 Kioicho||Chiyoda-Ku|Japan|102-8578|Japan
6|Sydney|+61 2 9264 2451|5-11 Wentworth Avenue|Floor #2||Australia|NSW 2
010|APAC
7|London|+44 20 7877 2041|25 Old Broad Street|Level 7||UK|EC2N 1HN|EMEA
```

4. View the column names from the `offices` table

Looking at the data above, you might be able to guess what the different columns mean: ID, city, phone number, address, etc. But if you want to be able to write more specific queries, you need to know exactly what they are called in the database. To do this, you can use the `.schema` SQLite command:

```
sqlite> .schema offices
```

You should see a print-out of the table schema, which looks something like this:

```
CREATE TABLE `offices` (`officeCode`, `city`, `phone`, `addressLine1`, `
addressLine2`, `state`, `country`, `postalCode`, `territory`);
```

So, the numbers 1 through 7 are the "officeCode", then we have the "city", "phone", etc. Nothing too surprising but not exactly what we might have guessed.

Now we have all the information we need to report on what is in the `offices` table of the `data.sqlite` database!

5. Exit the SQLite command prompt

When you are finished querying the SQLite database, make sure you quit out of the special terminal. You may have trouble writing future queries if you are still connected to the database this way.

To quit, use the SQLite command `.quit` :

```
sqlite> .quit
```

(If you're stuck in a SQLite command prompt and `.quit` isn't working for some reason, you can also use control-C twice, or control-D once.)

Connecting to a Database Using Jupyter Notebook

Technically, we can execute terminal commands directly from a Jupyter Notebook. For example, we have used `! ls` before to show the files in the current directory:

In [1]:
```
! ls
```

```
CONTRIBUTING.md  data.sqlite  index.ipynb  LICENSE.md  README.md
```

If we really wanted to, we could just use those same SQLite terminal commands directly in a Jupyter Notebook using magic commands.

Note: This is for demonstration purposes only. You don't need to know how to use this interface.

In [2]:
```
%%script sqlite3 data.sqlite --out tables
.tables
.quit
```

In [3]:
```
print(tables)
```

```
customers     offices       orders        productlines
employees     orderdetails  payments      products
```

In [4]:
```
%%script sqlite3 data.sqlite --out offices_data
SELECT * FROM offices;
.quit
```

In [5]:
```
print(offices_data)
```

```
1|San Francisco|+1 650 219 4782|100 Market Street|Suite 300|CA|USA|94080|NA
2|Boston|+1 215 837 0825|1550 Court Place|Suite 102|MA|USA|02107|NA
3|NYC|+1 212 555 3000|523 East 53rd Street|apt. 5A|NY|USA|10022|NA
4|Paris|+33 14 723 4404|43 Rue Jouffroy D'abbans|||France|75017|EMEA
5|Tokyo|+81 33 224 5000|4-1 Kioicho||Chiyoda-Ku|Japan|102-8578|Japan
6|Sydney|+61 2 9264 2451|5-11 Wentworth Avenue|Floor #2||Australia|NSW 2010|APA
C
7|London|+44 20 7877 2041|25 Old Broad Street|Level 7||UK|EC2N 1HN|EMEA
```

In [6]:
```
%%script sqlite3 data.sqlite --out offices_schema
.schema offices
.quit
```

In [7]:
```
print(offices_schema)
```

```
CREATE TABLE `offices` (`officeCode`, `city`, `phone`, `addressLine1`, `address
Line2`, `state`, `country`, `postalCode`, `territory`);
```

In [8]:
```
type(tables), type(offices_data), type(offices_schema)
```

Out[8]:
```
(str, str, str)
```

This works well enough for some basic exploration, but it's tedious to have to connect and quit for every single command, and the results come back as a giant string every time, rather than a more-usable data structure. For a better interface, we'll use the Python `sqlite3` library.

Connecting to a Database Using Python

As noted previously, SQLite databases are stored as files on disk.

In [9]:
```
! ls
```

```
CONTRIBUTING.md LICENSE.md      README.md       data.sqlite     index.ipynb
```

If we try to read from this file without using any additional libraries, we will get a bunch of garbled nonsense, since this file is encoded as bytes and not plain text:

In [10]:
```
with open("data.sqlite", "rb") as f:
    print(f.read(100))
```

```
b'SQLite format 3\x00\x10\x00\x01\x01\x00@  \x00\x00\x00\x10\x00\x00\x008\x00\x
00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x08\x00\x00\x00\x04\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0
0\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x
00\x00\x00\x10\x00.\x05B'
```

Connecting to the Database

Instead, we will use the `sqlite3` module ([documentation here (https://docs.python.org/3/library/sqlite3.html)](https://docs.python.org/3/library/sqlite3.html)). The way that this module works is that we start by opening a *connection* to the database with `sqlite3.connect` :

In [11]:
```
import sqlite3
conn = sqlite3.connect('data.sqlite')
```

Let's continue on and create a *cursor*.

A cursor object is what can actually execute SQL commands. You create it by calling `.cursor()` on the connection.

In [12]:
```
cur = conn.cursor()
```

Viewing the List of Tables

Let's use the cursor to find out what tables are contained in this database. This requires two steps:

1. Executing the query (`.execute()`)
2. Fetching the results (`.fetchone()` , `.fetchmany()` , or `.fetchall()`)

This is because some SQL commands (e.g. deleting data) do not require results to be fetched, just commands to be executed. So the interface only fetches the results if you ask for them.

In [13]:
```
# Execute the query
# (This is a special query for finding the table names. You don't need to memoriz
cur.execute("""SELECT name FROM sqlite_master WHERE type = 'table';""")
# Fetch the result and store it in table_names
table_names = cur.fetchall()
table_names
```

Out[13]:
```
[('orderdetails',),
 ('payments',),
 ('offices',),
 ('customers',),
 ('orders',),
 ('productlines',),
 ('products',),
 ('employees',)]
```

Selecting All Data from the `offices` Table

If we want to get all information about the `offices` table, we might do something like this (`*` means all columns):

```
In [14]: cur.execute("""SELECT * FROM offices;""")
         cur.fetchall()
```

```
Out[14]: [('1',
           'San Francisco',
           '+1 650 219 4782',
           '100 Market Street',
           'Suite 300',
           'CA',
           'USA',
           '94080',
           'NA'),
          ('2',
           'Boston',
           '+1 215 837 0825',
           '1550 Court Place',
           'Suite 102',
           'MA',
           'USA',
           '02107',
           'NA'),
          ('3',
           'NYC',
           '+1 212 555 3000',
           '523 East 53rd Street',
           'apt. 5A',
           'NY',
           'USA',
           '10022',
           'NA'),
          ('4',
           'Paris',
           '+33 14 723 4404',
           "43 Rue Jouffroy D'abbans",
           '',
           '',
           'France',
           '75017',
           'EMEA'),
          ('5',
           'Tokyo',
           '+81 33 224 5000',
           '4-1 Kioicho',
           '',
           'Chiyoda-Ku',
           'Japan',
           '102-8578',
           'Japan'),
          ('6',
           'Sydney',
           '+61 2 9264 2451',
           '5-11 Wentworth Avenue',
           'Floor #2',
           '',
           'Australia',
           'NSW 2010',
           'APAC'),
          ('7',
           'London',
           '+44 20 7877 2041',
           '25 Old Broad Street',
           'Level 7',
           '',
           'UK',
           'EC2N 1HN',
           'EMEA')]
```

Because `.execute()` returns the cursor object, it also possible to combine the previous two lines into one line, like so:

In [15]:
```
cur.execute("""SELECT * FROM offices;""").fetchall()
```

Out[15]:
```
[('1',
  'San Francisco',
  '+1 650 219 4782',
  '100 Market Street',
  'Suite 300',
  'CA',
  'USA',
  '94080',
  'NA'),
 ('2',
  'Boston',
  '+1 215 837 0825',
  '1550 Court Place',
  'Suite 102',
  'MA',
  'USA',
  '02107',
  'NA'),
 ('3',
  'NYC',
  '+1 212 555 3000',
  '523 East 53rd Street',
  'apt. 5A',
  'NY',
  'USA',
  '10022',
  'NA'),
 ('4',
  'Paris',
  '+33 14 723 4404',
  "43 Rue Jouffroy D'abbans",
  '',
  '',
  'France',
  '75017',
  'EMEA'),
 ('5',
  'Tokyo',
  '+81 33 224 5000',
  '4-1 Kioicho',
  '',
  'Chiyoda-Ku',
  'Japan',
  '102-8578',
  'Japan'),
 ('6',
  'Sydney',
  '+61 2 9264 2451',
  '5-11 Wentworth Avenue',
  'Floor #2',
  '',
  'Australia',
  'NSW 2010',
  'APAC'),
 ('7',
  'London',
  '+44 20 7877 2041',
  '25 Old Broad Street',
  'Level 7',
  '',
  'UK',
  'EC2N 1HN',
  'EMEA')]
```

Compared to the SQLite terminal interface, this data is much more usable, since it's a list of tuples rather than one giant string. If you wanted to select the phone number for the first office, for example, that would just be `[0][2]` tacked on the end of the previous Python statement.

Viewing the Column Names from the `offices` Table

Information about the column names can be retrieved from the cursor. Since the most recent query was `SELECT * FROM offices;` , the cursor will contain information about the `offices` table:

In [16]:
```
cur.description
```

Out[16]:
```
(('officeCode', None, None, None, None, None, None),
 ('city', None, None, None, None, None, None),
 ('phone', None, None, None, None, None, None),
 ('addressLine1', None, None, None, None, None, None),
 ('addressLine2', None, None, None, None, None, None),
 ('state', None, None, None, None, None, None),
 ('country', None, None, None, None, None, None),
 ('postalCode', None, None, None, None, None, None),
 ('territory', None, None, None, None, None, None))
```

If we wanted to combine the previous two steps together to make a dataframe with the right column names, that would look like this:

In [17]:
```
import pandas as pd

pd.DataFrame(
    data=cur.execute("""SELECT * FROM offices;""").fetchall(),
    columns=[x[0] for x in cur.description]
)
```

Out[17]:

	officeCode	city	phone	addressLine1	addressLine2	state	country	postalCode	territ
0	1	San Francisco	+1 650 219 4782	100 Market Street	Suite 300	CA	USA	94080	
1	2	Boston	+1 215 837 0825	1550 Court Place	Suite 102	MA	USA	02107	
2	3	NYC	+1 212 555 3000	523 East 53rd Street	apt. 5A	NY	USA	10022	
3	4	Paris	+33 14 723 4404	43 Rue Jouffroy D'abbans			France	75017	EM
4	5	Tokyo	+81 33 224 5000	4-1 Kioicho		Chiyoda-Ku	Japan	102-8578	Ja
5	6	Sydney	+61 2 9264 2451	5-11 Wentworth Avenue	Floor #2		Australia	NSW 2010	AF
6	7	London	+44 20 7877 2041	25 Old Broad Street	Level 7		UK	EC2N 1HN	EM

Disconnecting from the Database

Now that we have all of the information we need, we can close the connection to the database:

In [18]:
```
conn.close()
```

Connecting to a Database Using `sqlite3` + `pandas`

This is the most straightforward technique for writing SQL queries within a Python context.

Connecting to the Database

This is the same as the previous process, except you only need the connection, not the cursor.

In [19]:
```
conn = sqlite3.connect("data.sqlite")
```

Viewing the List of Tables

Now that we have the connection, we can use the `pd.read_sql` method (documentation here (https://pandas.pydata.org/docs/reference/api/pandas.read_sql.html)). Instead of using the cursor, all you need is the connection variable:

In [20]:
```
df = pd.read_sql("""SELECT name FROM sqlite_master WHERE type = 'table';""", conr
df
```

Out[20]:

	name
0	orderdetails
1	payments
2	offices
3	customers
4	orders
5	productlines
6	products
7	employees

In [21]:
```
type(df)
```

Out[21]:
```
pandas.core.frame.DataFrame
```

As you can see, this technique created a pandas dataframe as the result of the query, rather than a string or a collection of tuples. This tends to be very convenient for future analysis.

Selecting all Data from the `offices` Table and Viewing the Column Names from the `offices` Table

Here we have combined two steps into one! `pd.read_sql` automatically retrieves the relevant column names when you select data from a table.

In [22]:
```
pd.read_sql("SELECT * FROM offices;", conn)
```

Out[22]:

	officeCode	city	phone	addressLine1	addressLine2	state	country	postalCode	territ
0	1	San Francisco	+1 650 219 4782	100 Market Street	Suite 300	CA	USA	94080	
1	2	Boston	+1 215 837 0825	1550 Court Place	Suite 102	MA	USA	02107	
2	3	NYC	+1 212 555 3000	523 East 53rd Street	apt. 5A	NY	USA	10022	
3	4	Paris	+33 14 723 4404	43 Rue Jouffroy D'abbans			France	75017	EM
4	5	Tokyo	+81 33 224 5000	4-1 Kioicho		Chiyoda-Ku	Japan	102-8578	Ja
5	6	Sydney	+61 2 9264 2451	5-11 Wentworth Avenue	Floor #2		Australia	NSW 2010	AF
6	7	London	+44 20 7877 2041	25 Old Broad Street	Level 7		UK	EC2N 1HN	EM

It is still useful to be aware of the cursor construct in case you ever need to develop Python code that fetches one result at a time, or is a command other than `SELECT`. But in general if you know that the end result is creating a pandas dataframe to display the result, you don't really need to interface with the cursor directly.

Disconnecting from the Database

This is the same as the process for using `sqlite3` without `pandas`.

In [23]:
```
conn.close()
```

Comparing and Contrasting Interfaces

As a professional data scientist, you should be able to learn the fundamental concepts then switch between different interfaces that use the same tooling "under the hood".

For most of our data science program, you can go ahead and use `pd.read_sql` for most purposes, but it's important to have some familiarity with the other interfaces.

Terminal Interface

At some point in your career you will likely be expected to write SQL code directly in a terminal interface. It might be MySQL or PostreSQL rather than SQLite, but the same general approach applies (particularly the importance of including the semicolon at the end of a query). Most databases automatically come with a terminal interface, and you may or may not be able to install tools like Python or Jupyter Notebook on the same machine that hosts the SQL database.

For SQLite in particular, the terminal interface has some helpful tooling. For example, if you are exploring the list of database tables, `.tables` is a much easier command to remember than `SELECT name FROM sqlite_master WHERE type = 'table';` . There are also lots of configuration options you can explore using the `.help` command.

One of the main downsides of using the terminal interface (without additional tooling) is that the result of the query is simply a long string. This can prove challenging if you need to perform further calculations or highlight particular elements of your query results.

Python `sqlite3` Interface without `pandas`

Using Python to interface with SQL means that you can take advantage of your existing Python skills to craft a meaningful narrative with the results of a SQL query. The `sqlite3` library is the simplest way to achieve this.

If you ever need to use Python for tasks other than `SELECT` (e.g. inserting rows into a table), the `sqlite3` library has flexible and powerful options available.

Two main downsides of using `sqlite3` without `pandas` are the more-verbose syntax (i.e. you are writing more code, creating more opportunities to make a silly mistake) as well as the query result format (list of tuples). While a list of tuples is more usable than a long string, it can still be difficult to read, and navigating through the result requires numeric indexes rather than named labels like `pandas` has.

Python `sqlite3` Interface with `pandas`

Adding `pandas` to `sqlite3` means you can automatically display a table of query results in a tidy, visually-appealing way. You simply use `pd.read_sql` and pass in a query string and a `sqlite3` connection object.

There are some tasks where `pandas` is not appropriate (e.g. inserting rows into a table), but in general it's the easiest approach for most tasks in this data science program.

Summary

In this lesson, you saw how to connect to a SQL database via different interfaces and how to subsequently execute queries against that database. Going forward, you'll continue to learn additional keywords for specifying your query parameters!

[illegible]