3 SQL 1: Introduction

Lab Objective: *Being able to store and manipulate large data sets quickly is a fundamental part of data science. The SQL language is the classic database management system for working with tabular data. In this lab we introduce the basics of SQL, including creating, reading, updating, and deleting SQL tables, all via Python's standard SQL interaction modules.*

Relational Databases

A *relational database* is a collection of tables called *relations*. A single row in a table, called a *tuple*, corresponds to an individual instance of data. The columns, called *attributes* or *features*, are data values of a particular category. The collection of column headings is called the *schema* of the table, which describes the kind of information stored in each entry of the tuples.

For example, suppose a database contains demographic information for M individuals. If a table had the schema `(Name, Gender, Age)`, then each row of the table would be a 3-tuple corresponding to a single individual, such as `(Jane Doe, F, 20)` or `(Samuel Clemens, M, 74.4)`. The table would therefore be $M \times 3$ in shape. Note that including a person's age in a database means that the data would quickly be outdated since people get older every year. A better choice would be to use birth year. Another table with the schema `(Name, Income)` would be $M \times 2$ if it included all M individuals.

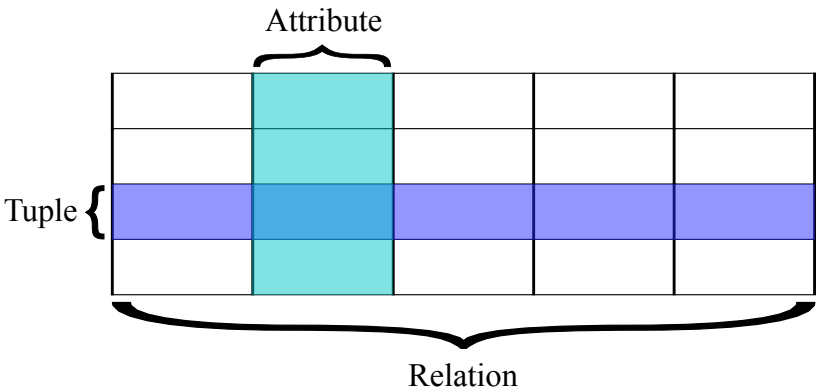

Figure 3.1: See `https://en.wikipedia.org/wiki/Relational_database`.

SQLite

The most common database management systems (DBMS) for relational databases are based on *Structured Query Language*, commonly called *SQL* (pronounced[1] "sequel"). Though SQL is a language in and of itself, most programming languages have tools for executing SQL routines. In Python, the most common variant of SQL is *SQLite*, implemented as the `sqlite3` module in the standard library.

A SQL database is stored in an external file, usually marked with the file extension `db` or `mdf`. These files should **not** be opened in Python with `open()` like text files; instead, any interactions with the database—creating, reading, updating, or deleting data—should occur as follows.

1. Create a connection to the database with `sqlite3.connect()`. This creates a database file if one does not already exist.
2. Get a *cursor*, an object that manages the actual traversal of the database, with the connection's `cursor()` method.
3. Alter or read data with the cursor's `execute()` method, which accepts an actual SQL command as a string.
4. Save any changes with the cursor's `commit()` method, or revert changes with `rollback()`.
5. Close the connection.

```
>>> import sqlite3 as sql

# Establish a connection to a database file or create one if it doesn't exist.
>>> conn = sql.connect("my_database.db")
>>> try:
...     cur = conn.cursor()                          # Get a cursor object.
...     cur.execute("SELECT * FROM MyTable")         # Execute a SQL command.
... except sql.Error:                                # If there is an error,
...     conn.rollback()                              #  revert the changes
...     raise                                        #  and raise the error.
... else:                                            # If there are no errors,
...     conn.commit()                                #  save the changes.
... finally:
...     conn.close()                                 # Close the connection.
```

Achtung!

Some changes, such as creating and deleting tables, are automatically committed to the database as part of the cursor's `execute()` method. Be **extremely cautious** when deleting tables, as the action is immediate and permanent. Most changes, however, do not take effect in the database file until the connection's `commit()` method is called. Be careful not to close the connection before committing desired changes, or those changes will not be recorded.

[1] See `https://english.stackexchange.com/questions/7231/how-is-sql-pronounced` for a brief history of the somewhat controversial pronunciation of SQL.

The `with` statement can be used with `open()` so that file streams are automatically closed, even in the event of an error. Likewise, combining the `with` statement with `sql.connect()` automatically rolls back changes if there is an error and commits them otherwise. However, the actual database connection is **not** closed automatically. With this strategy, the previous code block can be reduced to the following.

```
>>> try:
...     with sql.connect("my_database.db") as conn:
...         cur = conn.cursor()                     # Get the cursor.
...         cur.execute("SELECT * FROM MyTable")    # Execute a SQL command.
... finally:                                        # Commit or revert, then
...     conn.close()                                #  close the connection.
```

Managing Database Tables

SQLite uses five native data types (relatively few compared to other SQL systems) that correspond neatly to native Python data types.

Python Type	SQLite Type
`None`	`NULL`
`int`	`INTEGER`
`float`	`REAL`
`str`	`TEXT`
`bytes`	`BLOB`

The `CREATE TABLE` command, together with a table name and a schema, adds a new table to a database. The schema is a comma-separated list where each entry specifies the column name, the column data type,[2] and other optional parameters. For example, the following code adds a table called `MyTable` with the schema `(Name, ID, Age)` with appropriate data types.

```
>>> with sql.connect("my_database.db") as conn:
...     cur = conn.cursor()
...     cur.execute("CREATE TABLE MyTable (Name TEXT, ID INTEGER, Age REAL)")
...
>>> conn.close()
```

The `DROP TABLE` command deletes a table. However, using `CREATE TABLE` to try to create a table that already exists or using `DROP TABLE` to remove a nonexistent table raises an error. Use `DROP TABLE IF EXISTS` to remove a table without raising an error if the table doesn't exist. See Table 3.1 for more table management commands.

[2]Though SQLite does not force the data in a single column to be of the same type, most other SQL systems enforce uniform column types, so it is good practice to specify data types in the schema.

Operation	SQLite Command
Create a new table	`CREATE TABLE <table> (<schema>);`
Delete a table	`DROP TABLE <table>;`
Delete a table if it exists	`DROP TABLE IF EXISTS <table>;`
Add a new column to a table	`ALTER TABLE <table> ADD <column> <dtype>`
Remove an existing column	`ALTER TABLE <table> DROP COLUMN <column>;`
Rename an existing column	`ALTER TABLE <table> ALTER COLUMN <column> <dtype>;`

Table 3.1: SQLite commands for managing tables and columns.

Note

SQL commands like `CREATE TABLE` are often written in all caps to distinguish them from other parts of the query, like the table name. This is only a matter of style: SQLite, along with most other versions of SQL, is case insensitive. In Python's SQLite interface, the trailing semicolon is also unnecessary. However, most other database systems require it, so it's good practice to include the semicolon in Python.

Problem 1. Write a function that accepts the name of a database file. Connect to the database (and create it if it doesn't exist). Drop the tables `MajorInfo`, `CourseInfo`, `StudentInfo`, and `StudentGrades` from the database **if** they exist. Next, add the following tables to the database with the specified column names and types.

- `MajorInfo`: `MajorID` (integers) and `MajorName` (strings).
- `CourseInfo`: `CourseID` (integers) and `CourseName` (strings).
- `StudentInfo`: `StudentID` (integers), `StudentName` (strings), and `MajorID` (integers).
- `StudentGrades`: `StudentID` (integers), `CourseID` (integers), and `Grade` (strings).

Remember to commit and close the database. You should be able to execute your function more than once with the same input without raising an error.

To check the database, use the following commands to get the column names of a specified table. Assume here that the database file is called `students.db`.

```
>>> with sql.connect("students.db") as conn:
...     cur = conn.cursor()
...     cur.execute("SELECT * FROM StudentInfo;")
...     print([d[0] for d in cur.description])
...
['StudentID', 'StudentName', 'MajorID']
```

Inserting, Removing, and Altering Data

Tuples are added to SQLite database tables with the `INSERT INTO` command.

```
# Add the tuple (Samuel Clemens, 1910421, 74.4) to MyTable in my_database.db.
>>> with sql.connect("my_database.db") as conn:
...     cur = conn.cursor()
...     cur.execute("INSERT INTO MyTable "
...                 "VALUES('Samuel Clemens', 1910421, 74.4);")
```

With this syntax, SQLite assumes that values match sequentially with the schema of the table. The schema of the table can also be written explicitly for clarity.

```
>>> with sql.connect("my_database.db") as conn:
...     cur = conn.cursor()
...     cur.execute("INSERT INTO MyTable(Name, ID, Age) "
...                 "VALUES('Samuel Clemens', 1910421, 74.4);")
```

Achtung!

Never use Python's string operations to construct a SQL query from variables. Doing so makes the program susceptible to a *SQL injection attack*.[a] Instead, use parameter substitution to construct dynamic commands: use a `?` character within the command, then provide the sequence of values as a second argument to `execute()`.

```
>>> with sql.connect("my_database.db") as conn:
...     cur = conn.cursor()
...     values = ('Samuel Clemens', 1910421, 74.4)
...     # Don't piece the command together with string operations!
...     # cur.execute("INSERT INTO MyTable VALUES " + str(values)) # BAD!
...     # Instead, use parameter substitution.
...     cur.execute("INSERT INTO MyTable VALUES(?,?,?);", values)  # Good.
```

[a] See `https://xkcd.com/327/` for an example.

To insert several rows at a time to the same table, use the cursor object's `executemany()` method and parameter substitution with a list of tuples. This is typically much faster than using `execute()` repeatedly.

```
# Insert (Samuel Clemens, 1910421, 74.4) and (Jane Doe, 123, 20) to MyTable.
>>> with sql.connect("my_database.db") as conn:
...     cur = conn.cursor()
...     rows = [('John Smith', 456, 40.5), ('Jane Doe', 123, 20)]
...     cur.executemany("INSERT INTO MyTable VALUES(?,?,?);", rows)
```

Problem 2. Expand your function from Problem 1 so that it populates the tables with the data given in Tables 3.2a–3.2d.

MajorID	MajorName
1	Math
2	Science
3	Writing
4	Art

(a) MajorInfo

CourseID	CourseName
1	Calculus
2	English
3	Pottery
4	History

(b) CourseInfo

StudentID	StudentName	MajorID
401767594	Michelle Fernandez	1
678665086	Gilbert Chapman	NULL
553725811	Roberta Cook	2
886308195	Rene Cross	3
103066521	Cameron Kim	4
821568627	Mercedes Hall	NULL
206208438	Kristopher Tran	2
341324754	Cassandra Holland	1
262019426	Alfonso Phelps	NULL
622665098	Sammy Burke	2

(c) StudentInfo

StudentID	CourseID	Grade
401767594	4	C
401767594	3	B–
678665086	4	A+
678665086	3	A+
553725811	2	C
678665086	1	B
886308195	1	A
103066521	2	C
103066521	3	C–
821568627	4	D
821568627	2	A+
821568627	1	B
206208438	2	A
206208438	1	C+
341324754	2	D–
341324754	1	A–
103066521	4	A
262019426	2	B
262019426	3	C
622665098	1	A
622665098	2	A–

(d) StudentGrades

Table 3.2: Student database.

The `StudentInfo` and `StudentGrades` tables are also recorded in `student_info.csv` and `student_grades.csv`, respectively, with `NULL` values represented as -1 (we'll leave them as -1 for now). A CSV (**c**omma-**s**eparated **v**alues) file can be read like a normal text file or with the `csv` module.

```
>>> import csv
>>> with open("student_info.csv", 'r') as infile:
...     rows = list(csv.reader(infile))
```

To validate your database, use the following command to retrieve the rows from a table.

```
>>> with sql.connect("students.db") as conn:
...     cur = conn.cursor()
...     for row in cur.execute("SELECT * FROM MajorInfo;"):
...         print(row)
(1, 'Math')
(2, 'Science')
(3, 'Writing')
(4, 'Art')
```

Problem 3. The data file `us_earthquakes.csv`[a] contains data from about 3,500 earthquakes in the United States since the 1769. Each row records the year, month, day, hour, minute, second, latitude, longitude, and magnitude of a single earthquake (in that order). Note that latitude, longitude, and magnitude are floats, while the remaining columns are integers.

Write a function that accepts the name of a database file. Drop the table `USEarthquakes` if it already exists, then create a new `USEarthquakes` table with schema `(Year, Month, Day, Hour, Minute, Second, Latitude, Longitude, Magnitude)`. Populate the table with the data from `us_earthquakes.csv`. Remember to commit the changes and close the connection. (Hint: using `executemany()` is much faster than using `execute()` in a loop.)

[a]Retrieved from `https://datarepository.wolframcloud.com/resources/Sample-Data-US-Earthquakes`.

The WHERE Clause

Deleting or altering existing data in a database requires some searching for the desired row or rows. The `WHERE` clause is a *predicate* that filters the rows based on a boolean condition. The operators `==`, `!=`, `<`, `>`, `<=`, `>=`, `AND`, `OR`, and `NOT` all work as expected to create search conditions.

```
>>> with sql.connect("my_database.db") as conn:
...     cur = conn.cursor()
...     # Delete any rows where the Age column has a value less than 30.
...     cur.execute("DELETE FROM MyTable WHERE Age < 30;")
...     # Change the Name of "Samuel Clemens" to "Mark Twain".
...     cur.execute("UPDATE MyTable SET Name='Mark Twain' WHERE ID==1910421;")
```

If the `WHERE` clause were omitted from either of the previous commands, every record in `MyTable` would be affected. **Always** use a very specific `WHERE` clause when removing or updating data.

Operation	SQLite Command
Add a new row to a table	`INSERT INTO table VALUES(<values>);`
Remove rows from a table	`DELETE FROM <table> WHERE <condition>;`
Change values in existing rows	`UPDATE <table> SET <column1>=<value1>, ... WHERE <condition>;`

Table 3.3: SQLite commands for inserting, removing, and updating rows.

Note

SQLite treats `=` and `==` as equivalent operators. For clarity, in this lab we will always use `==` when comparing two values, such as in a `WHERE` clause. The only time we use `=` is in `SET` statements. Be aware that some flavors of SQL (such as MySQL) have no concept of `==`, and typing it in a query will return an error.

Problem 4. Modify your function from Problems 1 and 2 so that in the `StudentInfo` table, values of -1 in the `MajorID` column are replaced with `NULL` values.

Also modify your function from Problem 3 in the following ways.

1. Remove rows from `USEarthquakes` that have a value of 0 for the `Magnitude`.
2. Replace 0 values in the `Day`, `Hour`, `Minute`, and `Second` columns with `NULL` values.

Reading and Analyzing Data

Constructing and managing databases is fundamental, but most time in SQL is spent analyzing existing data. A *query* is a SQL command that reads all or part of a database without actually modifying the data. Queries start with the `SELECT` command, followed by column and table names and additional (optional) conditions. The results of a query, called the *result set*, are accessed through the cursor object. After calling `execute()` with a SQL query, use `fetchall()` or another cursor method from Table 3.4 to get the list of matching tuples.

Method	Description
`execute()`	Execute a single SQL command
`executemany()`	Execute a single SQL command over different values
`executescript()`	Execute a SQL script (multiple SQL commands)
`fetchone()`	Return a single tuple from the result set
`fetchmany(n)`	Return the next n rows from the result set as a list of tuples
`fetchall()`	Return the entire result set as a list of tuples

Table 3.4: Methods of database cursor objects.

```
>>> conn = sql.connect("students.db")
>>> cur = conn.cursor()

# Get tuples of the form (StudentID, StudentName) from the StudentInfo table.
>>> cur.execute("SELECT StudentID, StudentName FROM StudentInfo;")
>>> cur.fetchone()          # List the first match (a tuple).
(401767594, 'Michelle Fernandez')

>>> cur.fetchmany(3)        # List the next three matches (a list of tuples).
[(678665086, 'Gilbert Chapman'),
 (553725811, 'Roberta Cook'),
 (886308195, 'Rene Cross')]

>>> cur.fetchall()          # List the remaining matches.
[(103066521, 'Cameron Kim'),
 (821568627, 'Mercedes Hall'),
 (206208438, 'Kristopher Tran'),
 (341324754, 'Cassandra Holland'),
 (262019426, 'Alfonso Phelps'),
 (622665098, 'Sammy Burke')]
```

```
# Use * in place of column names to get all of the columns.
>>> cur.execute("SELECT * FROM MajorInfo;").fetchall()
[(1, 'Math'), (2, 'Science'), (3, 'Writing'), (4, 'Art')]

>>> conn.close()
```

The `WHERE` predicate can also refine a `SELECT` command. If the condition depends on a column in a different table from the data that is being a selected, create a *table alias* with the `AS` command to specify columns in the form `table.column`.

```
>>> conn = sql.connect("students.db")
>>> cur = conn.cursor()

# Get the names of all math majors.
>>> cur.execute("SELECT SI.StudentName "
...             "FROM StudentInfo AS SI, MajorInfo AS MI "
...             "WHERE SI.MajorID == MI.MajorID AND MI.MajorName == 'Math'")
# The result set is a list of 1-tuples; extract the entry from each tuple.
>>> [t[0] for t in cur.fetchall()]
['Cassandra Holland', 'Michelle Fernandez']

# Get the names and grades of everyone in English class.
>>> cur.execute("SELECT SI.StudentName, SG.Grade "
                "FROM StudentInfo AS SI, StudentGrades AS SG "
                "WHERE SI.StudentID == SG.StudentID AND CourseID == 2;")
>>> cur.fetchall()
[('Roberta Cook', 'C'),
 ('Cameron Kim', 'C'),
 ('Mercedes Hall', 'A+'),
 ('Kristopher Tran', 'A'),
 ('Cassandra Holland', 'D-'),
 ('Alfonso Phelps', 'B'),
 ('Sammy Burke', 'A-')]

>>> conn.close()
```

Problem 5. Write a function that accepts the name of a database file. Assuming the database to be in the format of the one created in Problems 1 and 2, query the database for all tuples of the form (`StudentName`, `CourseName`) where that student has an "A" or "A+" grade in that course. Return the list of tuples.

Aggregate Functions

A result set can be analyzed in Python using tools like NumPy, but SQL itself provides a few tools for computing a few very basic statistics: `AVG()`, `MIN()`, `MAX()`, `SUM()`, and `COUNT()` are *aggregate functions* that compress the columns of a result set into the desired quantity.

```
>>> conn = sql.connect("students.db")
>>> cur = conn.cursor()

# Get the number of students and the lowest ID number in StudentInfo.
>>> cur.execute("SELECT COUNT(StudentName), MIN(StudentID) FROM StudentInfo;")
>>> cur.fetchall()
[(10, 103066521)]
```

Problem 6. Write a function that accepts the name of a database file. Assuming the database to be in the format of the one created in Problem 3, query the `USEarthquakes` table for the following information.

- The magnitudes of the earthquakes during the 19th century (1800–1899).
- The magnitudes of the earthquakes during the 20th century (1900–1999).
- The average magnitude of all earthquakes in the database.

Create a single figure with two subplots: a histogram of the magnitudes of the earthquakes in the 19th century, and a histogram of the magnitudes of the earthquakes in the 20th century. Show the figure, then return the average magnitude of all of the earthquakes in the database. Be sure to return an actual number, not a list or a tuple.
(Hint: use `np.ravel()` to convert a result set of 1-tuples to a 1-D array.)

Note

Problem 6 raises an interesting question: are the number of earthquakes in the United States increasing with time, and if so, how drastically? A closer look shows that only 3 earthquakes were recorded (in this data set) from 1700–1799, 208 from 1800–1899, and a whopping 3049 from 1900–1999. Is the increase in earthquakes due to there actually being more earthquakes, or to the improvement of earthquake detection technology? The best answer without conducting additional research is "probably both." Be careful to question the nature of your data—how it was gathered, what it may be lacking, what biases or lurking variables might be present—before jumping to strong conclusions.

See the following for more info on the `sqlite3` and SQL in general.

- `https://docs.python.org/3/library/sqlite3.html`
- `https://www.w3schools.com/sql/`
- `https://en.wikipedia.org/wiki/SQL_injection`

Additional Material

Shortcuts for WHERE Conditions

Complicated `WHERE` conditions can be simplified with the following commands.

- `IN`: check for equality to one of several values quickly, similar to Python's `in` operator. In other words, the following SQL commands are equivalent.

```
SELECT * FROM StudentInfo WHERE MajorID == 1 OR MajorID == 2;
SELECT * FROM StudentInfo WHERE MajorID IN (1,2);
```

- `BETWEEN`: check two (inclusive) inequalities quickly. The following are equivalent.

```
SELECT * FROM MyTable WHERE AGE >= 20 AND AGE <= 60;
SELECT * FROM MyTable WHERE AGE BETWEEN 20 AND 60;
```