CSC120H Lab 9

1 Objectives

- Practise querying a SQL database
- Practise creating and manipulating tables in a SQL database

2 Querying SQL databases

A **SQL database** is a collection of tables of data, where each table contains columns that indicate what the data represents, and rows which contain records of data for those columns.
For example: the following is a table of flight information:

Flights

airline	flight_number	origin	destination	minutes
AS	98	ANC	SEA	194
AA	2336	LAX	PBI	279
US	840	SFO	CLT	293
AA	258	LAX	MIA	281
AS	135	SEA	ANC	215
DL	806	SFO	MSP	230
NK	612	LAS	MSP	170
US	2013	LAX	CLT	249
AA	1112	SFO	DFW	193
DL	1173	LAS	ATL	203
DL	2336	DEN	ATL	149
AA	1674	LAS	MIA	266
DL	1434	LAX	MSP	210
DL	2324	SLC	ATL	199

Each row (or 'record') in the table corresponds to a flight. The columns give us information about each flight record, including its airline, flight number, origin airport, destination airport, and total flight time in minutes.
From the Labs page, download the file `'flight_data.db'`. This file is a **SQL database file**, which contains data about flights. Right now, it only contains the **flight** table above.

Running SQL Queries review
We run **SQL queries** so that we can obtain data from tables in a database. We send a query command to the database, and the database gives us back all of the rows that match that query. SQL queries can only be interpreted by the database software, SQLite3, and are not alone interpretable by Python, which is why we have to `import sqlite3`.

The SQL language includes a `SELECT` statement, which indicates that we want to get (or 'select') certain rows from the database. A `SELECT` statement always requires at the minimum two parameters:
1. The **table** in the database we want to select data from.
2. The names of the **columns** in that table that we want to select, or a star character (`*`) if we want to select all of the columns.

The general format for selecting columns from a database table is:

```
SELECT <columns> FROM <table>
```

For example, if we wanted to select all columns from our flights table, we would write:

```
SELECT * FROM flights
```

This will select all of the columns from flights. But which rows does it select? By default, all rows in the table will be selected (we will later specify how to select rows more specifically).

What if we wanted to select only the airline and flight number columns from our flights table? We would write:

```
SELECT airline, flight_number FROM flights
```

Here we specified which columns we want, and we will only get those columns from the table. By default we will still get all of the rows.

Let's now run these queries in Python to actually get some data back from our database. Download the `lab9_functions.py` and `lab9_main.py` files.
In these files, we use an `import` statement in `lab9_main.py` to bring in all of the functions that you will **write** in `lab9_functions.py`. You will **call** your functions inside of the `if __name__ == '__main__'` block of `lab9_main.py`.

Look inside the `lab9_functions.py` file. The first function defined there is a helper function, `run_query()`. Inside this function, you will notice the code we've looked at in lecture for connecting, getting a cursor, executing a query through the cursor, fetching the data from the cursor, and closing the connection. With this function, you will not have to re-write these steps, which are required for every query.
Notice that the parameter `args` in that function has a **default value** of `None`. This means we don't have to supply an argument for `args` if we don't need it, which will be the case for our first function.

Query 1: Let's start by completing the function `get_all_flights()` in `lab9_functions.py`, which should query the database for all flights records and columns from the flights table.
Replace the query string with the correct query string for this query. In this function, `run_query()` is called for you and the result is returned.
Now, go to your `lab9_main.py`, and uncomment the call to `print_records()` for Query 1. Run the file and check the shell to make sure you get what you expect. If you're unsure, ask your TA.

Task: Complete the functions for queries 2-5. Uncomment out the correct place in the main block of `lab9_main.py` and run the file for each function you complete.

Selecting specific rows: Conditional statements
In the previous examples, we were specifying which columns we want to select. Let's now add to our queries to also limit which rows we should select.

We specify which rows we want by indicating a **condition** on the values for a particular column. For example, if we want still want all of the columns, but also wanted to limit the the values for the the `minutes` column to be more than 200 minutes, we would use the `WHERE` clause like so:

```
SELECT * FROM flights WHERE minutes > 200
```

In addition to the greater than and less than comparisons, we can also do equality, if we wanted to get get the records where the origin airport is `LAX`, we would write:

```
SELECT * FROM flights WHERE origin = 'LAX'
```

Notice that `LAX` has quotes around it. These are necessary for strings in SQL, and might cause issues if you accidentally forget to leave them out.
Instead, we usually do something called **string formatting**, which lets us define places in the string where we want to place values, and then separately define those values to be replaced later. For example, we can put a questions mark in place of where `LAX` would be:
`query = 'SELECT * FROM flights WHERE origin = ?'`
and then add an extra tuple argument to `run_query()`:
`run_query(query, ('LAX',))` (remember the comma for one-element tuples).

This will replace the question mark with the appropriate value, and will format it the correct way automatically.
You can also put multiple question marks, and then have a tuple with multiple elements. The question marks will be replaced left to right by the values of the tuples in order.
Here's an example of using the `AND` operator to check for two different conditions at the same time:

```
>>> query = 'SELECT * FROM flights WHERE origin = ?  AND minutes > ?'
>>> run_query('flight_data.db', query, ('LAX', 200))
```

This returns all rows that have LAX as the origin and took longer than 200 minutes.

Task: Complete the functions for Queries 6 to 10. Make sure you also print them properly in the main block in the other file.

3 Manipulating Tables

So far, we've been querying the database to select certain data from the tables. What if we want to actually edit the database data ourselves?
1. Creating a new table
After the query functions for section 2 above, there is a function called `create_flights_table()` This function was the function used to create the flights table in the databse from a `flights.csv` file.
Notice that it uses the `CREATE TABLE` command, followed by the column names and their types (`airline` is TEXT, while `flight_number` is a NUMBER).
Then, after reading each line of the csv file into a list of strings, we pick off each element, and then run an `INSERT` statement that inserts a new record into the flights table. Notice how we use string formatting here as well, with 5 question marks and 5 variables.

Task:
Download the `airlines.csv` file from the Labs page. Each row in this file has an airline code and the full name of that airline.
Write a function `create_airlines_table()` which creates a table called `airlines` based on the data in `airlines.csv`. Use the appropriate names for the columns (you can get them from the first line of the csv file).

When you are done, run your function `create_airlines_table()` to create that database (you can comment out the line in the main block or just run the function yourself). You only need to run it once, since you only need to create the table once.
(If you feel like you've messed up the database too much while trying to write this function, you can always download a fresh copy from the Labs page).

Try running a `SELECT *` query on the airlines tables. Make sure you're getting back the correct data.

4 Joining Tables

Sometimes, there are columns in two tables that are similar, which can allow us to create SELECT queries across multiple tables.
For example, the `airline` column in the `flights` table represents the same kind of data as the `code` column in the `airlines` table.
Let's say we want to get all of the full names of the airlines for the flights in the `flights` table. The names are only available in the `airlines` table. So, we will have to **join** the tables together on the airline codes (which both tables have), and then select only the name column from the `airlines` table.
Here is how that would look in SQL:

```
SELECT airlines.name FROM flights JOIN airlines
 ON airlines.code = flights.airline
```

This query statement is larger than we've seen so far. Here is a breakdown of each part and what it is expressing:

1. `SELECT airlines.name`: We want to select the `name` column from the airlines table (since that's the column with the full names).
2. `FROM flights JOIN airlines`: We want to combine the two tables together since we want to get the names for the airline codes for the flights in the `flights` table.
3. `ON airlines.code = flights.airline`: We want to compare these two columns when combining (joining) the tables.

This may be a bit confusing at first, and we will go over it in lecture. For now, try completing the function for this query and run it to see what you get. You can create a multiple line string using
docstring quotes `"""`:

```
query = """SELECT airlines.name FROM flights JOIN airlines
           ON airlines.code = flights.airline"""
```

Compare the selected data to the `airline` column of the `flights` table.

Task: Attempt to complete the remaining query functions. Ask your TA for assistance if needed.