SQLite3

With SQL you can create, read, update, and destroy data in your database. These exercises will cover many of the most common SQL commands, and let you get some experience interacting with a database.

Exercise 0: Checking Basic Setup

You're going to learn SQL through a software package called SQLite3. You can run SQLite3 from the command line. You will need a text editor, and an open Linux or Unix terminal. First, test that SQLite3 is installed correctly. Don't worry about understanding any of this now. Understanding will come gradually over the next several exercises. For now we just want to make sure you can proceed successfully!

Into the terminal window type the following exactly:

```
mkdir intro_sql          (this creates a new folder for our work)
cd intro_sql             (this switches to the new folder we created)
ls                       (this lists all files in the current folder)
```

Since we just created this folder, there should be no files in it, so after `ls` nothing should be returned.

Now enter the following:

```
sqlite3 test_database.db
```

This calls the sqlite3 software and tells it to read a file called test_database. This will be the file that stores the data you create in this session. You should see something close to the following (up to the version number, which may be slightly different) as output in the terminal:

```
SQLite version 3.8.5 2014-08-15 22:37:57
Enter ".help" for usage hints.
```
(If you don't see this version number or an error, please inform a supervisor.)

Now, at the sqlite prompt, enter the following, pressing enter between lines:

```
sqlite> create table test_table (id);      (Don't worry about what this means now…)
sqlite> .quit
```

To quite sqlite3, enter the following:

```
.quit
```

Now, type

```
ls
```

If everything is working correctly, you should see
`test_database.db`

Exercise 1: Create a Table

Let's create our first table! Open the text editor and create a file called ex1.sql. The running database example we'll be using is a collection of facts about the countries of the world.

Enter the following into ex1.sql:

```
CREATE TABLE gdp (
    country_name TEXT PRIMARY KEY,
    country_gdp INTEGER,
    rank INTEGER
);
```

Let's go through this step by step.

Remember, this can be viewed just like the column heading for a spreadsheet. So I have a table that collects this information about countries, and we're setting it up with a *schema*. A schema determines how data should be organized.

The table we'll create, conceptually, looks like this. We filled in some example data so you can get the picture.

country_name	**GDP**	**rank**
United States	17418925	1
China	10380380	2
Japan	4616335	3
Germany	3859547	4

`CREATE TABLE gdp` creates a table in your database called "gdp".
Inside the parenthesis you specify the "fields," or column names for your table.
Each column also has a "data type" specified. So far we've only seen the data types TEXT and INTEGER, but we'll see more.
Each command in sqlite3 ends with a semicolon.

To actually create a database with this schema, run the following command:

```
sqlite3 intro.db < ex1.sql
```

Now enter `ls` to see that the database intro.db was successfully created.

Finally, check that the schema intro.db is correct. Do this by entering the following:

```
sqlite3 intro.db
```

Then, once the prompt comes up,

```
sqlite> .schema
```

you should see

```
CREATE TABLE gdp (
    country_name TEXT PRIMARY KEY,
    country_gdp INTEGER,
    rank INTEGER
  );
sqlite>
```

Exercise 2: Multiple Tables

The value in a database lies not only in the ability to store the data of one table, but to store data in several tables and relate those tables to each other. So far, we have one table in intro.db (with no data in it, but we'll get to that!). Let's add a few more tables to make things interesting.

Create a new sql file in your text editor called "ex2.sql".

Into ex2.sql enter the following:

```
CREATE TABLE population (
    country_code TEXT PRIMARY KEY,
    rank INTEGER,
    population INTEGER
);
```

As an example, the data in this table will look like:

country_code	rank	population
CN	1	1371470000
IN	2	1275510000
US	3	321598000

Now, if these tables were filled out with the information for each country, we would have all the data we needed to calculate *per capita gross domestic product*, or the amount of money each country makes each year per person. However, you might have noticed that the first table we created uses country *name* as its primary key, whereas the second table uses *country code*. As of now, we don't have a systematic way of relating these. A way of doing this is with a third table, called `name_to_code`.

In ex2.sql, beneath the code to construct the `population` table, enter the following:

```
CREATE TABLE name_to_code (
    country_code TEXT PRIMARY KEY,
    country_name TEXT
);
```

Now it just remains to incorporate our new schema info in ex2.sql into our database schema in intro.db. We do this by running the following command in the terminal window:

```
sqlite3 intro.db < ex2.sql
```

Now check that the schema is correct

```
sqlite3 intro.db
sqlite> .schema
```

You should see the combined code from ex1.sql and ex2.sql. So we see that we can always add to the schema of a database without overwriting what was there before. It is also possible to remove undesired parts of a schema, but you don’t have to worry about that for now.

Exercise 3: Entering Data from a .sql file

Now let's finally add some data to our database! We can store the commands to do this, like the commands to create and modify the schema of the database, in a .sql file. Create a file in your text editor called ex3.sql.

Enter the following commands into ex3.sql:

```
INSERT INTO gdp (country_name, rank, country_gdp)
     VALUES ("United States", 1, 17418925);

INSERT INTO population (rank, country_code, population)
     VALUES (3, "US", 321598000);

INSERT INTO name_to_code
     VALUES ("US", "United States");
```

Some things to notice about the above:

- Entries of type TEXT must be enclosed in quotes.
- If you use the form where you specify the order of the fields explicitly after `INSERT INTO table`, you can enter in the values in a different order from which is specified in the schema (for instance, in the `population` insert code above, `rank` does not come first in the schema, but the data will still be entered into the correct column)
- If you do not specify the order of the fields, like in the 3rd `INSERT INTO` above, Sqlite3 will assume you mean the order in the schema (that is, the order used in your CREATE TABLE command). Note how easy it would have been to make a mistake here and enter in

  ```
  INSERT INTO name_to_code
       VALUES ("United States", "US");
  ```

  I probably wouldn't notice that I'd made a mistake now, but later on, when I tried to use county code, I would have an incorrect entry for the United States. It would be like trying to look someone up in the phone book if their first and last name were switched by accident and you didn't know it. That's why **it's always best practice to specify the field names in an insert command**. If you don't quite get this yet, don't worry, we haven't really explained it. Just for now remember this best practice, and in time you'll come to understand why it's important.
- Important note: If you cut and paste the lines above out of this google doc and into a text editor, the quotes will be rendered incorrectly, and the code will not work!

Now, run

```
sqlite3 intro.db < ex3.sql
```

Now that you know how to insert data, go ahead and write a .sql file to insert the data from the tables in the previous exercises, and then insert that data into intro.db. The tables are reproduced below.

country_name	GDP	rank
United States	17418925	1
China	10380380	2
Japan	4616335	3
Germany	3859547	4

country_code	rank	population
CN	1	1371470000
IN	2	1275510000
US	3	321598000

Exercise 4: The Basics of Reading Data: SELECT and FROM

To verify that our data was successfully entered in in the previous exercise, let's learn how to read our data!

In SQL the main read syntax looks like `SELECT <filed_names> FROM <table_name>`. We'll run through some examples below.

Enter in `sqlite3 intro.db` at the command line and run the following commands, taking note of the results you get.

```
SELECT country_gdp FROM gdp;
SELECT country_code FROM popñulation;
SELECT country_code FROM name_to_code;
SELECT rank FROM gdp;
```

Finally to check all the data you've entered so far, you can use a `*` after `SELECT` to mean "all fields in this table."

```
SELECT * FROM gdp;
SELECT * FROM population;
SELECT * FROM name_to_code;
```

A whole table is useful in some cases, but thankfully you also have the ability to select specific columns, in any order you want.

```
SELECT country_name, country_gdp FROM gdp;
SELECT country_gdp, country_name FROM gdp;
SELECT country_code, population FROM population;
```

You may have noticed while doing this that there are some entries you'd like to modify or delete. Perhaps there were some typos in your previous entries. We'll learn how to delete soon enough, but we're going to pause for a moment to get some real data to work with.

Exercise 5: Loading Real Data

Now let's take a shift away from learning about the SQL language itself and learn a few more details about how to actually load data into our database.

So far, we've been manually typing in all of our data, either in a .sql file or at the command line, but obviously for very big datasets, this approach will not scale! Data is commonly stored in many different formats, but for this exercise, we're going to focus on one of the most common, `".csv". "csv"` stands for "Comma Separated Values" and it's exactly what it sounds like; rows of text with columns separated by commas. This means that if I set everything up correctly, all I have to do is "point" one of my tables in sqlite3 at a .csv file and it will put the data into the table. The real work comes in knowing the data well enough to set up the import statement correctly.

For this exercise, you should have access to the three following .csv files:

- country_gdp.csv
- country_pop.csv
- name_to_code.csv
- life_expectancy.csv

Take a moment to look at each of these files in your text editor. Also, for convenience, move them into the same folder as your database (your .db file) from the previous exercises (this is the folder we made in Exercise 0). For each file I want you to answer the following questions: what would be a good name for a table that contained the data in this file? What would be good names for the columns? Guesswork is important here. You can use the names of the files and the previous exercises to inform your guesses. One very important thing is that when you get to specifying the field names, you must make them in the order that they appear in the .csv.

The name of the table is like a variable name. It can be anything, as long as it makes sense to you, or someone else who might read your code. Here are our suggestions:

- country_gdp.csv ⇒ gdp_csv
- country_pop.csv ⇒ pop_csv
- name_to_code.csv ⇒ ntc_csv
- life_expectancy.csv ⇒ life_csv

Now, go ahead and write out a new .sql file called `countries.sql` in which you will use to create a new database called `countries.db` in the same folder as the rest of our work. Use your guesses above in constructing the sql file. Your .sql file should have the commands to construct 3 tables. Don't forget about specifying the data types like `INTEGER` and `TEXT`. Once your work is complete, go ahead and make your database as we did in Exercises 0 and 1.

```
sqlite3 countries.db < countries.sql
```

Now, load your data. The process for loading data from a .csv file is as follows.
Start sqlite3 with your new database, which should already have all the needed tables.
Next, run the following commands in sqlite3:

```
sqlite> .mode csv;
sqlite> .import country_pop.csv pop_csv;
```

That is, a command of the form `.import <csv_filename> <corresponding_tablename>`

Give it a shot. Each time you load a file, check that it loaded correctly by using a command like

```
sqlite> SELECT * FROM pop_csv;
```

Phew, that was tedious, but that's exactly the type of thing you have to do if you want to harness the power of a database; you need to get the data into the database!

Exercise 6: WHERE, AND & ORDER BY

Please read the “Boolean Logic Aside” before continuing with this section.

This exercise is about the `WHERE` command. The syntax for the `WHERE` command looks like:

```
SELECT <field_names> FROM <table_name> WHERE
<statement_that_is_true_or_false>
```

A statement that is true or false is often called a “predicate” or ‘boolean function’. A predicate is the statement, and the boolean is the True or False value that is returned.

Examples of predicates are:

```
country_gdp > 10000000
country_name == “Angola”
population < 500000
```

Say you want to see all countries together with their populations whose population is less than 500,000. We’d do that as follows:

```
SELECT country_name, population_val FROM pop_csv WHERE population_val
< 500000;
```

Find all the countries whose GDP is over 10 trillion dollars.

```
SELECT * FROM gdp_csv WHERE gdp > 10000000000000;
```

What do you notice about your results? You should see that many gdp entries in your results are empty. This is because these tables have some countries as keys which may no longer exist. They also are from 2014 so for some countries the World Bank does not yet have these figures. This also tells us about how sqlite3 chooses to handle empty entries in this case. How would you fix this?

One way of fixing this is by using the `AND` operator between two predicates as follows:

```
SELECT * FROM gdp_csv WHERE gdp > 10000000000000 AND gdp != "";
```

“!=” means “is not equal to” and “==” means equal to.

Finally, you might want to arrange the results you get according to some logic. For instance, in the GDP query above, you might want the names of the countries returned in order of gdp from highest to lowest. In that case you’d add in:

```
SELECT * FROM gdp_csv WHERE gdp > 10000000000000 AND gdp != "" ORDER
BY gdp DESC;
```

Here gdp is the column you want to order by, and you want to see the largest GDP countries first. If I wanted them from lowest to highest, I'd instead write `ORDER BY gdp ASC`.

Now, however, you might be facing the problem of what are the names of those countries?

- Write 2 queries each to find the names of the 3 countries with highest and lowest gdp respectively. To do this, you will have to manually copy data from one query to another. We will learn how to do this on one query later on.
- Write 2 queries each to find the country codes of the most and least populous countries.

Exercise 7: Aggregate functions and GROUP BY

There's one important part of single-table manipulations we haven't touched on yet, and that's aggregate functions and GROUP BY. Before we define what an aggregate function is, we'll just list for you some of the most important ones in SQL:

- AVG() - Returns the average value.
- COUNT() - Returns the number of rows.
- FIRST() - Returns the first value.
- LAST() - Returns the last value.
- MAX() - Returns the largest value.
- MIN() - Returns the smallest value.
- SUM() - Returns the sum.

The idea with GROUP BY is it collects a bunch of rows together based on one column, call it "column_a", so you GROUP BY "column_a", meaning if two rows have the same value for column_a they are grouped together. Then, you will often use an aggregate function like the one above on a second column, "column_b", which combines together all the values of column_b for all the different values of column_a. This is really best understood by example.

Let's say I have the following table of letter grades and ages for students:

Name	Age	Grade	Letter Grade
Demetrius	12	94	A
Keller	11	75	C
Francois	12	72	C
Kat	13	81	B
Daniella	12	88	B
Janae	13	97	A

Go ahead and input this table into a db with a .sql file.

Now, say I want to find the average grade of students within each letter grade. I do that as follows:

```
SELECT letter_grade, AVG(grade) FROM STUDENTS GROUP BY age;
```

```
C|75.0
B|84.6666666666667
A|89.0
```

If I want to know how many students there are of each age

```
SELECT age, COUNT(age) FROM STUDENTS GROUP BY age;
```

```
11|1
12|3
13|2
```

If I want to know how many students there are of each age who scored above an 80:

```
SELECT age, COUNT(age) FROM STUDENTS WHERE grade > 80 GROUP BY age;
```

```
12|2
13|2
```

Note that 11 is not returned as an age because the only student of that age scored lower than 80.

Now, your turn. Turn your attention to the life expectancy table, which we haven't used so far. For many of these you will need aggregate functions and group by, but for some you will not.

- Find the countries with the largest and smallest life expectancies
- For each age, find the number of countries with that life expectancy
- Find the average life expectancy across all countries
- Find the median life expectancy across all countries*
- Fine the mode life expectancy*

* Median is the value which occurs in the middle of the sorted list of values. Mode is the most frequently occurring value.