# Database Admin 101

## Introduction

Now that you've seen how to access and retrieve information from a SQL database, let's investigate how you could create or alter an existing database. Although there is still much to learn, this will lead you into the realm of database administration.

## Objectives

You will be able to:

- Create a `SQL database`
- Create a `SQL table`
- `Create rows` in a SQL table
- `Alter entries` in a SQL table
- `Delete entries` in a SQL table
- Determine `when` it is necessary to commit changes to a database
- Commit changes via `sqlite3`

## Previewing Files in the Current Working Directory

Remember that you can use the bash `ls` command to preview files and folders in the current working directory. Run the cell below to do just that!

```
! ls
```

```
CONTRIBUTING.md
LICENSE.md
README.md
env
index.ipynb
```

## Creating a Database

You've seen how to connect to a database, but did you know creating one is just as easy? All you have to do is create a connection to a non-existent database, and voilà! The database will be created simply by establishing a connection.

```python
# import-sqlite
import sqlite3
# creating-a-connection
conn = sqlite3.connect('pets_database.db')
# creating-a-working-cursor
cur = conn.cursor()
```

## Re-preview Files

If you use the `ls` command once again, you should now see the pets_database.db file there.

```
! ls

CONTRIBUTING.md
LICENSE.md
README.md
env
index.ipynb
pets_database.db
```

## Creating Tables

Now that you have a database, let's create our cats table along with columns for id, name, age, and breed. Remember that we use our cursor to execute these SQL statements, and that the statements must be wrapped in quotes ('''SQL statement goes here''' or """SQL statement goes here"""). Indenting portions of your queries can also make them much easier to read and debug.

```
cur.execute("""CREATE TABLE cats (
                                id INTEGER PRIMARY KEY,
                                name TEXT,
                                age INTEGER,
                                breed TEXT )
            """)

#Creating the cats table
cur.execute("""
CREATE TABLE cats (
        id INTEGER PRIMARY KEY,
        name TEXT,
        age INTEGER,
        breed TEXT)
""")

<sqlite3.Cursor at 0x23baef7dea0>
```

## Populating Tables

In order to populate a table, you can use the `INSERT INTO` command, followed by the name of the table to which we want to add data. Then, in parentheses, we type the column names that we want to fill with data. This is followed by the `VALUES` keyword, which is accompanied by a parentheses enclosed list of the values that correspond to each column name.

**Important**: Note that you don't have to specify the "id" column name or value. Primary Key columns are auto-incrementing. Therefore, since the cats table has an "id" column whose type is `INTEGER PRIMARY KEY`, you don't have to specify the id column values when you insert data.

As long as you have defined an id column with a data type of `INTEGER PRIMARY KEY`, a newly inserted row's id column will be automatically given the correct value.

Okay, let's start storing some cats.

### Code Along I: INSERT INTO

To insert a record with values, type the following:

```
cur.execute('''INSERT INTO cats (name, age, breed)
                  VALUES ('Maru', 3, 'Scottish Fold');
            ''')

# insert Maru into the pet_database.db here
cur.execute('''
INSERT INTO cats (name, age, breed)
    VALUES ('Maru', 3, 'Scottish Fold');
''')

<sqlite3.Cursor at 0x23baef7dea0>
```

## Altering a Table

You can also update a table like this: cursor.execute('''ALTER TABLE cats ADD COLUMN notes text;''')

The general pattern is `ALTER TABLE table_name ADD COLUMN column_name column_type;`

## Updating Data

You use `UPDATE` keyword to change prexisting rows within a table.

The `UPDATE` statement uses a `WHERE` clause to grab the row you want to update. It identifies the table name you are looking in and resets the data in a particular column to a new value.

A boilerplate `UPDATE` statement looks like this:

```
cur.execute('''UPDATE [table name]
                  SET [column name] = [new value]
                  WHERE [column name] = [value];
            ''')
```

### Code Along II: UPDATE

Let's update one of our cats. Turns out Maru's friend Hannah is actually Maru's friend Hana. Let's update that row to change the name to the correct spelling:

```
cur.execute('''UPDATE cats SET name = "Hana" WHERE name =
"Hannah";''')
```

```
# update hannah here
cur.execute('''
UPDATE cats
    SET name = "Hana"
    WHERE name = "Hannah";
''')

<sqlite3.Cursor at 0x23baef7dea0>
```

## Deleting Data

You use the `DELETE` keyword to delete table rows.

Similar to the `UPDATE` keyword, the `DELETE` keyword uses a `WHERE` clause to select rows.

A boilerplate `DELETE` statement looks like this:

```
cur.execute('''DELETE FROM [table name] WHERE [column name] =
[value];''')
```

### Code Along III: DELETE

Let's go ahead and delete Lil' Bub from our cats table (sorry Lil' Bub):

```
cur.execute('''DELETE FROM cats WHERE id = 2;''')

# DELETE record with id=2 here
cur.execute('''
DELETE FROM cats
    WHERE id = 2;
''')

<sqlite3.Cursor at 0x23baef7dea0>
```

Notice that this time we selected the row to delete using the `PRIMARY KEY` column. Remember that every table row has a `PRIMARY KEY` column that is unique. Lil' Bub was the second row in the database and thus had an id of 2.

## Saving Changes

While everything may look well and good, if you were to connect to the database from another Jupyter notebook (or elsewhere) the database would appear blank! That is, while the changes are reflected in your current session connection to the database you have yet to `commit` those changes to the master database so that other users and connections can also view the updates.

Before you commit the changes, let's demonstrate this concept.

First, preview the results of the table:

```
cur.execute("""SELECT * FROM cats;""").fetchall()

#Preview the table via the current cursor/connection
cur.execute("""SELECT * FROM cats;""").fetchall()

[(1, 'Maru', 3, 'Scottish Fold')]
```

Now, to demonstrate that these changes aren't reflected to other connections to the database create a 2nd connection/cursor and run the same preview:

```
conn2 = sqlite3.connect('pets_database.db')
cur2 = conn2.cursor()
cur2.execute("""SELECT * FROM cats;""").fetchall()

# Preview the table via a second current cursor/connection
# Don't overwrite the previous connection: you'll lose all of your
work!
conn2 = sqlite3.connect('pets_database.db')
# create-another-cursor
cur2 = conn2.cursor()
# create-an-sql-execution
cur2.execute("""SELECT * FROM cats;""").fetchall()

[]
```

As you can see, the second connection doesn't currently display any data in the cats table! To make the changes universally accessible commit the changes.

In this case:

```
conn.commit()

# Commit your changes to the databaase
conn.commit()
```

Now, if you reload your second connection, you should see the updates reflected in the data!

```
conn2 = sqlite3.connect('pets_database.db')
cur2 = conn2.cursor()
cur2.execute("""SELECT * FROM cats;""").fetchall()

#Preview the table via a reloaded second current cursor/connection
conn2 = sqlite3.connect('pets_database.db')
# recheck-cursor-again
cur2 = conn2.cursor()
# recheck-sl-execute
cur2.execute("""SELECT * FROM cats;""").fetchall()

[(1, 'Maru', 3, 'Scottish Fold')]
```

Finally, it is important to `close` the connection

```
# first-connection
conn.close()
# second-connection
conn2.close()
```

## Summary

Congrats! In this lesson, you learned how to create, edit, and delete tables and databases using SQL!