CS50's Web Programming with Python and JavaScript

OpenCourseWare

Brian Yu
brian@cs.harvard.edu

David J. Malan
malan@harvard.edu

CS50 Certificate

Ed Discussion for Q&A
Quick Start Guide

---

edX
iTunes U
YouTube

Discord
Facebook Group
Facebook Page

SQL

Databases

- Databases can be used to make it easier for web applications to store, use, and manipulate data. Particularly useful are relational databases; in other words, tables.
- SQL (Structured Query Language) is a language designed to interact with these relational databases. In this class, PostgreSQL will be used, but there are many other versions with slightly different features.

Using SQL

- When making a database or table, it is important to note what type of data will be stored in a given column. Some SQL data types are:

    - `INTEGER`
    - `DECIMAL`
    - `SERIAL` : an automatically incrementing integer
    - `VARCHAR` : variable length of characters, i.e. string
    - `TIMESTAMP`
    - `BOOLEAN`
    - `ENUM` : one of a discrete number of possible values
- In addition to a data type, columns can also have a variety of other constraints:
    - `NOT NULL` : field must have a value; if field does not have a value, entry will be rejected
    - `UNIQUE` : no two fields in this column can have the same value
    - `PRIMARY KEY` : the main way to index a table
    - `DEFAULT` : set a default value for a column if no other value is given
    - `CHECK` : bound values; e.g. values greater than 50
- In order to get a database running, a Postgres server must be set up. To start a server locally on a computer, use the command `psql <database>`. To connect to an online server, use `psql <databaseURL>`.
- After starting up Postgres server, SQL commands can be entered directly into the terminal. Some other useful commands include:
    - `\d` : print all the different parts of the current database

Basic Operations

- Creating a table:

```
CREATE TABLE flights (
    id SERIAL PRIMARY KEY,
    origin VARCHAR NOT NULL,
    destination VARCHAR NOT NULL,
    duration INTEGER NOT NULL
);
```

- Inserting data into a table:

```
INSERT INTO flights
    (origin, destination, duration)
    VALUES ('New York', 'London', 415);
```

    - Note that there is no `id` field. Because `id` is of type `SERIAL`, it will increment and be set automatically.
    -

The order of values in `VALUES` must match the order listed earlier in the command.

- This command could also be entered all in one line.

- Reading data from a table:

```
SELECT * FROM flights;
SELECT origin, destination FROM flights;
SELECT * FROM flights WHERE id = 3;
SELECT * FROM flights WHERE origin = 'New York';
SELECT * FROM flights WHERE duration > 500;
SELECT * FROM flights WHERE destination = 'Paris' AND duration > 500;
SELECT * FROM flights WHERE destination = 'Paris' OR duration > 500;
SELECT AVG(duration) FROM flights WHERE origin = 'New York';
SELECT * FROM flights WHERE origin LIKE '%a%';
SELECT * FROM flights LIMIT 2;
SELECT * FROM flights ORDER BY duration ASC;
SELECT * FROM flights ORDER BY duration ASC LIMIT 3;
SELECT origin, COUNT(*) FROM flights GROUP BY origin;
SELECT origin, COUNT(*) FROM flights GROUP BY origin HAVING COUNT(*) > 1;
```

- The query after `SELECT` indicates what columns are being selected.
- The query after `WHERE` indicates constraints on what rows are being selected.
- `*` is a wildcard that indicates 'all'.
- If a SQL function is passed as a column selector, a column with the return value of that function will be returned. Useful functions include:
  - `AVG(column)` : returns the average value
  - `COUNT(*)` : returns the number of rows returned by the database
  - `MIN(column)` : returns the minimum value
  - `MAX(column)` : returns the maximum value
- `LIKE` is a keyword that takes a template string and returns all rows where the column fits that template. `%` is a wildcard that will match any text. In the example above, any row with an 'a' in the `origin` column will be returned.
- `LIMIT` sets the maximum number of rows to be returned.
- `ORDER BY` organizes rows by a given column in either ascending (`ASC`) or descending (`DESC`) order before returning rows.
- `GROUP BY` organizes rows by grouping the same values in a given column together.
- `HAVING` is an optional specifier for `GROUP BY` which limits what rows are going to be returned, similar to `WHERE`.

- Updating data in a table:

```
UPDATE flights
    SET duration = 430
    WHERE origin = 'New York'
    AND destination = 'London';
```

- `SET` overwrites a column in all the rows that match the `WHERE` query.

- Deleting data from a table:

```
DELETE FROM flights
WHERE destination = 'Tokyo'
```

Relating Tables and Compound Queries

- SQL is a relational database, which means that tables inside a database can be related to each other in some way. In order to do so, we can reference, say, the `id` column of one table A in some other column of table B. Inside table B, the `id` value (which corresponds to table A) is called a 'foreign key'.

- Here's an example to help demonstrate tables related by foreign keys:

```
CREATE TABLE passengers (
    id SERIAL PRIMARY KEY,
    name VARCHAR NOT NULL,
    flight_id INTEGER REFERENCES flights
);
```

- `flight_id` is marked as being a foreign key for the table `flights` with `REFERENCES flights`. Since `id` is the `PRIMARY KEY` for `flights`, that is the column that is `flights_id` will map to by default.

- Once these two tables are created, they can be queried simultaneously:

```
  SELECT origin, destination, name FROM flights JOIN passengers ON
passengers.flight_id = flights.id;
  SELECT origin, destination, name FROM flights JOIN passengers ON
passengers.flight_id = flights.id WHERE name = 'Alice';
  SELECT origin, destination, name FROM flights LEFT JOIN passengers ON
passengers.flight_id = flights.id;
```

- `JOIN` indicates that tables `flights` and `passengers` are being queried together.
- `JOIN` performs an 'inner join': only rows where both tables match the query will be returned. In this example, only flights with passengers will be returned.
- [redacted] indicates how the two tables are related. In this example, the column [redacted] in [redacted]

`ON` `flight_id` `passengers` reflects values in the column `id` in `flights`.

- As before, queries can be constrained with `WHERE`.
- `LEFT JOIN` includes rows from the first table listed even if there is no match (e.g. there are no passengers on that flight). `RIGHT JOIN` is analogous (e.g. passengers with no flights).

- When databases get large, it is often useful to 'index' them, which makes it faster to quickly reference a given column in a table any time a `SELECT` query is made. Note, however, that this takes extra space, as well as time. When updating the table, the index must be updated as well. It is therefore unwise to index every column of every table unnecessarily.

- Nested queries are yet another way to make more complex selections:

```
SELECT * FROM flights WHERE id IN
(SELECT flight_id FROM passengers GROUP BY flight_id HAVING COUNT(*) > 1);
```

- First, in the inner query, a table containing `flight_id` for flights with more than 1 passenger will be returned.
- Then, in the outer query, all rows from `flights` will be selected that have an `id` in the table returned by the inner query.
- In other words, this nested query returns flight info for flights with more than 1 passenger.

Security Concerns

- One potential concern when using SQL is that a user will be able to enter malicious commands into a database. Take, for example, a simple login form that asks for a password and username. What the user enters in those fields might be put into a SQL command to select their account from a table of accounts like so:

```
SELECT * FROM users
    WHERE (username = 'username')
    AND (password = 'password')
```

- If someone guesses that there is SQL code like that above running behind the scenes, they could potentially gain access to someone else's account by entering the follwing as their password: `1' OR '1' = '1`. While this may look strange out of context, when it's processed into the `SELECT` query, this is the result:

```
SELECT * FROM users
    WHERE (username = 'hacker')
    AND (password = '1' OR '1' = '1');
```

    - By putting single-quotes in smart places, the user cleverly edited the SQL query. `'1'` is always equal to `'1'`, so it doesn't matter what the user's password is. The account with username `hacker` will still be returned.
- In order to prevent these so-called 'SQL injection attacks', it is important to 'sanitize' any user input that is going into a SQL command. This means properly 'escaping' characters like `'`, which can drastically change the meaning of the command, so that it is interpeted as simply the `'` character. Otherwise, there is the risk of malicious users editing or even deleting entire databases in this way.
- Another way that things can go wrong is if two users try to modify or access a database at the same time, and SQL commands get executed in an unexpected order. This is the problem of 'race conditions'. Consider a case where a bank information is stored in a database and two customers, who share an account, try to make withdrawals simultaneously. The SQL commands that get executed when money is withdrawn might look like this:

```
SELECT balance FROM bank WHERE user_id = 1;
UPDATE bank SET balance = balance - 100 WHERE user_id = 1;
```

    - First, the customer's balance must be checked to make sure that they have enough money.
    - Then, the balance is updated to reflect their withdrawl.
- Since each command takes some amount of time to run, it is possible that two customers at two ATMs make withdrawls with just the right timing so that the customer B's `SELECT` query runs before customer A's `UPDATE` query. Even though customer A might already have taken the last $100 in the account, since the database hasn't been updated, when customer B asks for $100, the database will allow the withdrawl.
- The solution to race conditions is to implement SQL transactions. During a transaction, the database is essentially locked so that another user cannot make a request until it is complete. A transaction is opened with `BEGIN` and closed with `COMMIT`.

Python and SQL

- In order to integrate these databases into web applications, the Python code running the web server must also be able to run SQL commands. SQLAlchemy is a Python library that allows for this functionality.
- Starting with simple Python outside of a web context, here's how one might go about printing all the flights in the `flights` table:

```
import os

from sqlalchemy import create_engine
from sqlalchemy.orm import scoped_session, sessionmaker

engine = create_engine(os.getenv("DATABASE_URL")) # database engine object from
```

```
SQLAlchemy that manages connections to the database
                                                    # DATABASE_URL is an environment
variable that indicates where the database lives
  db = scoped_session(sessionmaker(bind=engine))    # create a 'scoped session' that
ensures different users' interactions with the
                                                    # database are kept separate

  flights = db.execute("SELECT origin, destination, duration FROM
flights").fetchall() # execute this SQL command and return all of the results
  for flight in flights
      print(f"{flight.origin} to {flight.destination}, {flight.duration} minutes.") #
for every flight, print out the flight info
```

- `flights` is a list of the rows the came back from the SQL query. The individual columns in each row can be accessed with dot notation.

- Data can also be inserted into a database with Python. In this example, the raw data is coming from a CSV (comma-separated values) file:

```
  import csv

  # same import and setup statements as above

  f = open("flights.csv")
  reader = csv.reader(f)
  for origin, destination, duration in reader: # loop gives each column a name
      db.execute("INSERT INTO flights (origin, destination, duration) VALUES
(:origin, :destination, :duration)",
                  {"origin": origin, "destination": destination, "duration":
duration}) # substitute values from CSV line into SQL command, as per this dict
      print(f"Added flight from {origin} to {destination} lasting {duration}
minutes.")
  db.commit() # transactions are assumed, so close the transaction finished
```

- The colon notation used in `db.execute()` call is Postgres' placeholder notation for values. This allows for the substitution of Python variables into SQL commands. Additionally, SQLAlchemy automatically takes care of sanitizing the values passed in.

Incorporating SQL into Web Applications with Flask

- Everything discussed so far can be implemented in the exact same way inside a Flask application. Some of the code to add to `application.py` (along with the necessary import and set up statements) could look like this:

```
  @app.route("/")
```

```
    def index():
        flights = db.execute("SELECT * FROM flights").fetchall()
        return render_template("index.html", flights=flights)

    @app.route("/book", methods=["POST"])
    def book():
        # Get form information.
        name = request.form.get("name")
        try:
            flight_id = int(request.form.get("flight_id"))
        except ValueError:
            return render_template("error.html", message="Invalid flight number.")

        # Make sure the flight exists.
        if db.execute("SELECT * FROM flights WHERE id = :id", {"id":
flight_id}).rowcount == 0:
            return render_template("error.html", message="No such flight with that
id.")
        db.execute("INSERT INTO passengers (name, flight_id) VALUES (:name,
:flight_id)",
                {"name": name, "flight_id": flight_id})
        db.commit()
        return render_template("success.html")
```

- The `try` block of code is always run. If there is an error, and in particular, a `ValueError`, the code in the `except` block is run. The program's flow then continues as normal.
- `rowcount` is a SQLAlchemy feature that is a property of db.execute(), which is equal the number of rows returned by the query.
- `error.html` and `success.html` could be generic templates that render the error `message` and some success statement, respectively.

- The corresponding `index.html`:

```
    <form action="{{ url_for('book') }}" method="post">

        <div class="form-group">
            <select class="form-control" name="flight_id">
                {% for flight in flights %}
                    <option value="{{ flight.id }}">{{ flight.origin }} to {{
flight.destination }}</option>
                {% endfor %}
            </select>
        </div>

        <div class="form-group">
            <input class="form-control" name="name" placeholder="Passenger Name">
```

```
        </div>

        <div class="form-group">
            <button class="btn btn-primary">Book Flight</button>
        </div>

    </form>
```

- Note that some elements, such as the `form-control` class, are Bootstrap components.
- `name` attributes are relevant for referencing them in Python code.
- As is shown, the same dot notation that can be used in Python can also be used in Jinja2 templating.

- Taking this example one step further, it is possible to set up individual web pages for each flight that display some information about that flight. Here's some Python code that would take care of the routing for these new pages:

```
@app.route("/flights")
def flights():
    flights = db.execute("SELECT * FROM flights").fetchall()
    return render_template("flights.html", flights=flights)

@app.route("/flights/<int:flight_id>")
def flight(flight_id):
    # Make sure flight exists.
    flight = db.execute("SELECT * FROM flights WHERE id = :id", {"id":
flight_id}).fetchone()
    if flight is None:
        return render_template("error.html", message="No such flight.")

    # Get all passengers.
    passengers = db.execute("SELECT name FROM passengers WHERE flight_id =
:flight_id",
                            {"flight_id": flight_id}).fetchall()
    return render_template("flight.html", flight=flight, passengers=passengers)
```

- `/flights` is a going to be a generic route to simply display a list of all flights.
- Additionally, `/flights/<int:flight_id>` provides for any individual flight's info page. `<int:flight_id>` is a variable that is going to passed to Flask by the HTML in `flights.html`. This variable is then passed to the `flight` function, which passes the id into a SQL query to get all the info about the flight, including all of the passengers on that flight.

- `flights.html`:

```
<ul>
```

```
        {% for flight in flights %}
            <li>
                <a href="{{ url_for('flight', flight_id=flight.id) }}">
                    {{ flight.origin }} to {{ flight.destination }}
                </a>
            </li>
        {% endfor %}
    </ul>
```

- It's in the link here that `flight.id`, which is an column from the row `flight`, which comes from looping through `flights`, which in turn was passed in from the Python code for `/flights`. It's given the variable name `flight_id`, which is what the python route for `/flights/<int:flight_id>` expects.

- `flight.html`:

```
    <h1>Flight Details</h1>

    <ul>
        <li>Origin: {{ flight.origin }}</li>
        <li>Destination: {{ flight.destination }}</li>
        <li>Duration: {{ flight.duration}} minutes</li>
    </ul>

    <h2>Passengers</h2>
    <ul>
        {% for passenger in passengers %}
            <li>{{ passenger.name }}</li>
        {% else %}
            <li>No passengers.</li>
        {% endfor %}
    </ul>
```

- The only new piece here is using `{% else %}` with a for-loop to account for the case where `passengers` is empty.