Lecture 8

Last time (and next times)

- The CS50 Hackathon is coming up, an overnight event where students and TFs will work together on final projects. See the Muppet Hackathon short video!
- The CS50 Fair will come after, where students will demo their final projects!
- We've been introduced to web programming, where we've learned to use Flask, a framework written in the language of Python, to build dynamic web-based applications.
- The internal structure of our applications have followed a paradigm, or methology, called MVC, Model-View-Controller, where code used for different functions are organized in different files and folders, and interact with each other in predictable ways.
- But until now, we haven't had much code in our Model layer. We've used CSVs to read and write data, but those rows of text are a bit clunky to work with.
- This week's problem set, CS50 Finance will use a database language, SQL, to work with data more efficiently. The problem set will also use a real third-party API, application programming interface, to get real-time data on stock prices, allowing users to "buy" and "sell" virtual stocks.

Logging in

- When we log in to a website, with our username and password, we're not prompted to log in again for each page we visit after, usually until we explicitly log out.
- It turns out that there is another web technology called **cookies**, small pieces of data that a website can ask a browser to store on a user's computer. Then, when the browser visits that website again, it will automatically send that cookie back, like a virtual handstamp that identifies ourself to the server, without having to enter our login information again. The cookie might store a long random string, to prevent adversaries from easily guessing it, and the server will remember that it corresponds to our account.
- When we visit a site like Gmail for the first time, our browser will send HTTP headers like this:

```
GET / HTTP/1.1
Host: gmail.com
...
```

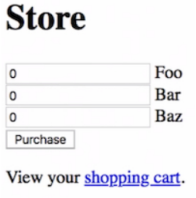
- Then, Gmail's server will reply with the login page. After we successfully log in, Gmail's server will then reply with headers like this:

```
HTTP/1.1 200 OK
Content-Type: text/html
Set-Cookie: session=value
...
```

  - The `Set-Cookie` header asks our browser to save the `session` and `value` key-value pair to our computer; `value` will be a long random string or number that identifies us to the server.
  - If we, as the user, set our browser to not save cookies, we'll have to log in for every page we visit. But cookies might also identify us to advertisers, who can then track us across different sites we visit, if those sites include embedded images or scripts that are from the same third-party advertising service. So political entities like the EU have passed laws to help ensure companies behind websites are explicit to users about the purpose of cookies they want to store.
- When we visit Gmail again later, our browser will send the same value back as part of the `Cookie` header:

```
GET / HTTP/1.1
Host: gmail.com
Cookie: session=value
...
```

  - And cookies can be set to expire by the server, which is why after some number of days, we might be asked to log in again.
- In today's source code directory in the CS50 IDE, we'll first look at the example called `store`. We'll `cd` into the `store` directory, and call `flask run` in our terminal to start our IDE's web server. Then, we can visit the link to see a simple "store":



  - We can change the quantity of each item, and click "Purchase" to see them added to a virtual cart that tell us the count of each item we have.
  - We can keep shopping, but even if we close the window and reopen it, we see that our cart still saved the number of each item we added before.
- With cookies, we can implement **sessions** on our server. A session is an abstraction of saved state for each user's visit to our website; our server might give me a cookie with `session=12345` and you a cookie with `session=78910`, and store some data for each user who visits, based on that session value.
- With Flask, we only need a few lines of code to use this abstraction:

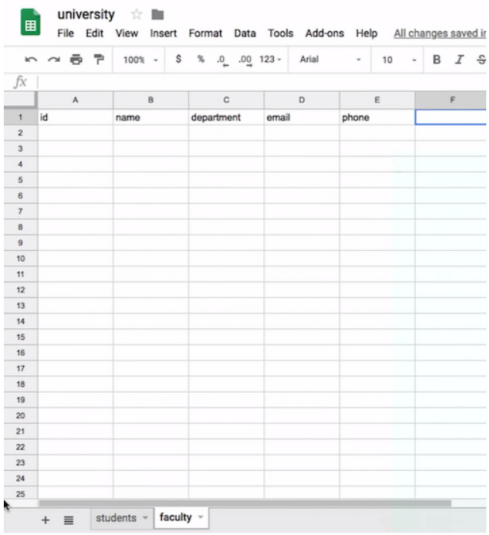
```
...
```

```
from flask_session import Session
...
app.config["SESSION_PERMANENT"] = False
app.config["SESSION_TYPE"] = "filesystem"
Session(app)
...
@app.route("/update", methods=["POST"])
def update():
    for item in request.form:
        session[item] = int(request.form.get(item))
    return redirect("/cart")
```

  - We set up our Flask app to use the Session library, and in each of our routes, we'll have access to a `session` dictionary, which we can store data in our server's memory or filesystem for each specific user.
  - We'll introduce this library in a bit more detail in this week's problem set.

Databases

- So far, we've seen how we can store data in CSV files. But finding data requires linear search, and we have to open, read, change, and save the entire file if we want to make a change.
- Databases are a set of data, usually organized and managed by some software for us. Database management software such as MySQL and Postgres commonly allow for selecting, inserting, updating, and deleting data with a language called SQL, Structured Query Language.
- A spreadsheet with rows and columns is like a simple database. Each column is a specific field of data, and each row contains values for one entry in the database.
- For example, we might store information about students, with a column for an ID, a column for a name, and so on.
- We might have different sheets, or tabs, within the same spreadsheet, and in databases these would be called tables.
- With Google Sheets, we can create a spreadsheet called "university", and create two sheets within that, one called "students" and one called "faculty":



- With a database, we can represent data in the same way, and SQL also requires us to specify the type of data we want to store in each field. By specifying the type, our database software can store and optimize our data more efficiently.
- There are different variants, or dialects, of SQL, depending on what database program we're using. In SQLite, a popular, lightweight software we'll be using, data types include:
  - `BLOB`
  - `INTEGER`
  - `NUMERIC`
  - `REAL`
  - `TEXT`
- And within the `INTEGER` type, we might specify the size as a `smallint`, `integer`, or `bigint`, each of which might be stored with a different number of bytes. We might be tempted to use a `bigint`, for example, but that might use unnecessary space and be more and more costly as we have more and more rows to store.
- `REAL` numbers can be a `real` type for a floating-point number, or `double precision`, with more bytes allocated.
- The `NUMERIC` type represents other types of numbers, such as a `boolean`. `date`, `datetime`, `numeric(scale, precision)` (for decimal numbers with a specific number of digits), `time`, and `timestamp`.
- A `TEXT` field can be a `char(n)` field, a fixed number of characters; a `varchar(n)`, a variable number of characters up to *n*, or a larger `text` field with no specified maximum. And we can infer, from our experience with arrays in C, that a fixed number of characters for each row would be faster to index into, since we can calculate exactly where each value will be. Our database software will provide this abstraction, and use the right data structures and algorithms for storing and accessing our data, faster than something we might be able to implement ourselves.

SQLite

- We can open the CS50 IDE and use the terminal to explore SQLite, a popular database management software. SQLite is a technology for storing data on a server's disk as a binary file, so it doesn't have a server or other software to set up. (Other technologies, like Postgres and MySQL, use a running program that acts as a database server, which has better performance but requires some configuration and memory.) Instead, we'll use the `sqlite3` program on our IDE as a human interface to a database file, and in our code, we'll use abstractions that can open and work with an SQLite database file.
- We'll start by typing `sqlite3 froshims3.db`, and we'll be able to create a table in that database with a command like `CREATE TABLE 'registrants' ('id' integer, 'name' varchar(255), 'dorm' varchar(255));`. We specify the name of our new table, and for each column or field, the type of data. And by convention, we use 255 for our `varchar` fields, since that used to be the maximum for many older databases, and are probably enough for all realistic possibilities, without being too excessive.
- Nothing happens at our command line after, but we can type `.schema` and see the schema, or description, of our table:

```
sqlite> CREATE TABLE 'registrants' ('id' integer, 'name' varchar(255), 'dorm' varchar(255));
sqlite> .schema
CREATE TABLE 'registrants' ('id' integer, 'name' varchar(255), 'dorm' varchar(255));
sqlite>
```

- We can add a row to our table with `INSERT INTO registrants (id, name, dorm) VALUES(1, 'Brian', 'Pennypacker');`, and conventionally the uppercased words are SQL keywords, while the rest are words specific to our data.
- We can see our table with `SELECT * FROM registrants;`, and see our table printed out.
- And we can easily filter our data with `SELECT * FROM registrants WHERE dorm = 'Matthews';`. We can specify just the fields we want to get back, too, with something like `SELECT name FROM registrants WHERE dorm = 'Matthews';`
- We can change rows with something like `UPDATE registrants SET dorm = 'Canaday' WHERE id = 1;`.
- We can delete rows with something like `DELETE FROM registrants WHERE id = 1;`.
- The CS50 IDE also has a graphical program, phpLiteAdmin, which can open SQLite files too. We can double-click `froshims3` in our files list on the IDE, and be able to browse rows. We can try to insert a row, too, by clicking on the name of the table and the Insert link, and see the SQL that phpLiteAdmin runs:

- We can start over by deleting the file `froshims3.db`, and creating a blank file with the same name. Now we can double-click it, and phpLiteAdmin will let us create a new table. We'll create 7 fields this time, and we'll have more options:

  - For our first field, `id`, we'll make that an `integer`, and we can use the Primary Key option to indicate to our database that this column will be the one used to uniquely identify each record. We'll use autoincrement so our database will automatically provide the next value for `id` each time we add a record, and the Not Null option ensures that there is a value for that field for each record.
  - Then we'll have a `name` field that is a `varchar` with a maximum length of 255, and make that Not NULL.
  - We'll have a `dorm` field, `varchar` with a maximum length of 255, and a `phone` field that we'll set to a fixed `char` of 10 characters. If we were to use a numeric field, phone numbers that start with 0 would lose those leading zeroes, and we might consider needing more characters if we wanted to support international phone numbers.
  - We'll have an `email` field as a `varchar` with maximum length 255, and store a `birthdate` as a `date`. For `sports`, there might be more information we need someday, so we'll have that as a `varchar` with a maximum length of 1024, an even power of 2.
- We can add fields to the table later, too.
- We can click the SQL tab to insert rows manually, or use the Insert tab, but writing code to execute queries will be lead to the most organized data, since we'll be able to set everything consistently.

SQL

- We can import the CS50 SQL library to execute queries easily, with `lecture.py`:

```
from cs50 import SQL

db = SQL("sqlite:///froshims.db")

rows = db.execute("SELECT * FROM registrants")

for row in rows:
    print(f"{row['name']} registered")
```

  - Here, we're opening a file called `froshims.db` in the same directory and calling it `db`, using `SQL` to open it. We call `db.execute` to run a query, and save the results into the `rows` list. Then, we can iterate over each row and print out any fields we'd like.
  - We can run `python lecture.py`, and the CS50 SQL library will also helpfully print a debug line showing what we sent to the database.
- Since we can query a database in Python, we can also integrate that into a Flask application. We can create a Flask `application.py` file:

```
from flask import Flask, render_template, request

from cs50 import SQL

app = Flask(__name__)

db = SQL("sqlite:///lecture.db")

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

  - We select everything from the `registrants` table and store that in a variable called `rows`.
  - Then, we pass the results, `rows`, to the template.
- In our template, `templates/index.html`, we'll iterate over a list of rows that are passed in, displaying each one's `name` field:

```
{% extends "layout.html" %}

{% block body %}

    <ul>
        {% for row in rows %}

            <li>{{ row["name"] }} registered</li>
```

```
        {% endfor %}
    </ul>

{% endblock %}
```

- We can implement a search functionality too, by adding a `q` URL parameter:

```
...
@app.route("/")
def index():
    q = request.args.get("q")
    rows = db.execute(f"SELECT * FROM registrants WHERE name = '{q}'")
    return render_template("index.html", rows=rows)
```

  - Now, only rows that have a matching name will be returned to our template to display.

lecture.db

- A sample database of music metadata, `lecture.db`, is in this week's source directory. Importantly, it demonstrates how we can relate data in different tables.
- With our database of students, we might have noticed that the `dorm` field will have the same strings repeated over and over again. Instead of storing the same data, we can store a reference to some other table of dorms, using fewer bytes to represent the same data.
- In the sample database, we have a table for Albums, Artists, and Tracks. In the Album table, each row has an AlbumId, Title, and ArtistId. The Artist table has an ArtistId and Name for each row, so by joining the two tables together, we can figure out the artist's name. And since each artist has multiple albums, we're saving space. If a row in the Artist table has more data, we can update it just once, rather than in every row of the Album table, if that data was repeated there.
- We can use phpLiteAdmin in the CS50 IDE, as before, to look at the tables and rows in `lecture.db`, and run a query like `SELECT * FROM Album WHERE ArtistId = 1;` to see all the albums by the artist with ID 1. We can use SQL to join tables, getting the artist's name too, with `SELECT * FROM Album, Artist WHERE Album.ArtistId = Artist.ArtistId;`. The `name` from the Artist table will also be selected (since we said `SELECT *`), and matched to each row in Album where the `ArtistId` field is the same. Another way to express the same idea would be `SELECT * FROM Artist JOIN Album ON Artist.ArtistId = Album.ArtistId;`.
- Normalizing our database is this method of storing redundant data once, and using a reference to another table as needed to save space, with a slight cost to performance and simplicity.
- In SQL, we can add a `UNIQUE` constraint to a field, without using it as a primary key. We can also indicate that a field should be an `INDEX`, where the database should build an index (with a tree, hash table, or some other data structure) for looking up fields more quickly. Finally a `FOREIGN KEY` is what we would call a field that refers to a row in some other table; for example, the `ArtistId` field in the `Album` table is a foreign key to the `Artist` table.
- SQL also has functions for numeric operations like `AVG`, `COUNT`, `MAX`, `MIN`, and `SUM`.

Problems

- One problem with databases is **race conditions**, where the timing of two actions or events cause unexpected behavior.
- For example, when we sign up for a new account on a website, it might ask us for a username we'd like. If the username is taken already, we'll see a message that tells us so, and if not, we're able to start creating an account with that username. And if we take our time with putting in the rest of our information, that username might be taken by someone else by the time we actually submit the form. But if the web server didn't check again, there would be a problem where the same username is now reassigned to us!
- If two people, or web server threads, are checking the state of a variable at the exact same time, and then make a change based on that, after some amount of time, then there is a race condition in that window of time.
- Another example is two people, withdrawing money from two different ATMs at the exact same time, with the same account information. If the account has $100, but both people try to withdraw $100 at the same time, each ATM might check the account balance, see there is a balance of $100, and saves $0 back into the account. But each ATM did this, so a total of $200 would have been withdrawn!
- Another example involves two roommates and a fridge. The first roommate comes home, and sees that there is no milk in the fridge. So the first roommate leaves to the store to buy milk, and while they are at the store, the second roommate comes home, sees that there is no milk, and leaves for another store to get milk. Later, there will be two jugs of milk in the fridge. By leaving a note, we can solve this problem. We can even lock the fridge so that our roommate can't check whether there is milk, until we've gotten back.
- In the database world, we can also lock rows and tables. We can use **transactions**, where a set of actions is guaranteed to happen together. That property is called **atomicity**, where, for example, we can check the value of a row and change it, without anything else being able to read or change that value.
- We might have also seen some websites, like for airline tickets or hotel rooms, provide a window of time after we add something to our cart, that reserves it for us, letting us fill out our information without someone else purchasing it in the meantime.
- Another problem in SQL is called a **SQL injection attack**, where an adversary can execute their own commands on our database. Earlier, we passed in the `q` URL parameter to filter registrants based on their name, with `rows = db.execute(f"SELECT * FROM registrants WHERE name = '{q}'")`. But if `q` had the value of `Brian'; DELETE FROM registrants WHERE name = 'Brian`, it would end our previous statement and run another statement.
- To guard against this, we can sanitize user data, or escape characters like semicolons and single quotes, such that they are interpreted as part of the string, rather than special characters that end strings or commands.
- The CS50 SQL Library allows us to escape user input with the `execute` function, and we can write `rows = db.execute("SELECT * FROM registrants WHERE name = :name", name=q)` where we use a special placeholder, `:name`, that will be escaped before it is substituted into the string.
- Another example would be typing in `' OR '1' = '1` in a password field; if the query is `db.execute(f"SELECT * FROM users WHERE username = '{username}' AND password = '{password}'")`, then substituting that password would get us `db.execute("SELECT * FROM users WHERE username = 'me@examplemailprovider.com' AND password = '' OR '1' = '1'")`, and that would select that user since `1 = 1`.
  - On the other hand, the escaped input would be substituted as `db.execute("SELECT * FROM users WHERE username = 'me@examplemailprovider.com' AND password = '\' OR \'1\' = \'1'")`, preventing the intention of our command from being changed since the single quotes are escaped.