CS50 Web Programming with Python &JavaScript

Lecture 0: Intro to Git, HTML/CSS

Used for version control—collaborating on code, deploying it to testing then production, checking back on changes you made over time. Git synchronises code between people working on the same code automatically. Git is a version control system, and GitHub is the way in which you access the repositories people make using Git.

A GitHub repository is just where you store and keep track of all the code in your project. You must type Git code into the command interface (terminal), whereby it does things with your code, including uploading it to GitHub.

Commands:

git clone <url> // downloads the file associate with that URL onto your computer
git add (*) <filename> // saves code that you've written, to later commit to a repository on GitHub when you next save
git commit -m "message"// this just essentially moves the code you've written on your computer to a web server repository, with a message detailing what the code does/adds to the existing versions. what this does is have Git save a snapshot of this new version of the code (*with t*he change, as well as older versions)
git push // this will send all commits to be saved on your GitHub profile automatically
git status // this tells you where you are relative to the master file last saved on GitHub—you can configure this master file to be whichever repository you want by linking your computer to your GitHub account on the terminal window. If someone makes a change to the code— either pushing from their computer, or changing directly within GitHub, you can use *git pull* to download the new code onto your computer.

sometimes when you make a change, and someone else has made a different change and committed it, you get a "merge" conflict as your change no longer reflects a change on the newest version. in this scenario, GitHub will notify you and ask you to change your version accordingly, before you can commit the result.

pipeline: add—> commit—> push

git log // shows you all the commits you've made, who and when made the change

git reset - - hard <commit hash number> // you can find the hash number from git log
git reset - - hard origin/master // this resets to the version you originally got the code onto your computer from (GitHub stores the "master" file)

git commit -am "message" // this adds and commits in the same line the reason that adding and committing were created, as two separate steps is that, often, you're working on many files in parallel. git commit commits all the files you added, and so how do you choose to push some files but not others—by adding.

in terminal, you can make shorthand for these commands using things like *alias ga = "git add"* etc.

think about the "document object model" which is a way of visualising HTML code as a tree, with each sub tag going within a parent tag, and this is useful for understanding how browsers parse HTML, and how they find things within a webpage (important for things like PageRank, etc.)

when making style tags, you can just do element1, element2 { CSS } to give them both the same style properties.

to collapse a double border you get by adding a "boder: 1px solid blue" to a table's rows and columns, just know the border-collapse: collapse CSS property.

remember that you can only give one element an id but you can give lots of elements the same class.

in addition to "div" you have "span", which is used more in-line for specific portions of text or things within a larger framework, whereas div is more general and used for larger sections of code. they are often used together, with span nested inside div to give a specific portion of it some particular properties.

You can deploy your code to the internet, if it's a small personal project, by using GitHub pages, a service that GitHub makes whereby you can take any repo you've pushed to GitHub and deploy it as a functional site online by just going to settings and deploying that repo and saving the link.

Lecture 1: Advanced Git, HTML, CSS

Advanced Git

Git allows a technique of modifying code called "branching". Sometimes you want different people to take the original code base (master branch) and take it in different directions, each experimenting with adding some new feature. They would then clone the master branch, and start a new branch and work on that feature there. Then, when they're done developing the feature, they can "merge" that branch with the master branch.

new Git commands:

git branch // tells you all the branches that exist, as well as what branch the code you're working on is in
git branch nameOfBranch // creates a new branch of code
git checkout nameOfBranch // switches to another branch
git log // this shows you the commits you've made *to a particular branch*
git merge nameOfBranch // this merges one branch with the master branch
git push - - set-upstream origin nameOfBranch // this creates a new branch *on the online repo*

very often, you create a new local branch (on your computer) and when comfortable they're working as intended, just combine them with the master branch *on your computer* and then push the updated master branch

a *remote* is the version of a repo that lives on the web/online. origin is the name of the remote master branch that is on GitHub. so, every time you do git pull, what you're really doing is git fetch (obtain the origin remote and put it onto my computer, with my master branch being still what it is) and git merge origin/master (which means merge my master branch, which I am currently on, with the origin/master branch, that is, the one I just got from GitHub, which now sets that updated branch as my new default working branch).

you can also start a "branch" at the very beginning of the master branch, in which case it's called "forking" the code since you're starting at the very beginning, leaving the original code unaffected, and starting your own path modifying it—which may include branches of its own. I'll do this with the cs50 projects for example, forking their code and then doing my own thing with it. it's basically copy+pasting then modifying, but more separate work on it to the main code than mere branching.

a *pull request* is when you're working on a project, and have made some changes on a branch/fork, and want to integrate the new code with the original code base (like in many open-source projects) and want to get other people's feedback before you merge the code with the original repo. this is usually a way to talk about changes and get a conversation started about your code.

Advanced HTML

you link to things internally by setting href = "#nameOfThingWthisID"

in the change from html4 to html5, <header class="header"> has become native as <header>. same for nav, footer, section, audio, video, datalist (for autocomplete, etc.) and more. datalist, for example, you might use in a form asking for someone's country, with the datalist taking in text but having many nested "option name = "United Kingdom" tags of the like nested within it.

Advanced CSS

Responsiveness is crucial in the modern age. When Ben visited CareerFear, it was on his mobile, not on computer. When Gaspard browses the web every evening, it's on his iPad, not his laptop.

Modern CSS comes with a lot of flexibility with its selectors—that is, the h2 {} or the .className {} so that you can change the styling on a very specific part of your site easily, which will become important as your site grows in complexity.

a pseudo-class is a special case, or particular state, of some element—like hover.

a pseudo-element is like an extension of an actual element, like before or after. another good example is selection, as when you select part of an element, it's almost as if you're creating another element.

examples of CSS tricks:

```
h1, h2 {
	property: value;
}

ul li {
	applied to <li> within a <ul>, but it applied to ANY descendants of <ul> that may be <li>, even if you had another nested list that was an <ol>
}

ul > li {
	this ONLY applies to immediate children of the ul property and not grand-children or any other descendants nested within it
}

h5[class = "className"] {
	this ONLY applies to h5 tags that have a certain class
}

a::before {
	content: "\21d2 click here:"  // the cryptic number is symbol for an arrow symbol, and content property just takes some text or actual content that might otherwise seem to go in the HTML section of the site
}

p::selection {
	background-color: yellow;
}
```

Mobile Responsiveness

How do we tackle this? Media queries, Flexbox and Grid.

You can set something's CSS properties to display: none in the case that it's a computer screen only.

examples:

```
@media (max-width: 500px) {
        property: value;
}
```

you can even concatenate things we've learned to be:

```
@media (min-width: 500px) {
        h2::before {
        content: “Welcome to my Web Page!”;
        }
}


@media (max-width: 499px) {
        h2::before {
        content: “Welcome!”;
        }
}
```

when you're adding functionality for other screen sizes, you want to include:

<meta name="viewport" content="width=device-width, initial-scale=1.0"> which just tells the browser to redefine it's viewport (screen) width to the width of the device it's being viewed on, so it doesn't display text on a phone as if it were on a computer.

Flexbox and grid are actually designed to make responsiveness easier to implement. You'd set:

```
.mobileResponsive {
        display: flex;
        flex-wrap: wrap;
}
```

which would, if the screen size decreases, change, say, the 6 elements you have in a row to 5, 4, and so on, would make 6 rows of 6 into, say, 12 rows of 6 if you were to view it from a phone as opposed to a laptop. Note that flex box only controls the re-laying of the individual amounts per row as opposed to controlling the layout of the grid as a whole.

But using these flex box and grid features, implementing each property by hand each time can become tedious. Enter Bootstrap. They've implemented a host of different web components using detailed flex and grid analysis to make beautiful, responsive layouts that will change seamlessly depending on what they're viewed in. You use these by calling specific classes (learning about all these classes that you can use is what people mean by "learning Bootstrap")

The way Bootstrap works is by, as I initially theorised when learning CSS grid, dividing the screen into 12 columns, and then customising each one of those. It's built *mobile first*, so all of the CSS has been written to look most beautiful on mobile, and then scaled up to web size.

you add Bootstrap CSS to your site by doing <link rel="stylesheet" href="somelink.com"> and so then onwards it knows what you mean whenever you call a class you didn't explicitly declare yourself. when you would normally need to include display: flex; flex-wrap: wrap; and more to get it to be mobile responsive, you just call bootstraps code to style it as you want, and all of the responsiveness is taken care for you.

to change the number of columns you use, you set class = "col-X" where X represents the number of columns that you want your element selected with that class to span. you can change the columns something takes up by changing its class in the case of a small or large screen in the following way: “col-lg-3 col-sm-6” so it occupies more space on a smaller screen (and is thus clearer) in the class declaration.

if your own stylesheet’s class declaration conflict’s with bootstraps, they’ll get merged unless there’s direct contradiction, in which case the more specific selector will be used.

Using SASS

As your own stylesheets start to get more complicated, and you want to include variables in your CSS, you can’t actually do that with CSS alone, but it’s made easy with SASS, a framework that extends CSS and adds additional functionality. You’d just do that by declaring $color: red; at the top of your document, and anywhere else it says color, it’s just replaced with “red”, making changes easy to implement.

But browsers don’t understand .scss files, so you have to download SASS and compile these into CSS programs by using the command: *sass nameOfSASSfile.scss nameOfCSSfileYouWant.css*

and that *css* file will be used in the actual web app you’re implementing. but if you don’t want to keep recompiling a .scss file every time you make a change, just use the command: *sass - - watch name.scss:desiredName.css*

Many sites like GitHub will have in-built support for SASS, whereby if you push a .scss file to GitHub, it’ll automatically compile it into CSS and use that.

another feature of SASS is nesting. you can write:

```
div {
        property: value;
}


div p {
        property: value;
}


div h2 {
        property: value;
}
```

OR could abstract that down to:

```
div {
        property: value
        p {
        property: value }
        h2 {
        property: value }
}
```

which just makes things slightly easier to write.

another important functionality feature of SASS is in the % operator, whereby you can abstract away a layer of implementing CSS in the same way to many slightly different outcomes. say you have an alert you’re designing the style of using CSS, and you have one green “success” alert, and one red “danger!” alert (amongst others)—it feels redundant that most of the lines of CSS they share (font size, family, weight, bg color, position, width, border-radius, etc.) will be be the same, with just the colour (font) changes. So if you set up a “template” %alert {CSS} at the top of the SASS file, you can create:

```
.successAlert {
        extend %alert;
```

```
	color: green;
}
```

though frankly, it’d be equally easy to manually write the CSS by doing .successAlert, .dangerAlert {commonCSS}

```
.successAlert {
	unique: value;
}

.dangerAlert{
	unique: value;
}
```

and that doesn’t require any new compilation since it’s not anything SASS.

Lecture 2: Python with Flask

Intro to Python

You can run Python by typing “python3 filename.py” in terminal.

While in C you’d store NULL, the equivalent is None in Python.

you can use x[n] notation to get at the first part of any group of things—characters within a string, strings within a list, values inside of a tuple (it’s an ordered group of numbers—x,y remember)

a set and dictionary, importantly, are *unordered*. you’d use set() to create an empty set, and setName.add(value) to add it into the set, much like .append(value) into a list.

for i in range(n) will repeat n times, from 0 onwards, and therefore end up at n-1.

you can do *from* module *import* function from either some external, standard, library, OR you can do it from another file itself on your local server (in this case, module.py)

the reason that it’s good practise to indent your whole program into a main function and then call the function at the end is because if you just called module.py and asked to import a function from it, you’d import the function but also execute everything in the program, which you don’t want to do. By adding that small layer of complexity, you’re then able to import specific functions from other files without running the entirety of those programs.

```
[
if __name__ = “__main__”
	main()
]
```

you can create “objects” in Python by creating their equivalent, classes (it’s an OOP language) by defining:

```
	class objectName:
		def __init__(self, property1, property2…):
			self.property1 = property1
			self.property2 = property2
```

where “self” was an abstraction literally created to refer to the thing itself. so when creating objectName, you’d need to specify the name of the object, and two properties in this case, which in future you could then access through dot notation.

Intro to Flask

Fundamentally, whenever you type in a site, your computer is sending a textual signal to the web server, in a given structured format (number first, signifying what what they're asking for, then type of file they're expecting back—and these agreed upon things comprise HTTP). Writing back-end code basically involves writing computational logic that will generate appropriate HTML/CSS based on the request received.

Flask is some code written in python. do not forget this. there are a few key lines of code you want to include to set up a flask web application:

```
from flask import Flask

app = Flask(__name__)

@app.route("\")
def index():
        return render_template("index.html")
```

to run the application, you'd just go to the directory in which the file is located, and type flask run (already configured to get it to work)

to add the first element of dynamism into the site, you might create a route that creates HTML that dynamically says hello to anyone in the ULR, not just some hardcoded values. this would be done by:

```
from flask import Flask

app = Flask("__name__")

@app.route("/")
def index():
        return "Hello, world!"

@app.route("<string: name>")
def hello(name):
        name = name.capitalize()
        return f"Hello, {name}"
```

notice the HTML-esque writing within the app.route, as well as the fact that since Python is OOP, and name (like most things, in some way) is an object, it has a method capitalise() built into and tied with it, that just capitalises the first letter of the name.

you can actually add conditionals in Jinja2, in the same way as you might add {% for a in b %}, and you can add html that will be displayed in one condition {% if boolA %} as well as in the {% else }% condition. Just make sure to create the boolean condition in the Python file, as well as to endif inside the Jinja.

you can link to other pages by defining a link in Jinja2 in terms of a function that you defined in the app.route as opposed to a particular page, which hs a name that is subject to change. For example, if you define the "cat.html" page to come under the route def cat(): then even if the name of the page changes, using {{ url_for('cat') }} will remain the same.

when creating layout.html, you define the customisable bit as:

```
        {% block body %}
        {% endblock %}
```

or {% block heading %} {% endblock %}

then use {% extends "layout.html" %}

when you have a form, you want it to do something upon submission; in other words, to have an *action,* which you can define as the URL to which you want to send the "POST" request (by defining an addition attribute called method="post" right afterwards), so you might do:

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

or, more concretely, with associate back-end example given:

say we had a form called form.html:

<form action="hello.html" method="post">
<input type="text" name="name">Enter your name here </input>
…
>> in Python back-end you'd have the associated:

```
@app.route("hello.html", methods=["POST"])
def hello():
        name = request.form.get("name")
        return render_template("hello.html", name=name) // the name here connects front-end to back-end!
```

note that if you tried to access URL/hello.html, that wouldn't work since it hasn't yet been set up to handle GET requests. if you do allow that, though, you need to make sure you're explicit defined a path for *if request.method == "GET" and "POST"* within the app route function definition. Also, the "methods=["POST"]" in the app.route of the back-end, is what method the *user submits by, and not indicative of the fact that you "post" the page*.

you can use dot notation in Jinja2 to access particular parts of a Python dictionary that you're iterating or manipulating over in the Jinja part of your HTML.

When you're using the AJAX technique, you transport data between the server and front-end in JSON notation, where JSON is JavaScript object notation. This literally means data is stored as a Javascript object; more specifically, as objects with several key-value pair properties (like employees object storing several first and last names of, say, 6 employees). This notation makes it easy to manipulate the data received from the server, and is why data is often transported in this route when the front-end and back-end communicate with each other.

The Jinja2 {{ value }} draws from the Python, which in turn, draws from the "name" = part of the front-end, which is how you go from input to output through the tech stack.

the "form action="" " attribute dictates where you go after pressing "submit", since you don't otherwise specify that submitting the form takes you down a particular URL route.

The default request method on a page when you visit its page is a GET request. and so if you just type in the /hello page, you're sending a GET request to a page that's normally only brought up when the form submits something to it via POST, hence triggering an error. When you submit data via GET, the data gets put into the URL via an HTTP parameter like hello?name=name

sessions

used to retain users' information. if you wanted to build, say, a notes application, where you can type notes into an <input> and have them show up next time you open up the browser as, say, a to-do list. you would first import session from flask, a built-in function that will help keep track of every different users notes.

if you just, say, defend a global notes = [], then you wouldn't be able to distinguish between who's notes to display/who is currently logged in. session functionality does all of this, showing only *your* notes (not someone else's) to you (but this resets when the server shuts down—for more permanent data storage, use databases). more concretely, you might implement:

```
from flask import … session

app.config("SESSION_PERMANENT") = False
app.config("SESSION_TYPE") = filesystem
Session(app)
```

```
@app.route("/")
if request.method == ["POST"]:
        if session["notes"] is None:
                session["notes"] = []
        else:
                note = request.form.get("note")
                session["notes"].append(note)
return render_template("index.html", notes=session["notes"])
# the notes was then defined as showing each element in a list in Jinja in index.html to correspond to the session functionality here
# the reason we use session["notes"] is because session, as an object (python is OOP), is a dictionary by default, and we're trying to access the "notes" key, which has a value of a list.
```

Lecture 3: SQL

We're working with PostgresQL here. A serial data type is like an auto incrementing counter (like flight #).

use "psql" to say that you want to start specifying postgres commands into the terminal, with the name of the fine you want to apply those commands t coming afterwards.

when creating a table, make sure to specify constraints as well as data types for each column. for example, in the flights table, you'd have "origin" being specified in the CREATE function as: origin SERIAL NOT NULL where the first specification is a data type (autoincrement) and the second one a constraint. alternatively, you might have 'duration' as INTEGER CHECK >500, with other contstraints being like "default X" or "unique".

use \d to get a list of all tables you have up and running at the moment after you enter "psql" to get into sql mode.

just as you have sum() of the values in an int column, you have another function avg(duration) and count(*), the latter to count how many flights meet a certain requirement—which you can imagine would be useful if you have a table with thousands upon thousands of flights running every day. also, there's min(duration) and max(duration), too.

you can also use the "IN" command just as you append "WHERE" at the end of things, like select…WHERE destination IN ('LHR', 'AUH') to return all flights going to those two places. even more powerfully, you have the 'LIKE' command, select…WHERE origin LIKE ('%a%') to get everything that is flying from an origin that has the letter a in it—the % is merely a placeholder.

when you pay for a web service like AWS, Azure or Heroku, you're not just paying for cloud storage that serves any requests your site gets from around the world, but also for database infrastructure (like versioning etc.) that they offer you access to when manipulating the data you're taking in and feeding out.

if you have something with a certain SERIAL UNIQUE id, and you delete it, the id is lost with it, and no other term in the table then inherits that id (if you're not setting id and it's being auto-incremented, that is)

you can also execute the ORDER BY command (with optional ASC/DESC) to order query results by a certain criterion, and use the LIMIT X constraint to only get the X longest or shortest flights in your table.

you also have the HAVING command that you add like a "WHERE" except that it goes strictly after your "group by". say you wanted to select all the locations with >1 flights landing there, you might execute a command like:

```
select destination, count(*) FROM flights group by destination having count(*) >
1;
```

Foreign Keys

SQL is called a relational database because you're able to take multiple tables and relate them or show how their data is linked in some way (such as JOIN). just as a primary key is what uniquely identifies items in a given table, a foreign key is when you relate some other table's primary key/id to identify some property amongst, like passengers on a particular flight might all have the same “foreign key” identifying which flight they’re on.

you can set up a relationship between two different tables by specifying in the second table as you’re creating it, that one field (column) has something to do with a column in another table. for example, if we made a passengers table listing all the passengers taking flights on a given day/in the airport at a given time, we might include name, age, and flight_id REFERENCES flights to tell SQL that the flight_id column of each passenger is referring to the fact that they’re on a specific flight, identified by “id” (the primary key of the flights table we’ve written has been referenced).

if you wanted to join the passengers and flights table to see what flight each passenger was on, you might use:

```
Select origin, destination, name FROM flights JOIN passengers ON flights.id =
passengers.flight_id;
```

where ON specifies the condition upon which they are both being joined. you can also use left and right join instead to say that all the elements in the left or right table (as written) must be listed, even if they don’t correspond with anything in the other table.

indexes

you can add an index, a stored data set of queries you commonly use, to improve speed of access. this is much like indexes by number on a book which make access of specific contents on a given page much easier.

CREATE INDEX NYflights ON flights (origin ‘New York’)

which would allow you to access flights that have ‘origin’ NY very easily, and if that’s a query you did very many times, this would speed up querying since you wouldn’t have to write the query that fetches all the flights from NY. however, equally, next time you add a flight from NY, you now no longer have to just update the flights table, but now also the NYflights index, which adds time (in addition to the index taking up new memory), so it’s a trade-off when to start using it (only when you use this query a lot)

you can also nest queries, such as using a long query to isolate the flight_ids from passengers table that have multiple passengers on them, but say you want to select all the passengers going on those flights, you might keep the original query, but then stick it inside brackets and add on the outside the second query, to get exactly what you want from the very get-go.

security considerations in SQL

when the user has to input some information, say a username and password, you don’t want to just blindly plug in their input into the SQL code, since they could actually type in SQL instead of the password, and gain control of the data in your database. you want to somehow “escape” or “scrub” or “sanitize” their input, and you should *never* trust the user (both because they’re often stupid, or if not, then they’re malicious)

using PostgreSQL in Flask

you can begin a “chunk” of transactions in SQL by typing BEGIN, at which point any of the ensuing code (until you execute it ALL at the same time using COMMIT) will be executed as one large chunk.

to be able to connect Python and SQL, that is, to have inbuilt python functionality to interface with databases (much like you use Flask to get inbuilt functionality to have Python interface with the web by going deep into the weeds of implementing HTTP requests and setting up servers so that you don’t have to do all of that), you use a particular python library responsible for SQL interactions, called SQLAlchemy.

an environment variable is one that’s bigger in scope than even a “global”, that is, it exists outside of the program and the OS is responsible for importing it into the program to be used income way.

there are a number of commands you have to enter to set up SQLAlchemy, including:

```
import os

from sqlalchemy import create_engine
from sqlalchemy.orm import scoped_session, session maker

engine = create_engine(os.getenv(“DATABASE_URL”)) # this “engine” is an object that interfaces with SQL
db = scoped_session(sessionmaker(bind=engine)) # ensures each database is treated as a different session

def main():
	flights = db.execute(“SELECT (origin, destination, duration) FROM flights”).fetchall()
	# the above command returns a list of dicts: [{or1, des1, dur1 }, {or2, des2, dur2 }…]
	for flight in flights:
		print(f“{flight.origin} to {flight.destination} takes {flight.duration}”)
	# note how dot notation here allows us to grab the different values because the list is a list of objects
each having different properties; and it is those that we’re accessing using dot notation
```

alternatively, you could write a program that takes in a csv file and inserts it into the database for further manipulation in the future. this would be similar to the above, but done more concretely like this:

.csv file:

```
‘New York’, ‘London’, 445
‘Paris’, ‘Rome’, 215
‘Delhi’, ‘Beijing’, 510
```

python program to import into *flights* table of database:

```
import os

from sqlalchemy import create_engine
from sqlalchemy.orm import scoped_session, session maker

engine = create_engine(os.getenv(“DATABASE_URL”)) # this “engine” is an object that interfaces with SQL
db = scoped_session(sessionmaker(bind=engine)) # ensures each database is treated as a different session

def main():

	f = open(file.csv)
	flightsData = csv.reader(f)
	for (or, des, dur) in flightsData: # shows how reader naturally puts csv in columns and rows
		db.execute(“INSERT INTO flights VALUES (:origin, :destination, :duration)”, {“origin”=or,
“destination”=des, “duration”=dur})
		print(f”{or} to {des} taking {dur} was just inserted.”)
	db.commit()
#note that the above syntax is slightly different from SQLite3, something to keep in mind

if __name__ == “__main__”
	main()
```

with Postgres, you seem to have to use passenger.name instead of passenger[‘name’] when you’re returned a dict from the database.

SQL itself will let *you* manipulate the database to get results, but if you were, say, designing a system for the public to get this information (say, an airline’s website), you’d need to implement a python back-end to take in and mainpulate data that the user inputs and then interface with SQL *for them* to give the right output.

you can even use Jinja2 with HTML when you’re defining the value = “{{ flight.value }}” a useful python functionality is “try…except” which will try a certain thing (converting a certain input that’s supposed to be an integer to int type) and *except* will be in the case of a certain failure (like except ValueError) will display an error message (you have to render that template).

there’s a SQLAlchemy feature called .rowcount() that, when appended to an SQL command via db.execute(), will return the number of rows that satisfy that SQL command.

the :placeholder syntax is powerful because it’s telling SQLalchemy to take care of SQL injections behind the scenes by cleaning the data before using it.

remember how you can create URL placeholders using url/route/<int:flight_id> to therefore loop over flight_id in Jinja and create a new URL for each flight dynamically (and use Jinja2 to populate each page with corresponding information about that flight).
an easy way to validate that a flight exists when some user inputs some parameter is to merely enter the SQL request to search for that flight_id and append .fetchone() which will return None if there are no flights with that ID.

return to review how “airplanes1” was built after watching the API lecture, since project 1 requires both API knowledge as well as dynamic python page knowledge

Lecture 4: ORMs and APIs

object oriented programming

an ‘object’ is just a thing. you can create ‘classes’ which are general versions of objects, and class definitions define the features that objects will have. a ‘flight’ class will define how a ‘flight’ will have passengers, origins and destination associated with it, as well as an id. then you might be able to go on to define flight1, flight2, and so on, as objects in this class.

you’d define and manipulate python classes as follows:

```
class Flight(self, origin, destination, duration)

Flight.counter = 0
Flight.passengers = []
        def __init__():
                self.origin = origin
                self.destination = destination
                self.duration = duration

self.id = Flight.counter
Flight.counter += 1

        def print_info():
                print(origin)
                print(destination)
                print(duration)
                print()
                print(id)
                print(“Passengers on this flight are: \n”)
                for passengers in self.passengers:
                        print(passenger.name())

        def delay(amount):
                self.duration += amount

        def add_passenger(passengerName):
                self.passengers.append(passengerName)

class passenger(self, name)
```

```
        def __init__():
                self.name = name
```

the above code shows how you can define a class (what the syntax is) as well as how you can get methods to associate with a certain class, and how different objects of different classes can interact, as well as how your object can have properties that weren't defined explicitly in the __init__ part of the class declaration.

but how do we get databases involved? ORM means object-relational mapping. We use a library called flask_SQLalchemy to get our classes and objects to correspond to databases, rows and columns, since there are lots of structural similarities between objects, with several properties, each having values, and databases, with several fields/columns, each having values.

we can write python to be able to take action on a database instead of SQL itself—now moving into yet another layer of abstraction. we do this by declaring a particular class, say Flight, that has properties corresponding to fields in a table, and the syntax for doing so is:

```
class Flight(db.Model) #all the db stuff is standard, you've just got to learn this format
        __tablename__ = "flights"
        id = db.column(db.integer, primary_key=True)
        origin = db.column(db.string, nullable=False)
        destination = db.column(db.string, nullable=False)
        duration = db.column(db.integer, nullable=False)

db.create_all() # this actually creates the tables in SQL that you declared using python above
```

if you were creating another column, passengers, and had a field there that corresponded (via JOIN ON type relationship) then you would specify an argument db.ForeignKey("flights.id") if it was in the passengers.flight_id field that you were declaring this, like in the brackets above.

when doing this in context of a flask app, there's a few things you need to do to configure things.

after app=Flask(__name__):

```
from fileWclassDeclarations import *

app.config["SQLALCHEMY_DATABASE_URI"] = os.getenv("DATABASE_URL")
app.config["SQLALCHEMY_TRACK_MODIFICATIONS"] = False

db.init_app(app)

def main():
        db.create_all #create everything we declared in the file that we imported

if __name__ == "__main__":
        with app.app_context(): # don't need to know what this does, just include it in this context
        main() # the reason we do this at all is because if we just replaced this whole thing with main() then it
would run when we imported this file into another file, and we don't want it to run unless we run it (when
imported to another file).
```

instead of "INSERT" in SQL, we might use, in python:

    db.session.add(flight) where flight is an object of the kind Flight created by doing Flight("id"=A, "origin"=B…)
    then db.session.add(flight) after creating the object

The advantage to doing this is that you no longer have to interface between two different languages, and can write only in Python. This makes development faster and easier since you require less specific language knowledge.

how do we use the "SELECT * FROM flights" equivalent? use *Flight.query.*all() (since class Flight = table flights)

and if you wanted to add filter functionality, i.e. only return flights that satisfy a certain condition:
Flight.query.filter_by(destination="New York").all(), where we use all() instead of first(), for example, which only selects one response (the first one) and returns that, and .count() which has the SQL COUNT functionality

if you wanted to extract something using SQLalchemy by ID, which is extremely common, you can go Flight.query.get(IDnumber) instead of Flight.query.filter_by(id=IDnumber), which is longer and more cumbersome.

db.session.delete(flight) allows you to do that, much like db.session.add(flight)

db.session.commit() allows you to push all the changes you'd written using all of the above commands in the text editor.

other syntax for using flask-SQLAlchemy:

Flight.query.order_by(Flight.origin).all() = SELECT * FROM flights ORDER BY origin;

```
Flight.query.filter(Flight.origin != 'Paris') #notice .filter() NOT .filter_by() since you can add bools into this format
Flight.query.filter(Flight.origin.like('%a%')).all()
Flight.query.filter(Flight.origin.in_(['Tokyo', 'Paris'])).all()
Flight.query.filter(and_(Flight.destination == 'Tokyo', Flight.duration > 500))
```

Where the SQL might be:

SELECT * FROM flights JOIN passengers ON flights.id == passenger.flight_id

the corresponding Flask-SQLAlchemy would be:

```
db.session.query(Flight, Passenger).filter(Flight.id == Passenger.flight_id).all()
```

But why would we want to use this functionality in Python as opposed to SQL? Sure it's one language instead of two but you're still learning the same amount extra vocab, and it's just as verbose. A reason this is so powerful is because now we're speaking in the language of Python objects and classes, we can start creating unique methods in those classes that abstract a lot of the SQL pain away, and we can reuse those methods again and again, greatly shortening the amount of SQL we need to add.

another powerful concept is the idea of giving different tables "relationships" in SQLalchemy. in your Flight class (your flights table in Python), you can declare one more "field":

```
duration = db.column(db.integer, not_null=True)
passengers = db.relationship("Passenger", backref="flight", lazy=True)
```

in the second line of the above code, you're not defining a formal field or column, but instead a relationship between the passengers in a Flight class, and the Passenger object defined later. the "backref" refers to the relationship being created in the opposite direction; now instead of saying Flight.query.get(p.flight_id), just as you can now say Flight.passengers to get the flight's associated passenger objects, you'll then, with backers in place, be able to say passenger.flight to get the flight they're on. the lazy value just means only find me the list of passengers associated with a flight when I ask for it to save space and operational time.

In sum total, the SQL:

SELECT * FROM flights JOIN passengers ON flight.id = passengers.flight_id WHERE passengers.name == 'Alice'

becomes the python:

Passenger.query.filter_by(name='Alice').first().flight

APIs

Application programming interfaces are protocols that dictate how any two parts of an application, or two different applications, interact and exchange information. The reason we use JSON is because it's a standard, human-readable way to pass information from one part of an application to another (or from one application to an entirely different one).

If you had a JSON object counting flight information we have above, how might you add the information about the airport codes? Well, you could add two more key value pairs, but a more efficient (standard practise) way of doing so would be to create origin and destination as JSON objects *themselves* within your flights JSON object, and within *those* add information in the form of two key-value pairs, one each being the airport code (and the other actual information about the name of the city).

in context of APIs, you can make a few different types of HTTP requests—GET, POST (usuals), PUT (to replace some data), PATCH (update some data), DELETE (delete data via the API)

The *requests* library in python allows us to write python and make these various kinds of HTTP request to different web servers that we're accessing via an API. doing:

```
response = requests.get("www.google.com")
print(response.text) # to get the raw text instead of the JSON format
```

would merely output the Google home page's HTML source code. you can similarly implement post, patch, put functions and more to different APIs, often when you want to use a certain API, the website for that API will have usage information on what method (get/post/patch etc.) to use and how to use it with what URL to access the data in JSON format via your python program.

when you get data back, you often want to do data = requests.get()… value = data.json() and use the final value object because the json() function converts the computer-readable JSON into human-parsable JSON text we can read. when writing code that uses APIs, you can often add:

```
if request.get(URL).status_code != 200:
    raise Exception("error")
```

common other status codes include: 200 OK, 201 created, 400 bad request, 403 forbidden, 404 not found, 405 not allowed, 422 unprocessable entity

The reason we love JSON for data transactions is that both humans and machines can understand it. if we want to manipulate the response in some way, to say, output meaningful text instead of just the JSON, we can just treat it as a dictionary, accessing certain parts of it and assigning them to variables we can embed into a text statement we then write.

so you can just send a request.get(URL) to the API URL, and you'll get fresh, accurate information, where the forex API will handle the getting of the currency exchange rate by scraping websites, cleaning the responses, storing the data, repeating daily, etc. This is why you don't actually need to understand what's going on inside the black box, just that it's a function with the output you're looking for.

you can also take one currency and convert to another, getting custom, user-inputted information from the API by just using placeholders and reading the API documentation on what parameters to put into the URL as placeholders, and by using variables to take in what currencies the user wants exchanged.

but how do we make our own API? jsonify() takes in a dictionary and puts in all the necessary metadata that allows the information to be sent. you'd implement functionality in python that allows someone to put in the flight ID they're looking for information on (origin, duration, etc.), you process that and make some variables that query whatever database you need and get that information, then pass it back in the form of jsonify() dictionary so they can use that however they need.

with larger APIs, you create API keys to limit the amount of requests a particular user can send to the web server so it's not overloaded computationally (which would prevent others from accessing it—this could be malicious in the from of a DDoS attack). you're given a key at the beginning, and your computer stores it to use as a "key" to get into the API (almost like an identifier so the web server sees it's your computer making a request and can monitor how many requests you're making).

we create APIs obviously because we can't grant random users access to our database, and so APIs are how they are able to *interface* with our program's database in an abstracted way.

Intro to JSON

It's a data interchange format used to send data between server and client. It's easy to read and write is why it's used—it's often used as AJAX (XML has fallen out of flavor). JSON uses key-value pairs. Double quotes around *both key and value* (and hence different from just a JS object—but numbers don't need quotes). Must fall into one of the standard JSON data types.

EG:

```
{
"name": "Tanishq Kumar"
"age": 18
"address": {
        "Street": "yeboi ave."
        "apt.": 504
        }
"siblings": ["kush", "vedha"]

}
```

You have to use JSON as a data interchange the when communicating between server and client; a normal JS object just won't work—know the difference. You can't treat a JSON as a JS object—if you did thing.value, it wouldn't work with a JSON, even if "value" was one of the keys used, whereas it *would* work if that same thing had been done to a jS object. the .stringify converts JSON to jS object and .parse() the opposite.

If you made a jS object that contained a list of different objects, each of which had a "name" and "age" attribute, and then created a <ul> with an 'id', you could then get a list to dynamically be created on the web page by:

```
document.getElementbyID('ID').innerHTML = '<li>' + list[0].name + '</li>'
```

Note that this doesn't involve any AJAX.

On the other hand, you can create a JSON file, with a similar array, called 'people within it, and then fetch that data from that file using AJAX, before outputting it onto the screen. Note that you can comment large chunks of text out using /" "/ in jS.

Project 1: Book Review Web App

- you can host a database on your own computer as a local server, but heroku gives you more space & functionality—and it's free. so they are "hosting your site" (like a smaller scale AWS)
- you log into Heroku and start a new app, creating a new database in the process
- then you go onto a GUI host like Adminer and enter the custom details you can find in the database credentials of Heroku, and you can graphically manipulate the database online.
- make sure you create a "templates" directory in your project folder when running on terminal, otherwise flask can't recognise it when running the web app

ask advith: do I need to put in the below every time I want to do flask run, and do I need to keep two terminal window hosting processes on ALL the time if I want my flask app to be always "live" like it would be if it was on a remote server through heroku.

```
// all of this in terminal after setting up Heroku app
        export FLASK_APP=application.py
        export FLASK_DEBUG=1
        export DATABASE_URL=postgres://
zsjxsjaflsmgku:c7ea1a0d0e764e7730d13d72786ee796fce595dc710a600365da4b2ba7e6c257@ec2-23-21-109-177.compute-1.amazonaws.com:5432/d7qefqc0h7pu2d
        flask run
```

- you need to sign up to Heroku and the API source to get 1) the database credentials to use with Adminer and 2) the developer key you'll need to use to make API GET requests
- you should be able to manage all deployment and updating of code using Git on CLI. With Heroku, you simply push the code to a Git repository, and then use "Heroku create" after downloading Heroku CLI & logging in. then after creating the web app space on the remote server, you do "git push heroku master" to push all your code onto the remote master server so heroku can actually deploy your code. also use this to update your code after deploying your web application. you may need to adjust buildpacks on heroku/python while deploying as you follow along the below article. Make sure you use pip3 whenever you need to use "pip" otherwise it'll use the default pip2.
- the process of deploying your code on complex architecture is called "devOps" so a devOps engineer is the one that would sift through all this shit to get the code live and connected with all the other moving parts and would know the ins and outs of things like Heroku—engineers spend a reasonable chunk of time working on deploying things like this, which is interesting.
- ngrok allows you to demo your site via a public URL linked to your local server without the overhead of full scale deployment on a remote server

https://devcenter.heroku.com/articles/getting-started-with-python

HEROKU for goodreadsclone

Host
ec2-23-21-109-177.compute-1.amazonaws.com
Database
d7qefqc0h7pu2d
User
zsjxsjaflsmgku
Port
5432
Password
c7ea1a0d0e764e7730d13d72786ee796fce595dc710a600365da4b2ba7e6c257
URI
postgres://
zsjxsjaflsmgku:c7ea1a0d0e764e7730d13d72786ee796fce595dc710a600365da4b2ba7e6c257@ec2-23-21-109-177.compute-1.amazonaws.com:5432/d7qefqc0h7pu2d
Heroku CLI
heroku pg:psql postgresql-concentric-22507 --app goodreadsclone

after applying for goodreads API:

key: wdEMbZwCMgj17LJQZ0mEiw
secret: mdHm3MTK47P7RuJOwmPj0VEaLrFL9oqW6x0O4ui1qrE

Lecture 5: JavaScript

intro to jS

Why would you want to have computation ever occur client side? Well, to take some load off of the server when it's handling large amounts of requests, and the response for the user is faster since there's no communication happening between the front-end and back end to make that behaviour happen.

you can define a function() {} in the <script> and then have it execute when something is clicked by adding the *onclick* attribute inline of the html such as <img onclick="function()">. this is an example of jS responding to an *event* — other events you can set it to respond to include onmouseover (hover), onkeydown, onkeyup, onload, onblur, and more.

document.querySelector only extracts the first element that matches that condition! so if you had DQS(h1).innerHTML = "Lol"; it would only change the *first* <h1></h1> it saw, *not* all of them (you can implement other functionality to do that).

to template a string, like format string in python, use the backpack and $ like this (for variable counter having been defined): *alert(`The count is: ${counter}`)*

you can factor out the “onclick” functionality out of the html so that it’s cleaner and more efficient at scale. you do this by adding some code at the top that listens for a click like this:

```
document.addEventListener(‘DOMContentLoaded’, function() {
        document.querySelector(button).onclick = count // notice how we’re treating count() as if it’s a
variable by just setting DOM.onclick equal to it—this will be a powerful feature later
});
```

and then we define count()… and so on. The DOMContentLoaded is a listener that only executes the function after the entire page (DOM) has been loaded.

while *let* defines a local variable (within the scope of whatever has been defined) and *var* defines a global variable that can be manipulated outside of wherever it was defined. const just sets a variable to a fixed value so it can’t be mistakenly changed later on in the program.

In addition to manipulating html, we can use jS to manipulate CSS. when a certain button with a certain selector ID is clicked, you can use document.querySelector(“#red”).style.color = ‘red’; and access the CSS’s colour property that way. but how do we abstract/generalise this if there are lots of buttons that need to be able to be converted to lots of different colours? That might take place in this sort of code:

```
document.EventListener(“DOMContentLoad, function {
        document.querySelectorAll(‘.color-change’).forEach(function(button) {
                button.onclick() {
                        document.querySelector(‘#hello’).style.color = button.dataset.color
                };
        });
});
// where you’ve defined data-color: blue or whatever for each button you’ve assigned
```

in newer versions of jS, you can summarise these functions into shorthand called arrow notation, like this:

function() { } becomes ()=>{} and function(x) becomes x=>{}

the value “this.whatever” refers to the stem of the event change that caused the function change to be assigned to this.whatever. for example, if you had a drop down with a list of colors, when the state of that drop down changes (another option is selected), you can give the text above the dropdown “this” value—that is, the new value in the drop down selected. note that you need to use conventional function() notation and not arrow notation when you want this to happen.

when building something like a to-do-list, you’re creating a to-do list that’s added to when a button corresponding to a text field is submitted. you’d define a form with ID ‘new-task’ and a text field within that called “task”—all of this in the HTML. And then use li = doc.createElement(‘li’) to get a (const) into which you need to put the task you’ve obtained from the text field. then set li.innerHTML = task.value then append(li) to the end of the ul you created in the beginning (id taskList). then reset the text field’s (by grabbing using ID).value() to ‘’ so they can repeat. and finally, returning false will stop the form from submitting to a server (because no server is defined) and the change will just dynamically take place on the page.

what if you wanted to edit this so that you can’t submit a blank response? then you’d by default set the button.disabled = true; then after that check for an .onkeyup on the text field, and when there is, you can set set button.disabled=false; for the submit button.

setInterval(function, interval) executes a function every X seconds

how do you have data stored between sessions, even when the browser is closed, without using a database or communicating with a back-end? use local storage, a browser functionality that most modern web browsers offer. you use it via localstorage.getItem(‘name’) OR localstorage.setItem(‘name’, value)

AJAX: more formal introduction—without using $.ajax (jQuery) since vanilla jS is powerful enough.

when you make variable = response of an API call, you can do variable.status_code to check if it went through, and assign a safety error route in case it didn't (so the app doesn't crash if that API call fails). note that you still have to take variable.json() to be able to operate on what you were returned.

you have to give the Api request certain "params" depending on how the documentation wants you to send the request to the server. to see the full implementation of vanilla AJAX, refer to the code in the lecture.

web sockets

socket IO is the name of a jS library that allows for full duplex communication between two clients. to make use of this, you'd need to put at the top of your app.py: socketio = SocketIO(app). this basically is made use of in a jS file linked to the app.py that has a route that checks for receipt and then broadcasts an event every time something is done on the page—such as a message sent—allowing other clients to "catch" that event and manipulate it. then you can go back to the jS and "catch" the message/vote and display it on some page via standard DOM manipulation. this is how you'd build a chatbot—by "emitting" messages individuals write.

Project 2: Chatroom Web App

- terminal notes- use "wget" to download a file from a given url, and "cat" to output the text contents of a file, and "nano" to get the text editor up in the CLI. for many projects, you'll need external python packages like socketIO (for chatroom functionality) and flask, and all of these are listed in a file that's conventionally called "requirements.txt", and then you just do "pip3 install -r requirements.txt" to get pip (the python package manager) to read the file and install whatever is inside it

- socketIO is fundamentally a JS library. socketIO abstracts ajax away so might just need to implement that with the Jinja and not any manual ajax itself.

- debug from first principles. I was incredibly frustrated by the socket functionality in the jS not working, not realising that it was because my damn file wasn't being loaded in since Flask looks for the static file instead the operating directory (much like I had to learn to create templates for Flask to get at my html files). If my console.log at the very top wasn't being printed in dev tools, the file very clearly wasn't being loaded correctly. From that assumption, I could've worked out where the problem was—it really isn't that hard.
- the "room" functionality takes care of partitioning content by channel, you just need to pass in the right room to flask-socketIO. and session functionality in flask is powerful because you can create any custom key that will be valid only for the user that is logged in on their computer—user ID, channel name, and more.
- window.location.replace is useful to redirect in jS as opposed to redirect() in python

Lecture 6: Front-Ends

single page apps

- you can use ajax-like functionality to compress the content of a multi-page app into one where everything is part of a unified layout and only specific things are changed; this is how you create an endless newsfeed on things like Quora or Facebook, and other sites like Google Docs use it to edit stuff on your screen in real time without changing the URL of what you're at (that is, without reloading).
- you'd want this because that makes the 'reloading' (introudction of content) both faster and more visually seamless for the user. for simple apps with just a few different pages, it might not make sense putting the overhead infrastructure in place, but for large, scalable apps, this simple change made by adding some javascript to get some info from the backend about the content you want to load and then inserting it into the innerHTML of whatever you're trying to load it makes a big difference to responsiveness and UX, which is why monsters like FB and Google do it.
- you can keep the URL changing to guide the user as to what page they're on using this pseudo-ajax function using the HTML5 history API, which changes the state of the URL when swapping the content on the page (via jS).

- you can think of the different pages as elements in a stack, where you can pop and push states to get there browser to remember which URL goes with which set of contents on the page when you're using the html5 history API
- you can access the built in *window* and *document* objects, the first of which tells you about important screen dimensions, including how far the user has currently scrolled, which is how you can implement endless news feeds, by using jS to load more content every time you clock that the user has scrolled to the bottom of their feed.
- just as you can use the templating language Jinja to dynamically insert things that you have on the back end onto the front end, you have a similar language built for jS itself, which you can use to template things on html pages when you don't want to get the backend involved (or, indeed, you don't have one) and it's called Handlebars.
- you can manipulate CSS animations to create beautiful UXs using jS. For example, you can manipulate how a button influences the playback of a certain named CSS animation to make an element on a social media feed fade away or slide away, or things of the like.
- just as how you can create div's which are rectangular containers, there's no reason you shouldn't be able to create arbitrary containers of other dimensions or shapes. D3 is a jS library mainly used for data visualisation that makes manipulation of vector graphics (arbitrary shapes) easy. you can see how this would be useful when you're pushing data about proportions of, say, genders in a population as fractions, and you want to output shapes that correspond to those.
- extending that, you can use your own jS to be able to create transitions between those shapes, going from one to another, and you can define buttons and functions that allow you to change the colours and animations of these svg objects on command (like creating a traffic light image that is interactive).
- you can also create interactive applications that are very visual like a drawing application where a user can drag to draw an image on the screen. you'd do this by using d3 to write a jS function that, when a user clicks the mouse down, turns a bool true, and that bool activates a function that will insert a small black circle (svg graphic) constantly (at given time intervals determined by the browser). you can extend this by taking width and colour as inputs, and rejigging the draw_point() function in jS to be able to connect two points by taking the x and y of both the current point, and the last point, and then using innate d3 functionality to connect them (which fixes the issue of discrete points being visible if you draw too fast using the previous functionality).

an aside: the UX is the abstract feeling that a person comes away with having just used a product, and that encompasses, and is informed/determined by UI, which is the mechanical implementation of how the user interfaces with the product. more concretely put, the UI of a site is the actual layout, buttons and way the user touches the functionality, and that informs the UX, but the UX is also informed by the content on the page, the branding around the product, and more. A beautiful UI is a powerful tool in creating a memorable and delightful UX.

Lecture 7: Django

Django separates things into different projects, each containing one or more applications which can interface and execute certain functions to build out the overall functionality that you're looking for. It automatically creates a bunch of files that have some innate functionality for you to use from the get go, and this contributes to the "overhead" that you have to get used to when starting to use it, more powerful though it may be. These sub-files immediately created are called "project components"

after you've initialised all the django files by doing "django-admin startproject name", you need to initialise an actual app within that project via "python manage.py startapp name". at this point, you'll have a directory called "name" inside your project, and when you navigate into it, a bunch of files set up for you, including migrations (for ORM), and models/tests, amongst others. Here, you have views.py (already created), which is the equivalent of app.py when making a flask application (that is, where you write you main back-end code).

you'll have an extra urls.py file for the application as well as one for the project as a whole. this is so that you can create the equivalent of app.routes; that is, you can tell django you want to associate a certain route/path with a certain view (or, more specifically, a certain function inside views.py). there, you can write:

```
        from django.urls import path
from . import views

urlpatterns = [
    path("", views.index)
]
```

and then link the *project's* URLs.py with that of the application. this is because each app within a project will have its different routes, and so this project urls.py is where you get routed to the correct app in the first place when hitting an endpoint. something you have to be very careful about is the naming and placing of files with django. django looks for specific filenames by default, and if your views is in the wrong directory (of which there are may layers), you'll run into an error—so you have to be very systematic about these things.

you can do the ORM equivalent of Flask-SQLAlchemy in the models.py of your django application. You'd create a SQL database by writing:

```
class Flight(models.Model):
    origin = models.CharField(max_length=64)
    destination = models.CharField(max_length=64)
    duration = models.IntegerField()
```

and then going to CLI and migrating over the changes to the file (make sure your project knows about the models via the settings file by adding *'flights.apps.FlightsConfig'*). This is done by python manage.py makemigrations which then gets ready to update the database. To actually make the change, just do *"python3 manage.py migrate".*

you can *def __str__(self):* to define what the string representation of a certain object looks like when displayed in a shell or on an html template (when otherwise you might be returned "query Object 2", which isn't very informative).

it is convention to pass information into the front end (eg an html template) as a variable (dictionary) called context, which is passed in as a final argument in the render() function, of which you should make sure not to forget including the request.

the process of 'namespacing' involves adding the directory name before a file—so /flights/x.html instead of / x.html because you'd have multiple apps in one large templates file then and you could have multiple x.htmls for each app—so make sure you put x inside its app 'namespace', in this case, '/flights'

managing databases and constructing functionality around those in Flask was non-trivial, where you had to thin through the data processing and build web pages carefully around it, maybe with a little help from Adminer, a GUI. Similarly, but more powerfully, you have admin on Django. Just import then admin.site.register(className) to have it up on the interface. you can even customise the interface to be able to make certain addition operations easier and more intuitive by changing the settings/adding code to admin.py, where previously we only registered certain classes with the admin interface.

you have the *try: except*: clauses where you can do something and have validation readymade incase something doesn't work (where the error follows the except clause, by doing something like *raise*).

Django is so powerful because it abstracts so much away, so you can create much more complex and functional applications with the same lines of code compared to Flask. An example of this is in how it handle database manipulation—you create classes in your models.py and then just migrate them, specifying as an argument in the class properties how you want the classes to be related (eg, have flights be related to passengers and vice versa so you can access passengers on a flight as well as flights for a passenger) and it will understand that, and create 2 tables under the hood to keep track of the many to many relationship, so you can simply do f.passengers() to get at what you're looking for in the future. With flask, you would've had to create those tables after thinking about which to create and with what columns and how, and spent a lot more time "in the weeds".

errors will always have the stupidest origins. for example, the reason I was having so much database trouble was because sqlite wasn't told to cascade the deletion of things in one table to things in another table because my on_delete argument of the class declaration was in quotes when it shouldn't have been. this is why, when you're constructing your code in the first place, you have to be super careful from the get-go to avoid hours spent painfully debugging afterwards.

the “return” function breaks out of the current code set you’re in—so in the case of a django app it breaks out of the current function/route you’re operating in and moves onwards if it’s triggered

Project 2: Online Menu for Pizza Businesses

-

Lecture 8: Testing, CI/CD

- you should be automating the process to ensure that every method you call is airtight. doing 1 or 2 tests to get a sense isn’t very comprehensive—it is far more likely than you think that an error just manages to slip through without you noticing, which could leave you debugging the entire web app for hours, scratching your head. test each function in isolation, and the way you do it isn’t by typing in inputs and comparing them to expected outputs, it’s by writing a function that will take in the input and expected output and compare them, then writing a bash script (a bunch of lines of raw python as you’d input into the shell) and creating tests by hand (writing lines of shell code that would physically test a specific value), making sure to include a wide variety of edge, corner cases, like the good QA engineer you are. then just run the test script overtime you make a change to the method to make sure it’s not breaking, instead of testing things yourself each time.
- the *assert* function simply checks whether something is true. if the statements following the assert is true, shell says nothing, and if not, it’ll give you an error. you can check the error code of running a program by typing *echo $?* and 0 is the only code that means that everything went alright.
- unit testing is testing individual methods, or units, at the most granular level. above that is integration testing, where you start to test multiple functions working correctly together, in, say, one app in a project, and then you have systems testing, where you see if multiple systems (the different apps) work together correctly as intended.
- python comes in with an inbuilt framework to help you with unit testing. you import unitTest atop the file, and then define an entire class whose sole purpose it is to test. then build in each individual test case as a new function inside of that class by doing self.assertTrue(testThatShouldBeTrue)
- tests.py is the django file that is built in to the framework to specifically test your back-end’s integrity. in it, you have to define a setup function, in which you set up certain objects (via a = Flight.object.create(specification)) and then you have several test functions, each of which makes a certain self.assertTrue(function that should return true if that function is working correctly) and then Django makes a test database in which to play around with this and run all these tests so that you know everything is working correctly.
- in the same way that you test the backend, you can test the pages on the frontend to make sure all of them are loading correctly (i.e. giving a 200 response back to the server), you do x = Client() and then response = x.get(url).response() and then do assertion tests on response.status_code as well as more nitty gritty things about what you expect back (if you expect the query to return 2 objects, you count the amount in the querySet that is the reponse, and do assertEqual to check it’s 2). this ensures data is being passed correctly through model-view-template paradigm that django operates under.
- just run python3 manage.py test to put your pages through their paces. this sort of thing is both commonplace and necessary at any medium to large scale company, because if AirBnB or Stripe screws up a payment, or Google serves someone misleading information, they could genuinely do horrific harm to may people.
- selenium is a software that acts as the glue between python and UIs written in things like jS. you can’t really use django’s testing functionality to run tests on the nitty gritty of the javascript files operating on your front end (how do you get at individual tags and elements using python django?) and so you use selenium, which basically lets you take control of your internet browser using python. so you can access files locally, select things on a web browser page using python, and more. it’s basically the da Vinci surgery machine for a pythonista toying with javascript. and so you use it to run tests on the UI to ensure that a certain element has a certain innerHTML at a given time, or do other similar things. still do this using assertTrue or things of the like.
- CI refers to the practise of, whenever you push production code to a branch (main or not), it could fail itself or cause some other part of the app to break (since developers working on a given feature can’t simultaneously think about how it’ll interface exactly with every other part of the app). A CI too llike Travis returns the verdict of running these automated tests (that you set up in some tests.py doc using Django or Selenium) to GitHub, which then tells you. CD is then pushing these verified features into the actual product so users can start using it.

- a “web hook” is just software that runs when something happens. In the case of GitHub, it sees when a user pushes code, and then notifies travis (they communicate through the travis API, exchanging data) which then runs tests on the code. but to do so, make sure to include the .yml file (a Json-like data type containing metadata on the type of language used to write the code you’re shipping and want to run tests on)
- how you make sure another developer has all the dependencies required to run your application seamlessly? you *could* run a virtual machine on their computer that is configured to have all the required functionality (simulating an operating system) but that’s much slower than an actual computer by virtue of the fact that it’s a computer running on a computer.
- instead, use docker. docker was made for code/applications to be shared and their functionality discussed by different developers. so, it is a developer tool. instead of taking all the overhead of a virtual machine on the other developer’s computer (which runs painfully slowly) it instead takes a different approach and runs the app on top of their OS itself (as opposed to the virtual OS) on top of the docker platform. it essentially contains not just the code, but all the information about the different services used to build the code (what libraries/frameworks, what versions) as well as all the settings/databases the code came with, and so it creates a chamber in which it executes it exactly how it would execute it on your computer (even if they don’t have those dependencies on their own computer/have different versions), making comparing software that much easier. you just have to make sure to match the docker IP matches your own localhost IP. all the dependencies etc. are stored in a yml Dockerfile.
- therefore, different engineers working on the same project would all share a docker and so instead of running *python3 manage.py run server* which runs everything based on the frameworks/settings/files/configurations you have going on your computer, it creates a vacuum chamber in which it simulates the sender’s frameworks/settings/files/configurations on your computer so you get the same effect as them.
- I’ve actually been using this under the hood this whole time, as when i deploy something to heroku/AWS, how do I know their servers have the same settings/dependencies as mine? I don’t, but docker takes care of (abstracts it away) that for me, which is why it takes a lot of pain out of software development.

Lecture 9: GitHub, Travis CI

- here, a GitHub employee shows us their workflow—exactly what steps they’d go through at work to make a tweak to a feature. they start by forking the repo they want to modify, cloning it onto their computer, spinning up a server using docker.
- “open source” means what it says on the tin—that the source code; that is, the code I write, is open for everyone to see, use, modify, tweak, extend, improve, talk about, and more, all in its entirety. And so there are full-time employees that oversee improvements suggested by members of the public who actually go on to make changes that are implemented on GitHub classroom (the web app the employee is demonstrating their workflow on).
- you can contrast things like desktop software which has to be downloaded to be used (think Photoshop) and then software which can be accessed from a client (that doesn’t download the software) like a web app, since you can update the server and therefore all clients’ apps that are feeding off that server, but a downloaded app then lives on the client’s computer and you can’t use continuous deployment since the user’s server is separate from the server to which you’re pushing code, and then you have to version it off —like LoL has different patches I have to wait to download when it’s used, or different softwares have versions 1, 2, etc.
- on the latter method of CD, the employee actually does just flip between branches doing work on features until they’re happy with the features they want to implement, then going to code review, where other engineers look over your code to ensure 1) efficacy (is there a glaring flaw you haven’t spotted?? 2) efficiency (is there a faster or less memory intensive way to do it?) 3) design (is the approach chosen maintainable and understandable?) after which unit tests are run to make sure it works as intended, and then it’s pushed to production. despite this, some bugs do inevitably squeeze out into the production environment, and can be heavily exploited by malicious users (and are—never trust the user!)
- the actual Travis builds when deploying in a large organisation can take a few minutes—yes, that long-- because there are hundreds if not thousands of meticulously thought out tests that really poke and prod every part of the application based on the new changes—and it’s running in a container (not on the actual computer), which slows it down a bit.
- a lot of deployment to things like heroku/AWS are already automated—this employee just types something into a slack chat about deploying a certain branch, and a bot will run a deployment script for him.
- developers shipping big features don’t want to ship a huge feature at once for everyone since they want to make sure it really does work. so they enable it for a given % of users to test it on (by writing if_enabled into the html and then getting the developer infra to alot the enabled feature for every 10th user by taking

all the IDs and simply turning it on for ID%10 ==0 or something like that) and that way they can almost beta test it before full release by rolling it out piece by piece.
- a pull-request is just a call for feedback on code you're planning on pushing to production
- continuous delivery is pushing code that's prepped for deployment (just needs to be actually inserted into the heroic pipeline) and continuous deployment is actually deploying it very often.

Lecture 10: Scalability

- a server is a finite machine that can only handle so many HTTP requests from clients at a time, and you should figure this out before you start handling requests for real by "benchmarking" the maximum load a server can take so you don't hit a wall and have your site crash when serving users.
- to get around this problem, you can scale vertically (improve the Hz capacity/performance of one server), or, alternatively, horizontal scaling, whereby you simply add more servers. but this adds additional challenges; the servers have to interface with a database, and now you potentially have two servers making requests to the same database at the same time. and when a client sends a request, how do you know which server to send it to and when? this is how scaling can add dimensions of complexity. you can solve the problem of where to route the server by using a load balancer (reminiscent of 'load shedding' when routing energy around the grid at times of peak demand). there is a whole field of study *within computer networking* that examines the efficiency and efficacy of these routing algorithms in performant and resource-constrained systems.
- since choosing different load balancing algorithms has trade-offs (if you choose the algo that routes depending on number of people the balancer now needs computational power itself vs just randomly assigning) and these are part of the design decisions of the particular system.
- another challenge with load-balancing is session-awareness. if a client sends a request and makes, say, a shopping decision on one server, you want it to be re-routed back to *that* server, since that server contains the shopping information (previous info) you need to serve the client, and so your load balancer needs to be session-aware for users.
- cloud providers can provide *autoscaling* services whereby you only pay for the amount of servers you use, which the cloud's load balancers adjusts automatically depending on traffic to your website. the cloud takes care of all the load balancing implementation details so you can just worry about solving the problem your company solves.
- what happens if a server goes down? the balancer should know not to route data to that server now by receiving "heartbeats", that is, specified signals from each server at a given time interval (eg every 200ms) to verify the server is still functional so it can continue routing data to it.
- there are also important security concerns that come with these thoughts; if you store cookies on the client so that you don't have to store session data on the server, what if the cookie is intercepted and used by someone else to log into that client's account? you have to start using cryptographic algorithms like checksum to ensure that the users really are who they are…
- analogously, as you think about scaling servers, you need to think about scaling the databases that the servers hit. you want to avoid having "single points of failure" in your network and so try to have multiple of everything (LBs, servers, dbs) and remember that as you scale, having more rows in your db means that access and manipulation becomes slower, potentially in a non-linear manner depending on what sorting and searching algorithms that database is using. this is why time complexity becomes very important at scale—imagine if Facebook took even 10 seconds to find something you were looking for…you'd very quickly stop using it. And that's 10 seconds!
- a technique used here is database partitioning—that is, splitting data up to make access easier or to reduce the load on a given database. vertical partitioning is splitting the columns of a table up such that they're located on a different table, and you can JOIN them if you ever need to communicate the data. alternatively, split the rows up so that you simply put some of the rows on a different table (like domestic vs int'l flights), but each of these, while reducing the load on the one database since access to any one table isn't as high, presents trade-offs—now certain other queries will be slower or more strenuous, like querying for all flights out of JFK means you have to query *two* tables instead of one and then JOIN the information—which means things take longer now, and all of this is systems design decisions you'd have to make.
- there are two main approaches to make databases more reliable. you can have single-primary replication, where you have multiple databases but only one can be written to, and so you can have the read/access benefit of distributing load across three (and for something like the NYT where reads>>writes this makes sense) and then the written database just updates the other reads at some frequency. Alternatively, you have can multi-primary replication, where you can read/write to all three, and then they calibrate with each other at some frequency, and while this is more efficient (time-wise for the user) it also introduces race conditions and is more computationally expensive. Again, design decisions.

- caching is something a browser does to save time so it doesn't have to query a whole database overtime a certain request is made (like for a homepage of a site, which is unlikely to change in between success requests, usually on the order of a few seconds apart) so it just stores the cache on the client's browser so next time they make a request, it's faster and computationally less expensive for the server to deliver the material. this can be annoying, like when jS doesn't update on chrome when you make changes to your file when you're developing, but is generally a powerful time-saver.
- what if you set the cache expiration to 1 day, then should the client side go back to the server and redownload after 1 day? is that the most efficient thing to do? not necessarily, especially with big files. and so the client often sends the data it currently has to the server, and the server compares it to the newest data, and returns a code telling it whether on not it needs to update the data it has.
- you can also cache on the server side, where if there's a particular computation that is asked for often (like 10 most popular books on amazon), instead of executing it again you can just pull from the server-side cache (often a literal hard disk in the data centre) and serve it up, renewing the data every X interval of time. you can also have cache *inside* a given server for data *that particular server* would need to repeatedly access, and real world cloud computing systems often use a complicated, specific mix of both to optimise their computing services for efficiency and cost.

Lecture 11: Security

open source software
- while you're giving a lot of people access to your code and they could certainly find bugs you didn't find yourself, and tell you so you could fix them, it also means a lot more people can make malicious attacks—and stronger malicious attacks, since they now have access to your source code and know the security protocols you've put into place and they have complete transparency into your software.
- two-factor authentication is becoming more popular as hacking techniques to get at a login password are becoming more successful and conniving.
- if you accidentally put a password/login credential on GitHub (as part of an admin or settings file—this extends to things like secret API keys also) then you should change them because GitHub allows for versioning and so even if you changed the version you used, it'd still be compromised.

html
- you could do things like make a page that looks like another (copy GS source code) and get credentials that way by misleading the user, or by conning the user into clicking on a particular link because the text says something that it doesn't link to (and once you've clicked on a virus link or trojan horse, it downloads onto your computer and you're effectively fucked)

flask
- it's a framework that abstracts away having to write nitty gritty http requests, and does all the web functionality built-in so you just write python logic. you can use two types of encryption to exchange data—private key encryption, where you have a key (private) used to encrypt and decrypt data, and both computers have access to that key. the problem is—how do you get that key from one to the other?
- a more sophisticated approach is public-key encryption, where you separate out the encryption and decryption functionality. this means the receiver can send the public key over the internet, the sender can use it to encrypt the message, and the receiver will keep the private key to decrypt it. the problem with this is even though you've separated functionality, since both keys are mathematical functions, the question is can a hacker see the public key and infer what the private key does? the answer is that's really difficult to do.

environment variables
- an environment variable is a variable whose value is set outside the program. an example of this being done is in the assignment of the "Secret key" for use of an API in a program; instead of simply writing out the key, which would be exposed when the code is pushed to github, you use an environment variable to reference the file in which the key is written (which you then store locally) and then the code in GitHub just says "os.environ.get('key')" which means data is then secure.

databases
- a database doesn't store passwords, but their hashes. hashes are deterministic, but non-reversible, and so even the server doesn't know what your password is—they just take the one you input, hash it, and compare the output to the table (like I did in the actual cs50 course building out cs50 finance's login).

APIs

- often times, you don’t want any random person to use your api, so having an endpoint called /api may not be the best call. for example, many businesses’ main service is literally to grand access to an api, and to ensure a client really is a legit person who paid for it, you probably want to have an api key as an authentication mechanism. similarly, this can be used for rate limiting so they don’t overload your servers with API requests. a key can help with that by simple keeping a table that maintains how many requests received per key.

javascript

- a major security threat is from the SQL injection equivalent of jS—XSS, cross site scripting, when you have your backend return some “string” + variable.that.the.user.controls.input.of, and then they type in javascript into that variable (for example into the url) and then the code is executed, where they can access DOM elements and get the values of sensitive passwords and things like that, or redirect the user (window.location.replace sort of thing) as well. this type of threat accounted for the majority of cybersecurity attacks in the past. this is a threat whenever you handle GET and POST requests from a user-controlled field.
- note that frameworks like flask/django and chrome’s browser engine has some (little, primitive) safety/escaping mechanisms in place to prevent injections of this sort.

django

- remember us having to user the {% csrf_token %} in django? why did we need to use that/what did it do? it stands for “cross site request forgery token”. say you had a bank that transferred money by implementing a get request to a certain url that passed in args of who to transfer how much from which account. by including this token in the code, you make it so that you don’t just need a GET/POST request to make the transfer, but a unique token as well that only the administrator has control over distributing.

more generally, you also want to make sure you’re always aware, as a user, of the third party apps that may have access to certain apps of yours that contain sensitive information—for example, Travis CI has access to all your github code, so any compromise there could mean any historic comprimises that crept into any code you pushed to GitHub are going to reach the hacker.

a DoS is just one computer sending requests to a server(s) and DDoS is multiple computers doing so (perhaps co-ordinated by a bot script). there are ways to fight this as a server by analysing incoming requests to verify legitimacy, but that’s hard to always get right, and so very often this just comes down to a battle of resources—do the servers or the bots have more resources, which is why robust and scalable infrastructure is important when scaling an app.