13-Web-Scraping-and-Document-Databases

?s

For ATOM…

Autocomplete…added some options.

Info when hovering (not possible – just use VSC)

Open in terminal…added.

Test scraping inside jupyter nb

After it is working, file.py, then run through terminal

HW is run thru jupyter nb

Robo 3T is a GUI for Mongo

9/29/18 – 1st day – Manuel

Overview

In this class, students will be introduced to the concept of the NoSQL database with MongoDB. By the end of the day, students should be able to perform basic CRUD operations with MongoDB and the Pymongo library.

* install MongoDB on their machines.
* understand how to make queries with MongoDB. Meeting this goal will build the necessary foundation for the next lecture which will integrate such queries with web-scraping.

Class Objectives

* create and connect to local MongoDB databases
* create, read, update, and delete MongoDB documents using the Mongo Shell
* create simple Python applications that connect to and modify MongoDB databases using the PyMongo library

**MongoDB is a very popular noSQL Database**

**It uses a document-oriented model** as opposed to a table-based relational model (SQL)

A NoSQL database is simply a non-relational database.

In other words, NoSQL databases do not employ SQL relational model when storing data.

MongoDB stores data in **BSON Format** (Binary JSON - effectively compressed JSONs)

working with BSON documents is essentially identical to working with JSON.

MongoDB has tons of **drivers and packages** for connecting to Node, C++, Java, etc.

The key differences between SQL and NoSQL databases can be seen in how related data points are stored in each.

* In SQL databases, relating data between tables requires the developer to join the rows of one with the rows of another.
  + ***SQL relies on Joins to combine relevant data***
* BSON data, on the other hand, do not require much in the way of joins because they can store objects within objects. This allows developers to save nested data directly and eliminates the need to model data relationally.
* With NoSQL, once data is added to the database, it is a cinch to traverse. Simply navigate through the data in the same manner one would JSON data.
* ***noSQL Databases on the other hand are effectively JSONs.***
* ***They excel at heterogeneous data formats and are easy to implement.***

***Terms are slightly different in the noSQL context.***

SQL vs. NoSQL

Database = database

Table = collection

Row = document

Column = field

Databases contain collections. Collections contain documents. Documents contain fields. Fields store data.

MongoDB is still an inherently different style of data storage than MySQL.

A BSON document is basically a more flexible form of JSON with individual documents capable of containing strings, ints, booleans, arrays, and even other objects.

CamelCase (in Mongo) vs. python’s snake\_case

While working within the mongo shell is fine and dandy, life would be far simpler if

there were an application to view/modify Mongo databases. Thankfully there is in

[MongoDB Compass] (<https://www.mongodb.com/products/compass>).

<https://robomongo.org/download>

got **Robo 3T** for a GUI for MongoDB

<https://github.com/the-Coding-Boot-Camp-at-UT/UTAUS201807DATA2/blob/master/class-content/13-Web-Scraping-and-Document-Databases/1/Supplemental/Installing-MongoDB.md>

Research the tradeoffs (pluses and minuses) of using a NoSQL vs a SQL database

**SQL vs NoSQL:**

Don’t believe there’s one database “to rule them all”, instead, it’s important to think about the tradeoffs and some mixture of the two are being used more and more in industry.

SQL or Structured Query Language just refers to the language used to access and modify data.

NoSQL refers to any other implementation of a database that does not use the SQL syntax to access and modify data.

When considering either SQL or NoSQL in your next project be sure to consider the tradeoffs. Between MySQL and MongoDB some of the tradeoffs are:

\* Schema - In the construction of the table in MySQL the columns are their datatypes must be defined. In MongoDB a collection can be created without any regard for the column names or datatype

\* Data to be stored - MongoDB allows for nested data structures, e.g, {“col1": 123, “col2”: {“a”: 2, “b”: [1,2,3]}}. MySQL can store data of this sort by converting the value of “col2" to a JSON and saving it with a TEXT datatype but you can’t index any of the fields. In MongoDB you can save a JSON as a document and you can index any field no matter the level of nesting.

\* Web Scale - a database that responds to many concurrent reads/writes as quickly as possible. MongoDB achieves this by not telling the writer whether the data was saved to disk. MongoDB uses “eventual consistency” so the data to be written is queued when the server is under high load but the writer doesn’t know that and will only see an error if there’s a high level issue (connection lost, query wrong) but if there’s an issue with the database (power failures, resets) then you lose data without knowing it. This may be unacceptable in some industries, though, MongoDB 4 can be configured to a mode that only responds after data is written to disk.

The SQL vs NoSQL debate can be a heated one.

Know the tradeoffs and you can avoid looking foolish. Check the link below for an example (warning...must enjoy programming humor and know what /dev/null is before watching)

<https://www.youtube.com/watch?v=b2F-DItXtZs>

/dev/null - redirects the command standard output to the null device, which is a special device which discards the information written to it

advantages of using a noSQL database like MongoDB according to the MongoDB Website?

* "Relational databases require that schemas be defined before you can add data.
* For example, you might want to store data about your customers such as phone numbers, first and last name, address, city and state – a SQL database needs to know what you are storing in advance."
* "Object-oriented programming that is easy to use and flexible."

advantages of using a noSQL database like MongoDB according to the web?  <https://stackoverflow.com/questions/2117372/what-are-the-advantages-of-using-a-schema-free-database-like-mongodb-compared-to>

* Deep query-ability. MongoDB supports dynamic queries on documents using a document-based query language that's nearly as powerful as SQL.
* No schema migrations. Since MongoDB is schema-free, your code defines your schema.

disadvantages of using a NoSQL database like MongoDB according to the web?

<https://stackoverflow.com/questions/2117372/what-are-the-advantages-of-using-a-schema-free-database-like-mongodb-compared-to>

Sometimes, using joins and having strict schemas is actually preferable to MongoDB.

"If your database has a lot of relations and normalization, it might make little sense to use something like MongoDB. It's all about finding the right tool for the job."

1.1-Ins\_MongoBasics

 start up MongoDB by typing mongod into terminal/bash windows.

While `mongod` is running, open up another terminal/bash window and run `mongo` to start

up the mongo shell.

**On Macs, no need for mongod. Just mongo**

Mongo db server is always running in the background

The shell is in the terminal.

Robo 3T is a GUI vs. MongoDB compass…also a GUI

**# Query 1 - Creating dbs, inserting data and finding data**

\* Start up a new database by switching to it. The db does not exist until you create a collection.

```

use travel\_db

```

\* Show the current db by running db.

```

db

```

\* Show current databases in existence

```

show dbs

```

\* Create a collection

```

db.createCollection("destinations")

```

\* See all collections in a database

```

show collections

```

\* Insert data into the travel\_db database with this command.

- NOTE: This will also create the collection automatically, the contents of the insert are basically a JavaScript object, and include an array.

```

db.destinations.insert({"continent": "Africa", "country": "Morocco",

"major\_cities": ["Casablanca", "Fez", "Marrakech"]})

```

\* As a class, come up with 3-5 more countries and insert them into the db using the same syntax as above.

```

db.destinations.insert({"continent": "Europe", "country": "France",

"major\_cities": ["Paris", "Marseille", "Bordeaux"]})

db.destinations.insert({"continent": "North America", "country": "USA",

"major\_cities": ["Washington DC", "New York City", "San Francisco"]})

```

\* Observe where the data was entered in the MongoDB instance (in mongod)

\* Find all data in a Collection with `db.[COLLECTION\_NAME].find()`

- The MongoDB \\_id was created automatically.

- This id is specific for each doc in the collection.

```

db.destinations.find()

```

\* Adding .pretty() makes the data more readable.

```

db.destinations.find().pretty()

```

\* Find specific data by matching a field.

```

db.destinations.find({"continent": "Africa"})

db.destinations.find({"country": "Morocco"})

```

\* Try a few queries with the examples we came up with as a class.

- Also, pick something that will find more than one entry so we can see how it will return all matches.

- Find specific data by matching an \\_id.

```

db.destinations.find({\_id: ObjectId("<ID Number Here>")})

db.destinations.find({\_id: ObjectId("5416fe1d94bcf86cd785439036")})

```

1.2-Stu\_MongoClass

**# Mongo Class**

**## A. Use the command line to create a classDB database.**

\* Insert entries for yourself and the people in your row in a classroom collection.

Each document should have:

1. A field of name with the person's name.

2. A field of the person's favorite Python library, e.g. pandas.

3. A field of a list of the person's hobbies .

**## Example:**

```

use classDB

db.classroom.insert({name: 'Mariah', age: 23, favorite\_python\_library: 'Seaborn', hobbies: ['Coding', 'Reading', 'Running']})

db.classroom.insert({name: 'Ricky', age: 34, favorite\_python\_library: 'Matplotlib', hobbies: ['Not Coding', 'Not Reading', 'Not Running', 'Guitar']})

db.classroom.insert({name: 'Srikanth', age: 28, favorite\_python\_library: 'Pandas', hobbies: ['Netflix', 'Guitar', 'Traveling']})

```

**## B. Use find commands to get:**

1. A list of everyone of a certain age.

```

db.classroom.find({age: 23}).pretty()

```

2. An entry for a single person.

```

db.classroom.find({name: 'Ricky'}).pretty()

```

1.3-Ins\_CrudMongo

**# Update, Delete and Drop in MongoDB**

\* Use the travel\_db

```shell

db

use travel\_db

```

\* Insert two countries in Africa

```shell

db.destinations.insert({'country': 'Egypt', 'continent': 'Africa', major\_cities: ['Cairo', 'Luxor']})

db.destinations.insert({'country': 'Nigeria', 'continent': 'Africa', major\_cities: ['Lagos', 'Kano']})

```

\* Show how to update data using `db.[COLLECTION\_NAME].update()`

```shell

db.destinations.update({"country": "Egypt"}, {$set: {"continent": "Antarctica"}})

```

\* Note that the above will only update the first entry it matches.

\* To update multiple entries, you can add `{multi:true}`, all countries listed as being in Africa will now show Antarctica as their continent

```shell

db.destinations.update({"continent": "Africa"}, {$set: {"continent": "Artic"}}, {multi: true})

```

\* Alternatively, we can use this syntax to update more than one record.

```shell

db.destinations.updateMany({"continent": "Artic"}, {$set: {"continent": "Antarctica"}})

```

\* what will happen when you run this command, even though a capital doesn't exist?

```shell

db.destinations.update({"country": "Morocco"}, {$set: {"capital": "Rabat"}})

```

\* Answer: it will add the capital field to the document & show the field can now be updated with the same command.

```shell

db.destinations.update({"country": "Morocco"}, {$set: {"capital": "RABAT"}})

```

\* Show how to push to an array with `$push`.

```shell

db.destinations.update({"country": "Morocco"}, {$push: {"major\_cities": "Agadir"}})

```

\* The upsert option updates a document if one exists;

it otherwise creates a new document.

```shell

db.destinations.update({'country': 'Canada'}, {$set: {'capital': 'Ottawa'}}, {upsert: true})

```

\* Show how to delete an entry with `db.[COLLECTION\_NAME].remove({justOne: true})`.

```shell

db.destinations.remove({"country": "Morocco"}, {justOne: true})

```

\* Show how to empty a collection with `db.[COLLECTION\_NAME].remove()`.

```shell

db.destinations.remove({})

```

\* Show how to drop a collection with `db.[COLLECTION\_NAME].drop()`.

```shell

db.destinations.drop()

```

\* Show how to drop a database

```shell

db.dropDatabase()

```

1.4-Stu\_DumpsterDB

**# Dumpster Database**

\* Create and use a database called "Dumpster\_DB".

```

use Dumpster\_DB

```

\* Create the "divers" collection and then insert a couple documents into it

```

db.divers.insert({"name":"Davey", "yearsDiving":10, "stillDiving":"True", "bestFinds":["Flat Screen", "Ruby Collar", "$100"]})

db.divers.insert({"name":"Jeanie", "yearsDiving":1, "stillDiving":"True", "bestFinds":["Movie Theater Chairs", "Music Box"]})

db.divers.insert({"name":"Boppo", "yearsDiving":5, "stillDiving":"True", "bestFinds":["Half-Eaten Hamburger", "Some Goop"]})

```

\* Update 'yearsDiving' so that it is one year higher for each diver

```

db.divers.update({"name":"Davey"},{$set:{"yearsDiving":11}})

db.divers.update({"name":"Jeanie"},{$set:{"yearsDiving":2}})

db.divers.update({"name":"Boppo"},{$set:{"yearsDiving":6}})

```

\* Update 'stillDiving' to False for Davey

```

db.divers.update({"name":"Davey"},{$set:{"stillDiving":"False"}})

```

\* Add a new value to Jeanie's "bestFinds"

```

db.divers.update({"name":"Jeanie"},{$push:{"bestFinds":"Mona Lisa"}})

```

\* Remove Boppo from the collection

```

db.divers.remove({"name":"Boppo"})

```

1.5-Ins\_PyMongo

the Pymongo library allows developers to use Python to work with MongoDB.

Pymongo serves as the interface between Python and MongoDB

# Module used to connect Python with MongoDb

import pymongo

# The default port used by MongoDB is 27017

# https://docs.mongodb.com/manual/reference/default-mongodb-port/

conn = 'mongodb://localhost:27017'

client = pymongo.MongoClient(conn)

# Define the 'classDB' database in Mongo

db = client.class3DB

# Query all students

# Here, db.students refers to the collection 'classroom '

classroom = db.classroom.find()

# Iterate through each student in the collection

for student in classroom:

print(student)

# Insert a document into the 'students' collection

db.classroom.insert\_one(

{

'name': 'Ahmed',

'row': 3,

'favorite\_python\_library': 'Matplotlib',

'hobbies': ['Running', 'Stargazing', 'Reading']})

# Update a document

db.classroom.update\_one(

{'name': 'Ahmed'},

{'$set':

{'row': 4}})

# Add an item to a document array

db.classroom.update\_one(

{'name': 'Ahmed'},

{'$push':

{'hobbies': 'Listening to country music'}})

# Delete a field from a document

db.classroom.update\_one({'name': 'Ahmed'},

{'$unset':

{'gavecandy': ""}})

# Delete a document from a collection

db.classroom.delete\_one(

{'name': 'Ahmed'})

1.6-Stu\_MongGrove

# Dependencies

import pymongo

import datetime

# The default port used by MongoDB is 27017

# https://docs.mongodb.com/manual/reference/default-mongodb-port/

conn = 'mongodb://localhost:27017'

client = pymongo.MongoClient(conn)

# Declare the database

db = client.fruits\_db

# Declare the collection

collection = db.fruits\_db

# Part I # A dictionary that represents the document to be inserted

post = {

'vendor': 'fruit star',

'fruit': 'raspberry',

'quantity': 21,

'ripeness': 2,

'date': datetime.datetime.utcnow()}

# Insert the document into the database

# The database and collection, if they don't already exist, will be created at this point.

collection.insert\_one(post)

# Part II # Ask the user for input. Store information into variables.

vendor = input('Vendor name: ')

fruit\_type = input('Type of fruit: ')

quantity = input('Number of boxes received: ')

ripeness = input('Ripeness of fruit (1 is unripe; 2 is ripe, 3 is over-ripe: ')

# A dictionary that will become a MongoDB document

post = {

'vendor': vendor,

'fruit': fruit\_type,

'quantity': quantity,

'ripeness': ripeness,

'date': datetime.datetime.utcnow()}

# Insert document into collection

collection.insert\_one(post)

# Verify results:

results = db.fruits\_db.find()

for result in results:

print(result)

10/1/18 – 2nd day – AD

Today's lesson dives into scraping websites with the Beautiful Soup library for Python.

Students will start out scraping simple HTML strings before moving onto live web pages.

Web **Scraping** (also termed Screen **Scraping**, Web Data Extraction, Web Harvesting etc.) is a technique employed to extract large amounts of data from **websites** whereby the data is extracted and saved to a local file in your computer or to a database in table (spreadsheet) format.

**### Class Objectives**

\* use Beautiful Soup to scrape data from the web.

\* save the results of web scraping into MongoDB.

brief introduction to Python's Beautiful Soup library,

an extremely powerful - albeit strangely named - tool for web scraping.

\* Up to this point, the class has been forced to rely upon analyzing web APIs and pre-

existing data sets. From this day forth, however, we can collect data from web resources that do not offer a full and convenient way to access to their data.

\* Beautiful Soup has to be installed before Python can use it.

To do this simply run `pip install bs4` within the terminal.

[Beautful Soup documentation]

(<https://www.crummy.com/software/BeautifulSoup/bs4/doc/> )

to read up on accessing attributes and navigating the DOM.

<https://www.crummy.com/software/BeautifulSoup/bs4/doc/#installing-a-parser>

\*Different Parsers ^\*

**Document Object** **Model** (DOM) is a cross-platform and language-independent API **application programming interface** that treats an **HTML**, XHTML, or XML **document** as a **tree** structure where in each node is an object representing a part of the **document.**

The DOM model represents a **document** with a logical **tree**.

**Right click – inspect element**

Use **Chrome**…in terminal, we can get token.

‘mongod’ in terminal to start MongoDB **JUST mongo FOR MACS**

Before Robo 3T

2.1-Ins\_SoupIntro

the DOM is a tree whose structure is defined by the nesting of tags. Beautiful Soup looks

through this tree and then converts it into a specialized object equipped with powerful methods for traversing and searching the HTML for attributes, text, etc.

# Dependencies

from bs4 import BeautifulSoup as bs

html\_string = """

<html>

<head>

<title>

A Simple HTML Document

</title>

</head>

<body>

<p>This is a very simple HTML document</p>

<p>It only has two paragraphs</p>

</body>

</html>

"""

# Create a Beautiful Soup object

soup = bs(html\_string, 'html.parser')

type(soup)

# Print formatted version of the soup

print(soup.prettify())

# Extract the title of the HTML document

soup.title

# Extract the text of the title

soup.title.text

# Clean up the text

soup.title.text.strip()

# Extract the contents of the HTML body

soup.body

# Extract the text of the body

soup.body.text

# Text of the first paragraph

soup.body.p.text

# Extract all paragraph elements

soup.body.find\_all('p')

# Extract paragraph by index

soup.body.find\_all('p')[0]

soup.body.find\_all('p')[1]

# The text of the first paragraph

soup.body.find('p').text

2.2-Stu\_CNNSoup

the [Beautful Soup documentation](<https://www.crummy.com/software/BeautifulSoup/bs4/doc/>

) to read up on accessing attributes and navigating the DOM.

# Dependencies

import os

from bs4 import BeautifulSoup as bs

# Read HTML from file

filepath = os.path.join("..", "Resources", "template.html")

with open(filepath) as file:

html = file.read()

# Create a Beautiful Soup object

soup = bs(html, 'lxml')

# Extract title text

title = soup.title.text

print(title)

# Print all paragraph texts

paragraphs = soup.find\_all('p')

for paragraph in paragraphs:

print(paragraph.text)

# Print all ten headlines

tds = soup.find\_all('td')

# A blank list to hold the headlines

headlines = []

# Loop over td elements

for td in tds:

# If td element has an anchor...

if (td.a):

# And the anchor has non-blank text...

if (td.a.text):

# Append the td to the list

headlines.append(td)

# Print only the headlines

for x in range(10):

print(headlines[x].text)

2.3-Ins\_Craigslist

# Dependencies

from bs4 import BeautifulSoup

import requests

# URL of page to be scraped

url = 'https://newjersey.craigslist.org/search/sss?sort=rel&query=guitar'

# Retrieve page with the requests module

response = requests.get(url)

# Create BeautifulSoup object; parse with 'html.parser'

soup = BeautifulSoup(response.text, 'html.parser')

# Examine the results, then determine element that contains sought info

print(soup.prettify())

# Another method of examining the HTML is to navigate to the webpage itself

# and open up the page's source within the inspector.

# results are returned as an iterable list

results = soup.find\_all('li', class\_="result-row")

# Loop through returned results

for result in results:

# Error handling

try:

# Identify and return title of listing

title = result.find('a', class\_="result-title").text

# Identify and return price of listing

price = result.a.span.text

# Identify and return link to listing

link = result.a['href']

# Print results only if title, price, and link are available

if (title and price and link):

print('-------------')

print(title)

print(price)

print(link)

except AttributeError as e:

print(e)

2.4-Stu\_RedditScrape

# Dependencies

from bs4 import BeautifulSoup

import requests

import os

filepath = os.path.join("Programmer-Humor.html")

with open(filepath, encoding='utf-8') as file:

html = file.read()

# Create BeautifulSoup object; parse with 'html.parser'

soup = BeautifulSoup(html, 'html.parser')

soup

# Find the number of subscribers

number\_subscribers = soup.find("span", class\_='subscribers').\

find('span', class\_='number').text

print(f"The number of subscribers: {number\_subscribers}")

# Examine the results, then determine element that contains sought info

# results are returned as an iterable list

results = soup.find\_all('div', class\_='top-matter')

# Loop through returned results

for result in results:

# Retrieve the thread title

title = result.find('p', class\_='title')

# Access the thread's text content

title\_text = title.a.text

# print(title\_text)

try:

# Access the thread with CSS selectors

thread = result.find('li', class\_='first')

# The number of comments made in the thread

comments = thread.text.lstrip()

# Parse string, e.g. '47 comments' for possible numeric manipulation

comments\_num = int(comments.split()[0])

# Access the href attribute with bracket notation

link = thread.a['href']

# Run if the thread has comments

if (comments\_num):

print('\n-----------------\n')

print(title\_text)

print('Comments:', comments\_num)

print(link)

except AttributeError as e:

print(e)

2.5-Ins\_MongoScraping

Ran this in jupyter nb, and activated MongoDB in it’s own terminal window, then I could see it in Robo 3T.

instead of `html.parser`, the `lxml` parser is being used. see [this

link] (<https://www.crummy.com/software/BeautifulSoup/bs4/doc/#installing-a-parser> ) to the class which provides an informative table on the various parsers available to Beautiful Soup and explain that some parsers are more flexible with parsing HTML than others. pip install lxml

# Dependencies

from bs4 import BeautifulSoup

import requests

import pymongo

# Initialize PyMongo to work with MongoDBs

conn = 'mongodb://localhost:27017'

client = pymongo.MongoClient(conn)

# Define database and collection

db = client.craigslist\_db

collection = db.items

# URL of page to be scraped

url = 'https://newjersey.craigslist.org/search/sss?sort=rel&query=guitar'

# Retrieve page with the requests module

response = requests.get(url)

# Create BeautifulSoup object; parse with 'lxml'

soup = BeautifulSoup(response.text, 'lxml')

soup

# Examine the results, then determine element that contains sought info

# results are returned as an iterable list

results = soup.find\_all('li', class\_='result-row')

# Loop through returned results

for result in results:

# Error handling

try:

# Identify and return title of listing

title = result.find('a', class\_='result-title').text

# Identify and return price of listing

price = result.a.span.text

# Identify and return link to listing

link = result.a['href']

# Run only if title, price, and link are available

if (title and price and link):

# Print results

print('-------------')

print(title)

print(price)

print(link)

# Dictionary to be inserted as a MongoDB document

post = {

'title': title,

'price': price,

'url': link}

collection.insert\_one(post)

except Exception as e:

print(e)

# Display items in MongoDB collection

listings = db.items.find()

for listing in listings:

print(listing)

2.6-Stu\_HockeyHeaders

# Dependencies

from bs4 import BeautifulSoup

import requests

import pymongo

# Initialize PyMongo to work with MongoDBs

conn = 'mongodb://localhost:27017'

client = pymongo.MongoClient(conn)

# Define database and collection

db = client.nhl\_db

collection = db.articles

# URL of page to be scraped

url = 'https://www.nhl.com/'

# Retrieve page with the requests module

response = requests.get(url)

# Create BeautifulSoup object; parse with 'lxml'

soup = BeautifulSoup(response.text, 'lxml')

# Retrieve the parent divs for all articles

results = soup.find\_all('li', class\_='mixed-feed\_\_item--article')

# Loop through results to retrieve article title, header, and timestamp of article

for result in results:

title = result.find('h4', class\_='mixed-feed\_\_header').text

lede = result.find('h5', class\_='mixed-feed\_\_subheader').text

# The time and date of article publication

date = result.find('time')['datetime']

# Slice the datetime string for the date

article\_date = date[:10]

# Slice the datetime string for the time

time = date[11:16]

# Determine whether article was published in AM or PM

if (int(time[:2]) >= 13):

meridiem = 'pm'

else:

meridiem = 'am'

# Concatenate time string

time = time + meridiem

print('-----------------')

print(title)

print(lede)

print(article\_date)

print(time)

# Dictionary to be inserted into MongoDB

post = {

'title': title,

'lede': lede,

'date': article\_date,

'time published': time}

# Insert dictionary into MongoDB as a document

collection.insert\_one(post)

# Display the MongoDB records created above

articles = db.articles.find()

for article in articles:

print(article)

2.7-Ins\_Splinter

chromedriver.exe file used for windows

Splinter is a Python module that automates browser actions such as visiting a URL,

filling fields, & clicking buttons.

It can be a very useful tool in the webscraper's arsenal!

chrome driver installation guide for the next exercise 2.7:

<https://splinter.readthedocs.io/en/latest/drivers/chrome.html>

must have the Chrome Webdriver installed for this activity

if `brew` installed, can simply run `brew install chromedriver` from terminal.

\* Up to this point, we have used Beautiful Soup to scrape a single, static page at

a time.

\* often, developers can only access interesting parts of a website after engaging in some kind of interaction with it.

\* typically, these interactions are pretty easy to automate: Logging in, filling out and submitting forms, etc.

\* when the data is "buried" behind such dynamic interactions, a web driver can be used to write scripts for the browser!

\* this allows developers to simulate user interactions programmatically and scrape multiple pages along the way.

from splinter import Browser

from bs4 import BeautifulSoup

# https://splinter.readthedocs.io/en/latest/drivers/chrome.html

!which chromedriver

executable\_path = {'executable\_path': '/usr/local/bin/chromedriver'}

browser = Browser('chrome', \*\*executable\_path, headless=False)

url = 'http://quotes.toscrape.com/'

browser.visit(url)

for x in range(1, 6):

html = browser.html

soup = BeautifulSoup(html, 'html.parser')

quotes = soup.find\_all('span', class\_='text')

for quote in quotes:

print('page:', x, '-------------')

print(quote.text)

# the application clicks on the `Next` button

browser.click\_link\_by\_partial\_text('Next')

2.8-Stu\_Splinter

**Bookscraper**

from splinter import Browser

from bs4 import BeautifulSoup

# https://splinter.readthedocs.io/en/latest/drivers/chrome.html

!which chromedriver

executable\_path = {'executable\_path': '/usr/local/bin/chromedriver'}

browser = Browser('chrome', \*\*executable\_path, headless=False)

url = 'http://books.toscrape.com/'

browser.visit(url)

# Iterate through all pages

for x in range(50):

# HTML object

html = browser.html

# Parse HTML with Beautiful Soup

soup = BeautifulSoup(html, 'html.parser')

# Retrieve all elements that contain book information

articles = soup.find\_all('article', class\_='product\_pod')

# Iterate through each book

for article in articles:

# Use Beautiful Soup's find() method to navigate and retrieve attributes

h3 = article.find('h3')

link = h3.find('a')

href = link['href']

title = link['title']

print('-----------')

print(title)

print('http://books.toscrape.com/' + href)

# Click the 'Next' button on each page

browser.click\_link\_by\_partial\_text('next')

2.9-Ins\_Pandas\_Scraping

Pandas has some built-in scraping capabilities.

use the `read\_html` function in Pandas to try and parse tabular data from HTML.

`read\_html` actually tries to convert as much of the HTML into DataFrames as possible and therefore returns a list of DataFrames (DF).

\* we can use list indexing to grab a reference to the DF that we are interested about.

import pandas as pd

# We can use the read\_html function in Pandas to automatically scrape any tabular data from a page.

url = 'https://en.wikipedia.org/wiki/List\_of\_capitals\_in\_the\_United\_States'

tables = pd.read\_html(url)

tables

# What we get in return is a list of dataframes for any tabular data that Pandas found.

type(tables)

# We can slice off any of those dataframes that we want using normal indexing.

df = tables[0]

df.columns = ['State', 'Abr.', 'State-hood Rank', 'Capital',

'Capital Since', 'Area (sq-mi)', 'Municipal Population', 'Metropolitan',

'Metropolitan Population', 'Population Rank', 'Notes']

df.head()

# Cleanup of extra rows

df = df.iloc[2:]

df.head()

# Set the index to the `State` column

df.set\_index('State', inplace=True)

df.head()

df.loc['Alabama']

# Pandas also had a `to\_html` method that we can use to generate HTML tables from DataFrames.

html\_table = df.to\_html()

html\_table

# You may have to strip unwanted newlines to clean up the table.

html\_table.replace('\n', '')

# You can also save the table directly to a file.

df.to\_html('table.html')

# OSX Users can run this to open the file in a browser,

# or you can manually find the file and open it in the browser

!open table.html

2.10-Stu\_Doctor\_Decoder

may need to install `html5lib` with the console command, `pip install html5lib`.

# Use Pandas scraping to help decode the medical abbreviations that a doctor might use.

import pandas as pd

url = 'https://en.wikipedia.org/wiki/List\_of\_medical\_abbreviations'

med\_abbreviations = ['BMR', 'BP', 'ECG', 'MRI', 'qid', 'WBC']

# Use Panda's `read\_html` to parse the url

tables = pd.read\_html(url)

tables

# Find the medical abbreviations DataFrame in the list of DataFrames and assign it to `df`

# Assign the columns `['abb', 'full\_name', 'other']`

df = tables[2]

df.columns = ['abb', 'full\_name', 'other']

df.head()

# drop the `other` column

del df['other']

# Drop the first row and set the index to the `abb` column

df = df.iloc[1:]

df.set\_index('abb', inplace=True)

df.head()

# Loop through the list of medical abbreviations and print the abbreviation

# along with the full description.

# Use the DataFrame to perform the lookup.

for abb in med\_abbreviations:

print(abb, df.loc[abb].full\_name)

10/3/18 – 3rd day – AD

**Rendering Your Data With Flask** **### Overview**

introduce students to rendering templates with Flask,

everything we need to know to display our data on a webpage.

**### Instructor Priorities**

\* create a Flask application that renders a static HTML template.

\* create a Flask application that renders an HTML template with data.

\* create a flask application that renders an HTML template with data from a Mongo database.

\* create a flask application that combines web scraping, document databases, and templating with Flask.

**## Class Objectives**

\* become comfortable rendering templates with Flask using data retrieved from a Mongo database.

\* use Beautiful Soup to scrape data

\* use PyMongo to save data to a Mongo database

\* use Flask to render templates

at the end of today's class, we will build a server that scrapes data; saves it to a database; and then renders that data to a webpage.

\* we will begin by rendering a static HTML template in Flask, and gradually work our way to serving templates whose data is from a running Mongo database.

We tied together python scripts, html code, and then ran the python script from terminal to see a web address, that we could load to a webpage

\* Rendering collections (dicts and lists) with Flask

\* Rendering views over MongoDB with Flask

\* Scraping data into MongoDB

3.1-Ins\_Render\_String

the basics of rendering a template with Flask.

using **\*\*templates\*\*** allows us to dynamically configure what is displayed in a "preconfigured" (i.e., templated) web page.

-- a major impetus for the use of templates is that it allows us to keep our webpage markup separate from our server logic.

--- the value of `text` is determined dynamically—we could set it equal to the result of a function call or *\_database query\_*, for example, and generate web pages reflecting the result of the query or function call.

# import necessary libraries

from flask import Flask, render\_template

# create instance of Flask app

app = Flask(\_\_name\_\_)

# create route that renders index.html template

@app.route("/")

def echo():

return render\_template("index.html", text="Serving up cool SHIZNIT from the Flask server!!")

if \_\_name\_\_ == "\_\_main\_\_":

app.run(debug=True)

3.2-Stu\_Render\_String

\* Consult the [Flask Render Docs](<http://flask.pocoo.org/docs/0.12/quickstart/#rendering->

Templates) for reference.

app.py -> run in terminal, then take http address from there to see it work

# import necessary libraries

from flask import Flask, render\_template

# create instance of Flask app

app = Flask(\_\_name\_\_)

# Set variables

name = "Aaron"

hobby = "Baseball"

# create route that renders index.html template

@app.route("/")

def echo():

return render\_template("index.html", name=name, hobby=hobby)

# Bonus add a new route

@app.route("/bonus")

def bonus():

return render\_template("bonus.html", name=name, hobby=hobby)

if \_\_name\_\_ == "\_\_main\_\_":

app.run(debug=True)

combined w index.html … and bonus.html

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<meta http-equiv="X-UA-Compatible" content="ie=edge">

<title>Templates 101</title>

<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css">

</head>

<body>

<div class="container">

<div class="jumbotron text-center">

<!-- Add Welcome message with name and hobby served from backend -->

<h1>Welcome {{ name }}</h1>

<p>{{ hobby }} sounds like a lot of fun!</p>

</div>

<!-- Add link to bonus -->

<p> Click <a href="bonus">here</a> to visit my bonus page!</p>

</div>

</body>

</html>

3.3-Ins\_Render\_List

app.py

# import necessary libraries

from flask import Flask, render\_template

# create instance of Flask app

app = Flask(\_\_name\_\_)

# create route that renders index.html template

@app.route("/")

def index():

team\_list = ["Jumpers", "Dunkers", "Dribblers", "Passers"]

return render\_template("index.html", list=team\_list)

if \_\_name\_\_ == "\_\_main\_\_":

app.run(debug=True)

index.html

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<meta http-equiv="X-UA-Compatible" content="ie=edge">

<title>Teams!</title>

<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css">

</head>

<body>

<div class="container text-center">

<h1 class="jumbotron">Team Rosters</h1>

<div>

<ul style="list-style: none;">

{% for name in list %}

<li>{{ name }}</li>

{% endfor %}

</ul>

</div>

</div>

</body>

</html>

3.4-Stu\_Render\_List

\* Add style to your webpage by using [Bootstrap

cards](<https://getbootstrap.com/docs/4.0/components/card/>)

# import necessary libraries

from flask import Flask, render\_template

# create instance of Flask app

app = Flask(\_\_name\_\_)

# create route that renders index.html template

@app.route("/")

def index():

movie\_list = ["Mighty Ducks", "Space Jam", "Clerks", "Batman", "Avengers"]

return render\_template("index.html", list=movie\_list)

if \_\_name\_\_ == "\_\_main\_\_":

app.run(debug=True)

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<meta http-equiv="X-UA-Compatible" content="ie=edge">

<title>Hurricanes!</title>

<link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.css">

</head>

<body>

<div class="container text-center">

<h1 class="jumbotron">Favorite Movies</h1>

<div>

{% for movie in list %}

<div class="col-lg-4">

<div class="card" style="width: 20rem;">

<div class="card-body">

<h4 class="card-title">{{ movie }}</h4>

<h6 class="card-subtitle mb-2 text-muted">Is a top five favorite movie</h6>

<p class="card-text">Search <a href="https://www.imdb.com/">IMDB</a> for more info!</p>

</div>

</div>

</div>

{% endfor %}

</div>

</div>

</body>

</html>

3.5-Ins\_Render\_Dict

"What is the difference between a list and a dict?

\* **\*\*dictionaries have key, value pairs\*\***.

\* in this activity we are going to access the dictionary values by using dot notation.

the "formula" for retrieving data via dot notation: `<dict\_name>.<key>`

# import necessary libraries

from flask import Flask, render\_template

# create instance of Flask app

app = Flask(\_\_name\_\_)

# create route that renders index.html template

@app.route("/")

def index():

player\_dictionary = {"player\_1": "Jessica",

"player\_2": "Mark"}

return render\_template("index.html", dict=player\_dictionary)

if \_\_name\_\_ == "\_\_main\_\_":

app.run(debug=True)

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<meta http-equiv="X-UA-Compatible" content="ie=edge">

<title>Sports!</title>

<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css">

</head>

<body>

<div class="container text-center">

<h1 class="jumbotron">Player Roster</h1>

<div>

<ul style="list-style: none;">

<li>{{ dict.player\_1 }}</li>

<li>{{ dict.player\_2 }}</li>

</ul>

</div>

</div>

</body>

</html>

3.6-Stu\_Render\_Dict

# import necessary libraries

from flask import Flask, render\_template

# create instance of Flask app

app = Flask(\_\_name\_\_)

# List of dictionaries

dogs = [{"name": "Fido", "type": "Lab"},

{"name": "Rex", "type": "Collie"},

{"name": "Suzzy", "type": "Terrier"},

{"name": "Tomato", "type": "Retriever"}]

# create route that renders index.html template

@app.route("/")

def index():

return render\_template("index.html", dogs=dogs)

if \_\_name\_\_ == "\_\_main\_\_":

app.run(debug=True)

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<meta http-equiv="X-UA-Compatible" content="ie=edge">

<title>Animal Adoption!</title>

<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css">

</head>

<body>

<div class="container text-center">

<h1 class="jumbotron">Dogs up for Adoption!</h1>

<div>

<ul style="list-style: none;">

<!-- Loop through the dictionary -->

{% for dog in dogs %}

<li style="color:blue">{{ dog.name }} who is a {{ dog.type }}</li>

{% endfor %}

</ul>

</div>

</div>

</body>

</html>

3.7-Ins\_Render\_From\_Mongo

Starting here, we used MongoDB

Type “ mongod “ then “mongo” in another tab in terminal to start it

from flask import Flask, render\_template

# Import our pymongo library, which lets us connect our Flask app to our Mongo database.

import pymongo

# Create an instance of our Flask app.

app = Flask(\_\_name\_\_)

# Create connection variable

conn = 'mongodb://localhost:27017'

# Pass connection to the pymongo instance.

client = pymongo.MongoClient(conn)

# Connect to a database. Will create one if not already available.

db = client.team\_db

# Drops collection if available to remove duplicates

db.team.drop()

# Creates a collection in the database and inserts two documents

db.team.insert\_many(

[

{

'player': 'Jessica',

'position': 'Point Guard'},

{

'player': 'Mark',

'position': 'Center'}])

# Set route

@app.route('/')

def index():

# Store the entire team collection in a list

teams = list(db.team.find())

print(teams)

# Return the template with the teams list passed in

return render\_template('index.html', teams=teams)

if \_\_name\_\_ == "\_\_main\_\_":

app.run(debug=True)

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<meta http-equiv="X-UA-Compatible" content="ie=edge">

<title>Basketball!</title>

<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css">

</head>

<body>

<div class="container text-center">

<h1 class="jumbotron">Basketball Players</h1>

<div>

<ul style="list-style: none;">

{% for team in teams %}

<li>{{ team.player }}: {{ team.position }}</li>

{% endfor %}

</ul>

</div>

</div>

</body>

</html>

3.8-Stu\_Render\_From\_Mongo

# insert\_data.py…run in terminal

import pymongo

# Setup connection to mongodb

conn = "mongodb://localhost:27017"

client = pymongo.MongoClient(conn)

# Select database and collection to use

db = client.store\_inventory

collection = db.produce

db.collection.insert\_many(

[

{

"type": "apples",

"cost": .23,

"stock": 333},

{

"type": "oranges",

"cost": .45,

"stock": 30},

{

"type": "kiwi",

"cost": .10,

"stock": 1000},

{

"type": "mango",

"cost": 1.30,

"stock": 20},

{

"type": "berries",

"cost": 2.99,

"stock": 99}])

print("Data Uploaded!")

# app.py…run in terminal

from flask import Flask, render\_template

import pymongo

app = Flask(\_\_name\_\_)

# setup mongo connection

conn = "mongodb://localhost:27017"

client = pymongo.MongoClient(conn)

# connect to mongo db and collection

db = client.store\_inventory

collection = db.produce

@app.route("/")

def index():

# write a statement that finds all the items in the db and sets it to a variable

inventory = list(db.collection.find())

print(inventory)

# render an index.html template and pass it the data you retrieved from the database

return render\_template("index.html", inventory=inventory)

if \_\_name\_\_ == "\_\_main\_\_":

app.run(debug=True)

# index.html

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<meta http-equiv="X-UA-Compatible" content="ie=edge">

<title>Stock!</title>

<link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.css">

</head>

<body>

<div class="container text-center">

<h1 class="jumbotron">Inventory</h1>

<div>

<!-- write a for loop that lists out your data and accesses the values using dot notation. -->

{% for item in inventory %}

<div class="col-lg-4">

<div class="card" style="width: 20rem;">

<div class="card-body">

<h4 class="card-title">{{ item.type }}</h4>

<h6 class="card-subtitle mb-2 text-muted">Cost: {{ item.cost }}</h6>

<p class="card-text">Potential Revenue: {{ item.cost \* item.stock}}</p>

</div>

</div>

</div>

{% endfor %}

</div>

</div>

</body>

</html>

3.9-Ins\_Scrape\_And\_Render

scrape a url, insert the data into Mongo, query it on the server, and render the query

results on the page.

This activity is similar to the previous one, with the additional requirement of web scraping.

new library being used called `flask\_pymongo`. Documentation can be found at <https://flask-pymongo.readthedocs.io/en/latest/>.

As defined by the docs *\_Flask-PyMongo bridges Flask and PyMongo, so that you can use Flask’s normal mechanisms to configure and connect to MongoDB.\_*

from flask import Flask, render\_template, redirect

from flask\_pymongo import PyMongo

import scrape\_craigslist

app = Flask(\_\_name\_\_)

# Use flask\_pymongo to set up mongo connection

app.config["MONGO\_URI"] = "mongodb://localhost:27017/craigslist\_app"

mongo = PyMongo(app)

# Or set inline

# mongo = PyMongo(app, uri="mongodb://localhost:27017/craigslist\_app")

@app.route("/")

def index():

listings = mongo.db.listings.find\_one()

return render\_template("index.html", listings=listings)

@app.route("/scrape")

def scraper():

listings = mongo.db.listings

listings\_data = scrape\_craigslist.scrape()

listings.update({}, listings\_data, upsert=True)

return redirect("/", code=302)

if \_\_name\_\_ == "\_\_main\_\_":

app.run(debug=True)

from splinter import Browser

from bs4 import BeautifulSoup

def init\_browser():

# @NOTE: Replace the path with your actual path to the chromedriver

executable\_path = {"executable\_path": "/usr/local/bin/chromedriver"}

return Browser("chrome", \*\*executable\_path, headless=False)

def scrape():

browser = init\_browser()

listings = {}

url = "https://raleigh.craigslist.org/search/hhh?max\_price=1500&availabilityMode=0"

browser.visit(url)

html = browser.html

soup = BeautifulSoup(html, "html.parser")

listings["headline"] = soup.find("a", class\_="result-title").get\_text()

listings["price"] = soup.find("span", class\_="result-price").get\_text()

listings["hood"] = soup.find("span", class\_="result-hood").get\_text()

return listings

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<meta http-equiv="X-UA-Compatible" content="ie=edge">

<title>Hot Finds</title>

<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css">

</head>

<body>

<div class="container">

<div class="jumbotron text-center">

<h1>Hot Finds On Craigslist</h1>

<p><a class="btn btn-primary btn-lg" href="/scrape" role="button">Find An Awesome Deal!</a></p>

</div>

<!-- Craigslist Listings -->

<div class="row" id="craiglist-listings">

<div class="col-md-12">

<h4 class="heading">{{listings.price}} {{listings.headline}}</h4>

<small>{{listings.hood}}</small>

</div>

</div>

</div>

</body>

</html>

3.10-Stu\_Scrape\_Weather

from splinter import Browser

from bs4 import BeautifulSoup

from datetime import datetime

# Initialize browser

def init\_browser():

# @NOTE: Replace the path with your actual path to the chromedriver

executable\_path = {"executable\_path": "/usr/local/bin/chromedriver"}

return Browser("chrome", \*\*executable\_path, headless=False)

# Function to scrape for weather in Cost Rica

def scrape\_weather():

# Initialize browser

browser = init\_browser()

# Visit the Costa Rica climate site

url = "https://weather-and-climate.com/average-monthly-Rainfall-Temperature-Sunshine-fahrenheit,san-jose,Costa-Rica"

browser.visit(url)

# Scrape page into soup

html = browser.html

soup = BeautifulSoup(html, "html.parser")

# Find today's forecast

forecast\_today = soup.find(

"div", class\_="weather-forecasts todays-weather forecast")

forecast\_today

# Get the max temp

max\_temp = forecast\_today.find("span", class\_="temp-max").text

# Print the min temp

min\_temp = forecast\_today.find("span", class\_="temp-min").text

# Get current time stamp

time\_stamp = str(datetime.now())

# Store in dictionary

weather = {

"time": time\_stamp,

"name": "Costa Rica",

"min\_temp": min\_temp,

"max\_temp": max\_temp,}

# Return results

return weather

# Function to scrape surf info

def scrape\_surf():

# Initialize browser

browser = init\_browser()

# Visit Costa Rica surf site

url = "https://www.surfline.com/surf-reports-forecasts-cams/costa-rica/3624060"

browser.visit(url)

# Scrape page into soup

html = browser.html

soup = BeautifulSoup(html, "html.parser")

# Find today's surf conditions

location = soup.find("h3", class\_="sl-spot-details\_\_name").get\_text()

surf\_height = soup.find("span", class\_="quiver-surf-height").get\_text()

# Store in dictionary

surf = {"location": location, "height": surf\_height}

# Return results

return surf

# import necessary libraries

from flask import Flask, render\_template, redirect

from flask\_pymongo import PyMongo

import scrape\_info

# create instance of Flask app

app = Flask(\_\_name\_\_)

# Use flask\_pymongo to set up mongo connection

app.config["MONGO\_URI"] = "mongodb://localhost:27017/weather\_app"

mongo = PyMongo(app)

# Or set inline

# mongo = PyMongo(app, uri="mongodb://localhost:27017/weather\_app")

# create route that renders index.html template and finds documents from mongo

@app.route("/")

def home():

# Find data

forecasts = mongo.db.collection.find()

# return template and data

return render\_template("index.html", forecasts=forecasts)

# Route that will trigger scrape functions

@app.route("/scrape")

def scrape():

# Run scraped functions

weather = scrape\_info.scrape\_weather()

surf = scrape\_info.scrape\_surf()

# Store results into a dictionary

forecast = {

"time": weather["time"],

"location": weather["name"],

"min\_temp": weather["min\_temp"],

"max\_temp": weather["max\_temp"],

"surf\_location": surf["location"],

"height": surf["height"],}

# Insert forecast into database

mongo.db.collection.insert\_one(forecast)

# Redirect back to home page

return redirect("/", code=302)

if \_\_name\_\_ == "\_\_main\_\_":

app.run(debug=True)

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<meta http-equiv="X-UA-Compatible" content="ie=edge">

<title>Document</title>

<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css">

</head>

<body>

<div class="container">

<div class="jumbotron">

<h1>Costa Rica Weather</h1>

<p><a class="btn btn-primary btn-lg" href="/scrape" role="button">Get Forecast!</a></p>

</div>

{% for forecast in forecasts%}

<div class="col-lg-4">

<div class="card" style="width: 20rem;">

<div class="card-body">

<h4 class="card-title">Costa Rica</h4>

<h6 class="card-subtitle mb-2 text-muted">{{ forecast.time }} - {{ forecast.name }}: {{ forecast.min\_temp }} | {{forecast.max\_temp}}</h6>

<p class="card-text">Surf: {{ forecast.surf\_location }} | {{ forecast.height }}</p>

</div>

</div>

</div>

{% endfor %}

</div>

</body>

</html>

**### Helpful terminal commands**

\* Find instances of Mongo `ps aux | grep mongod`

\* Kill process `kill -9 [pid]`

\* Drop Mongo Database `use <db name here>` then `db.runCommand( { dropDatabase: 1 } )`

Day 4 review

\* Review Mongo, Web Scraping, and Flask

- <https://github.com/the-Coding-Boot-Camp-at-UT/UTAUS201807DATA2/tree/master/class-content/13-Web-Scraping-and-Document-Databases/ReviewDay/Activities>

**## Unit 13 - Web Scraping and Document Databases**

**### Objectives**

\* Create and connect to local MongoDB databases.

\* Create, read, update, and delete MongoDB documents using the Mongo Shell.

\* Create simple Python applications that connect to and modify MongoDB databases using the PyMongo library.

\* Use Beautiful Soup to scrape their own data from the web.

\* Save the results of web scraping into MongoDB.

\* Become comfortable rendering templates w Flask using data retrieved from a Mongo db.

\* Use Beautiful Soup to scrape data.

\* Use PyMongo to save data to a Mongo database.

\* Use Flask to render templates.

- - -

**### Helpful Links**

\* [Mongo in 30 minutes](https://www.youtube.com/watch?v=pWbMrx5rVBE)

\* [Python Requests](http://docs.python-requests.org/en/master/)

\* [Webscraping with BeautifulSoup](https://www.dataquest.io/blog/web-scraping-tutorial-python/)

\* [Python Splinter](<https://splinter.readthedocs.io/en/latest/>)

\* [Flask Render Docs](http://flask.pocoo.org/docs/0.12/quickstart/#rendering-templates)

\* [Manage Mongod Processes](https://docs.mongodb.com/manual/tutorial/manage-mongodb-processes/)

\* [mongo vs mongod](https://stackoverflow.com/questions/4883045/mongodb-difference-between-running-mongo-and-mongod-databases)

\* [pymongo docs](https://api.mongodb.com/python/current/)

\* [splinter docs](https://splinter.readthedocs.io/en/latest/)