Data Analysis in Python

Apoorva Havanur, ahavanur

Judy Zhang, judyz

In this optional lecture, we're going to be covering the basics of how to analyze data with Python. We'll be covering the basics of retrieving data from the Yelp API, storing and querying that data in a SQL-lite database, and then analyzing and visualizing the data using Pandas and Matplotlib.

Yelp API

In order to access data from Yelp, we first need to acquire an API key from the Yelp Developers Authentication. After creating our account and starting a project, we can then make an api_key.txt file containing just our api key string in the same folder as our code. We can then access our api key string, make a few formatting edits, and then save it as a global variable to be used later.

In [95]:

```
def readAPIKey(filepath):
    # feel free to modify this function if you are storing the API Key differently
    with open('api_key.txt', 'r') as f:
        return f.read().replace('\n','')
```

In [96]:

```
apiKey = readAPIKey('api_key.txt')
```

Now that we have our api key, we can begin gathering data from Yelp. We first need to specify a header that authenticates and authorizes us as developers who can access Yelp's data. Then we need to specify the parameters of our search, what data we're trying to acquire. Let's say we want to get a list of some restaurants in a particular city. Then in our parameters we need to specify the location as the city we are looking for and the category of business as the restaurant. We'll leave the number and city to the user, so if we want the first 1000 restaurants in Pittsburgh, we can specify those as inputs to our function.

We also have to take in the limit that Yelp imposes on the number of search results we get. Most, if not all, APIs limit the number of results that you can receive in one call to the API. Yelp only allows us to get the first 20 results. However, it gives the option of adding an offset so that we can still access the next 20 results. Thus we need to increase our offset until we reach our desired results.

Finally, we need to access the API data itself. requests allows us to make HTTP requests to access the Yelp database and gather the information we need. It outputs a response in a JSON document, so we need to convert it into a Python-friendly format using json.loads. JSON documents are basically Python dictionaries. Finally we can collect the data on each business by accessing the correct key in the dictionary and appending it to our resulting list.

In [97]:

```python
import json
import requests
import time
def yelpSearch(api_key, city, total):
    """
    Make an authenticated request to the Yelp API.

    Args:
        api_key (string): Our Yelp API key
        city (string): the desired city
        total (integer): the number of restaurants

    Returns:
        businesses (list): list of dicts representing each business
    """
    offset = 0
    limit = 20

    header = {"Authorization": "Bearer %s" % api_key}
    param = {"location":city, "limit":limit, "offset":offset, "categories":"restaurants"}
    businesses = []
    while limit + offset <= total:
        time.sleep(0.25) #we want a slightly delay in our API call, otherwise Yelp might suspect that we're trying to
do something malicious
        param = {"location":city, "limit":limit, "offset":offset, "categories":"restaurants"}
        response = requests.get(url='https://api.yelp.com/v3/businesses/search', headers=header, params=param)
        data = json.loads(response.text)
        if "businesses" in data:
            businesses += data["businesses"]
        offset += 20
    return businesses
```

It's always good to test our code.

In [105]:

```
result = yelpSearch(apiKey, "Pittsburgh", 1000)
```

In [106]:

```
fields = result[0].keys()
for field in fields:
    print(field + ":", result[0][field])
```

```
1000
id: gaucho-parrilla-argentina-pittsburgh
name: Gaucho Parrilla Argentina
image_url: https://s3-media4.fl.yelpcdn.com/bphoto/OrC5JDiJz-XUtkTge9zjHA/o.jpg
is_closed: False
url:
https://www.yelp.com/biz/gaucho-parrilla-argentina-pittsburgh?adjust_creative=NiPHCwtXlXDTHR5J4QSuNw&utm
_campaign=yelp_api_v3&utm_medium=api_v3_business_search&utm_source=NiPHCwtXlXDTHR5J4QSuNw
review_count: 1401
categories: [{'alias': 'argentine', 'title': 'Argentine'}]
rating: 4.5
coordinates: {'latitude': 40.449043, 'longitude': -79.987573}
transactions: []
```

```
price: $$
location: {'address1': '1601 Penn Ave', 'address2': '', 'address3': '', 'city': 'Pittsburgh', 'zip_code': '15222',
'country': 'US', 'state': 'PA', 'display_address': ['1601 Penn Ave', 'Pittsburgh, PA 15222']}
phone: +14127096622
display_phone: (412) 709-6622
distance: 862.336104073366
```

Storing Our Data in a SQL Database

Now that we have a way of our collecting our data, we want a way to quickly and easily retrieve it, as well as being able to splice it efficiently. To do that, we're going to store our data into a **relational database.**

A database is a way of organizing data into tables that can be easily accessed. A relational database is one where the tables are connected to each other through identifying columns present in each table (like the how keys in a dictionary are unique). Nearly all relational databases use SQL (Structured Query Language) for querying and maintaining the database. You can create, modify, and interact with databases in Python using the sqlite3 library

Creating the Database

Our first step is to make the database that we are going to store our data in. To do this, we will need to identify what columns we want our database to have, and the type of each column. SQLite currently only supports 3 types: INTEGER, TEXT (strings), and REAL (floats), so you need to make sure that the data you feed it will match these types.

Looking at the fields that are returned from our API call, we can see what kind of data we'll get about each restaurant, and start thinking about what type we'll give them.

In [158]:

```python
import sqlite3
#SQL lite only supports the INTEGER, REAL, and TEXT datatypes
```

```
def createYelpDatabase(conn):
    c = conn.cursor()
    c.execute('''CREATE TABLE businesses
              (id TEXT,
               name TEXT,
               is_closed INTEGER,
               url TEXT,
               review_count INTEGER,
               alias TEXT,
               rating REAL,
               latitude REAL,
               longitude REAL,
               price TEXT,
               zip_code INTEGER,
               address TEXT,
               phone TEXT)
               ''')
conn = sqlite3.connect(":memory:") #tells sqlite to make the database in RAM
conn.text_factory = str #treat TEXT like strings and vice versa
createYelpDatabase(conn)
```

Loading Data

Great! Now that we've created our database, we're gonna fill it with data from our API call. Since we've already written the function that'll give us our data in a list of dictionary form, we're just going to parse through it and then add a list of tuples into our database using a SQL INSERT statement and sqlite's executemany method.

In [159]:

```
def loadDataIntoDatabase(conn, data):
    c = conn.cursor()
    businesses = []
    fields = ['id', 'name', 'is_closed', 'url', 'review_count', 'alias', 'rating', 'latitude', 'longitude', 'price', 'zip_code',
'address', 'phone']
    for business in data:
        row = []
        for field in fields:
            if field == 'is_closed':
                closed = business[field] #either True or False
                row.append(int(closed)) #convert the boolean into a integer 1 or 0
            elif field == 'alias':
                aliases = [] #alias is like a tag for the restaurant type.
                #To keep track of all of them, we'll put them together into a
                # string seperated by semicolons, i.e mexican;american;italian
                cats = business['categories']
                for subfield in cats:
                    aliases.append(subfield['alias'])
                row.append(';'.join(aliases)) #join the list together into semicolon seperated string
            elif field in ['latitude', 'longitude']:
                coordinate = business['coordinates']
                row.append(coordinate[field])
            elif field == 'zip_code':
                location = business['location']
                row.append(location['zip_code'])
            elif field == 'address':
                location = business['location']
                row.append(" ".join(location['display_address'])) #display address is a list of the components of the
address to the restaurant
            elif field not in business.keys(): #if our api call is missing any of the categories, we simply want to put in
a None
                row.append(None)
            else:
                value = business[field]
                row.append(value)
        row = tuple(row)
        businesses.append(row)
    c.executemany('INSERT INTO businesses VALUES (?,?,?,?,?,?,?,?,?,?,?,?,?)', businesses)
    conn.commit() #save the changes we made to our database
```

In [160]:

```
loadDataIntoDatabase(conn, result)
cursor = conn.cursor() #the cursor allows us to access the tables in our database
cursor.execute('SELECT COUNT(*) FROM businesses') # prints the number of rows in the database

cursor.fetchone()
```

Out[160]:

```
(1000,)
```

Querying the Database

Awesome! Now that we have our database set up, we can now start executing queries. Theres a whole lot that you can do with SQL, so we're just going to highlight some of the essentials.

To get a particular subset of the data, we can use the WHERE keyword, specifying a column name and the particular value we want the subset to have for that column. For example:

In [161]:

```
cursor.execute("SELECT * FROM businesses WHERE price = '$'") # SELECT * selects all the columns in a table (in our case, businesses)
cheap = cursor.fetchall() # will return a list of tuples, where each tuple represents a business that's price level is only one $ sign.
```

You can also use other operators in your WHERE clause, like !=, > , <, >=, <=, as well as special string formatting keywords. Here, we look for rows of our database where the alias value contains the word 'mexican'.

In [162]:

```
query = "SELECT name FROM businesses WHERE alias LIKE '%mexican%'"
cursor.execute(query)

mexican = cursor.fetchall()
for restaurant in mexican:
    print(restaurant)
```

```
('täkō',)
('Las Palmas Carniceria',)
('Las Palmas',)
```

```
('Doce Taqueria',)
('La Palapa, Traditional Mexican Kitchen',)
('Smoke BBQ Taqueria',)
('Condado Tacos',)
('Edgar Tacos Stand',)
('Totopo Mexican Kitchen and Bar',)
('Mad Mex - Shadyside',)
('Mad Mex - Oakland',)
('Reyna Foods',)
('Round Corner Cantina',)
('Bea Taco Town',)
("Emilliano's Mexican Restaurant and Bar",)
('Casa Reyna',)
('Tres Rios',)
('Las Velas',)
('Federal Galley',)
('Los Cabos Mexican Restaurant',)
('Bea Taco Town',)
('Patron Mexican Grill',)
('Steel Cactus',)
('Chipotle Mexican Grill',)
('Mad Mex - South Hills',)
('Steel Cactus',)
('Chipotle Mexican Grill',)
('Cocina Mendoza',)
('Mexi-Casa',)
('El Campesino',)
('Chipotle Mexican Grill',)
('Mendoza Express',)
('Bull River Taco',)
('Cuzamil Restaurante Mexicano',)
('El Paso Mexican Grill',)
('Bea Taco Town',)
('Chipotle Mexican Grill',)
('Patron Mexican Grill',)
('Steel Cactus',)
('La Catrina',)
('Chipotle Mexican Grill',)
```

Finally, we can aggregate our data based on a certain value. In this example, we aggregate our restaurants based on their zip code. Then, for each zip code, we count the number of rows using the COUNT function, and take the average rating of all the businesses in each zip code using AVERAGE. We also alias each column by using the as keyword.
We can also order our results based on a given column. Here, we order by the average rating, and specify that we want it in descending order using the DESC keyword.

In [164]:

```python
query = "SELECT zip_code, COUNT(1) as num_businesses, AVG(rating) as avg_rating FROM businesses GROUP BY zip_code ORDER BY AVG(rating) DESC"
cursor.execute(query)
byZipCode = cursor.fetchall()
for zipCode in byZipCode:
    print(zipCode)
```

```
(15208, 5, 4.6)
(15017, 1, 4.5)
(15116, 1, 4.5)
(15136, 1, 4.5)
(15223, 2, 4.5)
(15104, 2, 4.25)
(15243, 2, 4.25)
(15106, 11, 4.2272727272727275)
(15207, 7, 4.214285714285714)
(15218, 11, 4.090909090909091)
(15227, 11, 4.045454545454546)
(15215, 9, 4.0)
(15229, 8, 4.0)
(15240, 1, 4.0)
(15289, 1, 4.0)
(15201, 48, 3.9583333333333335)
(15221, 11, 3.9545454545454546)
(15224, 46, 3.9347826086956523)
(15210, 7, 3.9285714285714284)
(15209, 13, 3.923076923076923)
(15226, 13, 3.923076923076923)
(15233, 9, 3.888888888888889)
(15241, 8, 3.875)
(15206, 52, 3.8653846153846154)
(15228, 28, 3.8392857142857144)
(15234, 14, 3.8214285714285716)
(15211, 19, 3.8157894736842106)
(15203, 77, 3.811688311688312)
(15212, 48, 3.8020833333333335)
(15202, 10, 3.8)
(15205, 17, 3.7941176470588234)
(15236, 14, 3.75)
(15217, 55, 3.7454545454545456)
(15216, 32, 3.734375)
(15222, 149, 3.7248322147651005)
(15120, 14, 3.6785714285714284)
(15232, 34, 3.676470588235294)
(15237, 31, 3.629032258064516)
```

```
(15219, 47, 3.627659574468085)
(15220, 20, 3.625)
(15213, 89, 3.6179775280898876)
(15238, 22, 3.522727272727273)
```

We got data! Now let's visualize it.

Data Visualization

matplotlib is a great Python library for visualizing data. Data visualization is important so that we may draw conclusions from our results to answer questions or make decisions for the future. More information on what we can do with matplotlib can be found here.

For now, we'll just plot some basic bar and scatter plots. Using our newfound ~ SQL knowledge ~, we can begin to graph relationships on Yelp restaurant data. Say, we were curious as to which restaurants are the most reviewed. We can use pandas to query our database and compare the number of reviews a restaurant receives to how expensive it is. Thus, matplotlib makes our lives a lot easier.

In [165]:

```python
import pandas as pd
import matplotlib
import matplotlib.pyplot as plt
%matplotlib inline
plt.style.use('ggplot')
matplotlib.rcParams['figure.figsize'] = (10.0, 5.0) # you should adjust this to fit your screen

cheap = pd.read_sql("SELECT * FROM businesses WHERE price = '$' AND review_count > 10", conn)
pricey = pd.read_sql("SELECT * FROM businesses WHERE price > '$$' AND review_count > 10", conn)

plt.hist(cheap['review_count'])
plt.show()

plt.scatter(cheap['review_count'], cheap['rating'])
plt.show()

plt.hist(pricey['review_count'])
plt.show()

plt.scatter(pricey['review_count'], pricey['rating'])
plt.show()
```

We can see clearly that more expensive restaurants receive less reviews than their less pricey companions. There are clearly some restaurants that are more popular than others, judging from their number of reviews. Furthermore, more people are willing to rate a cheaper place higher than a more expensive place. We can draw some potential conclusions from this data. Maybe Pittsburgh just has better cheap food options. Maybe cheaper restaurants are less intimidating to review, or more commonly visited so they receive more reviews. Who knows? At least now, through data analysis and visualization, we have a greater understanding of what our community thinks of its restaurants.