Samantha Pratt, Natalie Anderson, and Mia Trotter
206 Final Project Report

Initial Goals

Our initial goals for the project were to utilize three APIs to conduct a comprehensive data analysis. The original APIs included a Kroger API, Zestful API, and Edamam API. From the Kroger API, we planned to collect item names, item availability, prices, and the Kroger store location. From the Zestful API, we planned to collect information about recipes including recipe names and ingredients. Finally, from the Edamam API, we planned to collect ingredient names and nutrition facts including calories, protein, fat, and carbohydrates. With this data, we wanted to analyze the nutritional profiles of recipes and identify commonly used ingredients. Then, we planned to examine the correlation between the availability and pricing of ingredients in Kroger supermarkets and their usage in recipes.

Achieved Goals

After discussing our original plans, we decided we wanted to move in a different direction. We did more research and found some additional APIs that we liked the format of better. Additionally, we decided we wanted to work with BeautifulSoup for this project. The website we worked with was a 2023 Quality of Life Index by City site. The APIs we used were a Population API and a Weather API. From the Quality of Life website, we collected the city name, country name, quality of life index, safety index, cost of living index, and traffic commute time index. From the Population API, we collected the city name, latitude, longitude, population, and if it was a capital city (boolean value). Finally, from the Weather API, we collected the city name, the current temperature in Fahrenheit, the 'feels like' temperature in Fahrenheit, and the wind speed in mph. Our main goal with this data was to analyze if the quality of life index of cities varies depending on factors like weather and population.

Problems

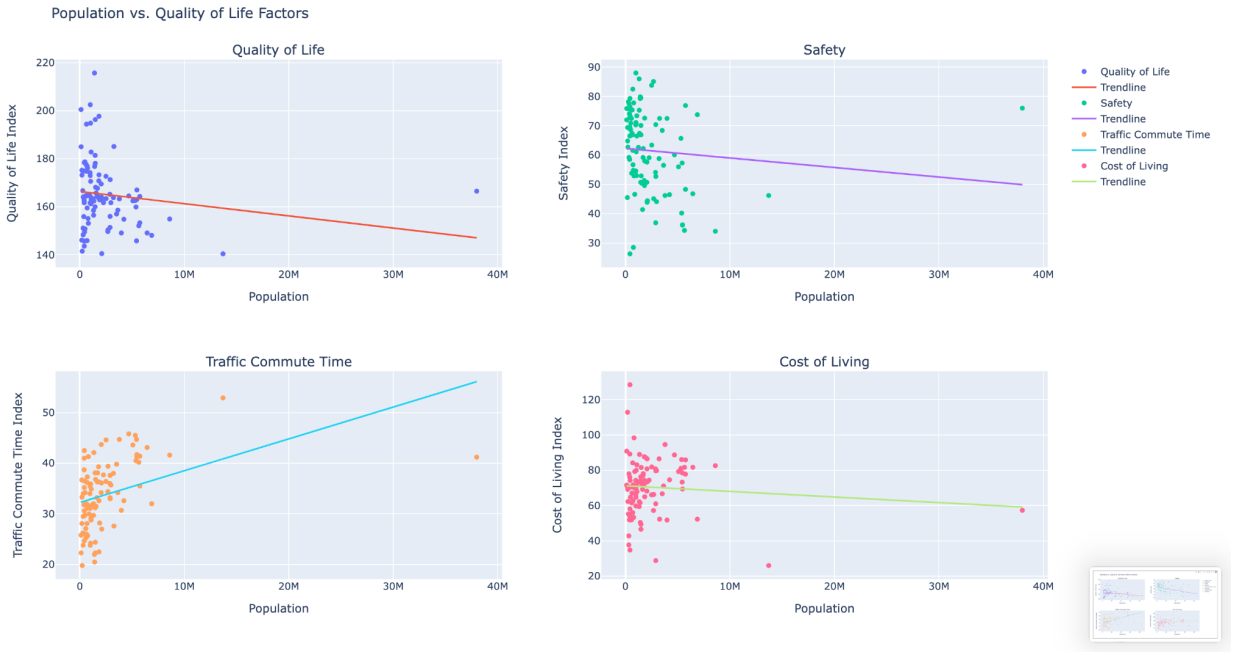
Initially, we struggled to formulate an initial project plan. It was difficult to find APIs that were easily accessible and that were easily connected to our ideas. This resulted in our implementation of using Beautiful Soup in the final project outcome in replacement of a third API. Additionally, the website we used for Beautiful Soup was difficult at first to extract information from using a loop. To solve this problem, we visited office hours to figure out what was going wrong on our end. Another major problem we encountered was with one city name obtained from the quality table not being listed the same as the query name for the population API. To mitigate this, we had to add an if statement to alter the city name for Porto Alegre before retrieving the city ID from the city table.

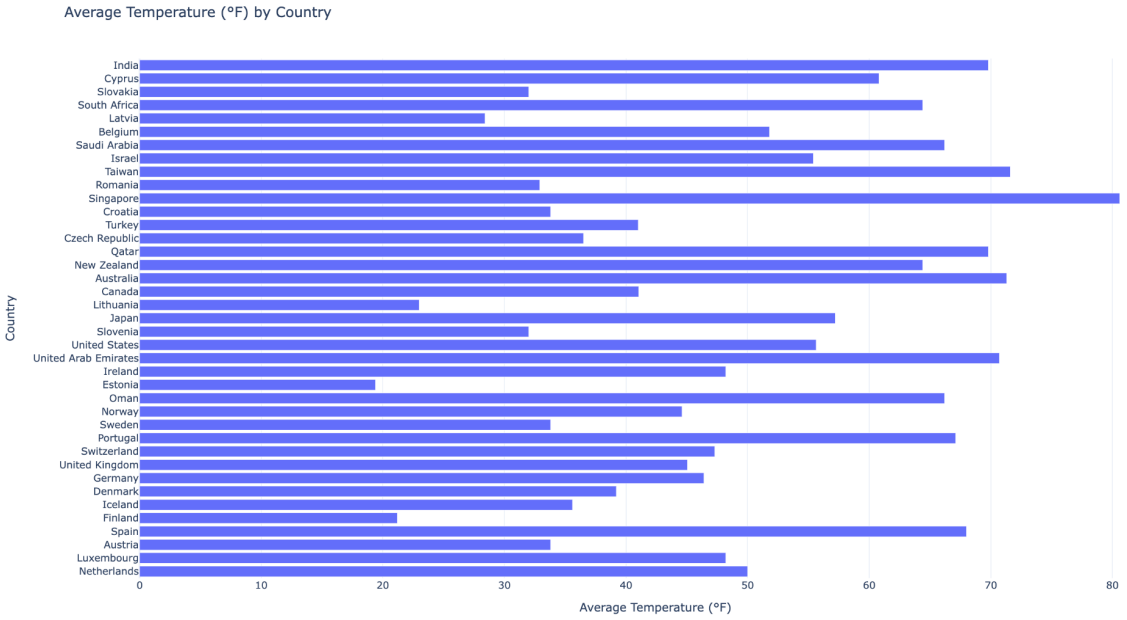
Calculations

```
Quality of Life Index per Capita

Top Five:
City: Quality of Life Index per Capita
Luxembourg: 0.001639773294185961
Reykjavik: 0.001436413469676147
Geneva: 0.0008681088901881893
Cork: 0.0008300226674781591
Brussels: 0.0007892902870293837

Bottom Five:
City: Quality of Life Index per Capita
Miami: 2.3132256465512227e-05
Riyadh: 2.1523034442668217e-05
Chicago: 1.8002829547373533e-05
Bangalore: 1.0242941562705188e-05
Tokyo: 4.3842325618137295e-06

-----------------------------

Average Temperature per Country

Top Five:
Country: Average Temperature ºF
Singapore: 80.6
Taiwan: 71.6
Australia: 71.3
United Arab Emirates: 70.69999999999999
Qatar: 69.8

Bottom Five:
Country: Average Temperature ºF
Slovakia: 32.0
Latvia: 28.4
Lithuania: 23.0
Finland: 21.2
Estonia: 19.4

-----------------------------

Average Safety Index of All Cities vs. Average Safety Index of Ten Most Populated Cities

All Cities: 61.44799999999999
Ten Most Populated: 50.86000000000001
```

```python
def calc1(cur, conn):
    cur.execute("SELECT cities.city_name, quality.quality / population.population FROM quality JOIN population ON quality.city_id = population.city_id JOIN cities ON cities.
    city_id = quality.city_id")
    results = cur.fetchall()
    return results

def calc2(cur, conn):
    cur.execute("SELECT countries.country_name, AVG(weather.temp_f) FROM weather JOIN population ON weather.city_id = population.city_id JOIN countries ON population.
    country_id = countries.country_id GROUP BY population.country_id")
    results = cur.fetchall()
    return results

def calc3(cur, conn):
    cur.execute("SELECT AVG(safety) FROM quality")
    avg_safety = cur.fetchall()[0][0]
    cur.execute("SELECT quality.safety FROM quality JOIN population ON quality.city_id = population.city_id ORDER BY population.population DESC LIMIT 10")
    avg_safety10 = cur.fetchall()
    total = 0
    for city in avg_safety10:
        total += city[0]
    avg_safety10 = total/10
    return avg_safety, avg_safety10
```

Visualizations




Average Temperature (°F) by Country
India
Cyprus
Slovakia
South Africa
Latvia
Belgium
Saudi Arabia
Israel
Taiwan
Romania
Singapore
Croatia
Turkey
Czech Republic
Qatar
New Zealand
Australia
Canada
Lithuania
Japan
Slovenia
United States
United Arab Emirates
Ireland
Estonia
Oman
Norway
Sweden
Portugal
Switzerland
United Kingdom
Germany
Denmark
Iceland
Finland
Spain
Austria
Luxembourg
Netherlands
Country
0
10
20
30
40
50
60
70
80
Average Temperature (°F)


Edmonton
Calgary
Vancouver
Victoria
Seattle
Portland
Minneapolis
Quebec City
Ottawa
Montreal
Halifax
Toronto
Mississauga
Chicago
Boston
Pittsburgh
Columbus
Philadelphia
Washington
Salt Lake City
Denver
Kansas City
Sacramento
San Jose
Las Vegas
Albuquerque
Raleigh
Charlotte
Atlanta
Phoenix
Tucson
San Diego
Dallas
Austin
San Antonio
Houston
Orlando
Tampa
Miami
Honolulu
Valencia

Instructions for Running Code

First, run the quality_gather_data.py file at least one time and at most four times. Next, you can run the pop_gather_data.py file at least one time and at most as many times as you ran quality_gather_data.py. Next, you can run the weather_gather_data.py file at least one time and at most as many times as you ran pop_gather_data.py. Then, you can run the visualizations.py

file, which writes to the text file calculation_results.txt and opens three visualizations in your browser.

Quality Gather Data File

The first function in this file, *setUpDatabase*, takes in a database name as the input. It creates a connection to a SQLite database with the specified name (‘cities.db’), using the file path derived from the current script's location, and returns a cursor and connection object for database operations. The main function of the file, *quality_gather_data*, takes in a URL, a cursor, and a connection object as inputs. It makes a request to the given URL, parses the HTML content, and extracts information about cities' quality, safety, cost, and traffic. It then creates tables for cities, countries, and quality in a SQLite database using *create_cities_table, create_countries_table*, and *create_quality_table* and adds the gathered data to the corresponding tables using *add_cities*, *add_countries*, and *add_quality_data*.

The *create_cities_table* function takes in the cursor and connection object as inputs. It creates a table named 'cities' in the database, defining columns for city_name and city_id, if it doesn't already exist, and then commits the changes to the database. The *create_countries_table* takes in the cursor and connection object as inputs. It creates a table named 'countries' in the database, defining columns for country_name and country_id, if it doesn't already exist, and then commits the changes to the database. The *create_quality_table* function takes in the cursor and connection object as inputs. It creates a table named 'quality' in the database, defining columns for city_id, country_id, quality, safety, cost, and traffic, if it doesn't already exist, and then commits the changes to the database.

The *add_cities* function takes in the cursor, connection object, and a list as inputs. It adds city data from the given list to the 'cities' table in a SQLite database. It iterates through a range of 25 items at a time, retrieves city_name from the list, and inserts this data into the 'cities' table. The function then commits the changes to the database. The *add_countries* takes in the cursor, connection object, and a list as inputs. It adds country data from the given list to the 'countries' table in the database. It iterates through a range of 25 items at a time, retrieves country_name from the list, checks if the country already exists in the table, inserts new countries into the 'countries' table with unique country_id, and then commits the changes to the database. The *add_quality_data* takes in the cursor, connection object, and a list as inputs. It adds quality data from the given list to the 'quality' table in the database. It iterates through a range of 25 items at a time, retrieves city_name, country_name, quality, safety, cost, and traffic from the list, retrieves corresponding city_id and country_id from the 'cities' and 'countries' tables, and inserts this data into the 'quality' table. The function then commits the changes to the database.

Then, the *get_cities_size* takes in the cursor and connection object as inputs. It retrieves the current count of records in the 'cities' table of the database by executing a SQL query to count and return the number of rows. Similarly, the *get_countries_size* function takes in the cursor and

connection object as inputs. It retrieves the current count of records in the 'countries' table of the database by executing a SQL query to count and return the number of rows. Finally, the *get_quality_size* function takes in the cursor and connection object as inputs. It retrieves the current count of records in the 'quality' table of the database by executing a SQL query to count and return the number of rows.

Population Gather Data File

The population gather data file begins with *setUpDatabase*, once again setting up the connection to the SQLite cities database, taking the database name as the input and returning a cursor and connection object. Next, the *population_gather_data* function takes in the cursor and connection object as inputs. It retrieves 25 city names from the cities table in the database, queries the Population API for population data for each city, processes the API responses, creates a population table in the database using the *create_population_table* function, and adds the gathered population data to the database using the *add_population_data* function.

The *create_population_table* function takes in the cursor and connection object as inputs. It creates a table named ‘population’ in a SQLite database, defining columns for city_id, latitude, longitude, country_id, population, and is_capital, if it doesn't already exist, and then commits the changes to the database. The *add_population_data* function takes in the cursor, connection object, and a list as inputs. The function adds population data from the given list to the population table in the database. It iterates through a range of 25 items at a time, retrieves the city id, latitude, longitude, country id, population, and if the city is the capital or not from the list, and inserts this data into the population table. Additionally, it checks for "Porto Alegre" and modifies it to "Porto" before processing and then commits the changes to the database. Finally, *the get_population_size* function takes in the cursor and connection object as inputs. It retrieves the current count of records in the population table of the database by executing a SQL query to count and return the number of rows.

Weather Gather Data File

Once again, this file starts with the *setUpDatabase*. The *weather_gather_data* function takes in the cursor and connection object as inputs. It retrieves latitude and longitude data from the 'population' table in the cities database, queries the Weather API for current weather information for 25 locations, processes the API responses, creates a 'weather' table in the database using *create_weather_table*, and adds the gathered weather data to the database using *add_weather_data*. The *create_weather_table* function takes in the cursor and connection object as inputs. It creates a table named 'weather' in a SQLite database, defining columns for city_id, temp_f, feelslike_f, and wind_mph, if it doesn't already exist, and then commits the changes to the database. The *add_weather_data* function takes in the cursor, connection object, and a list as inputs. It adds weather data from the given list to the 'weather' table in the database. It iterates through a range of 25 items at a time, retrieves temp_f, feelslike_f, and wind_mph from the list,

and inserts this data into the 'weather' table. The function then commits the changes to the database. Finally, the *get_weather_size* function takes in the cursor and connection object as inputs. It retrieves the current count of records in the 'weather' table of the database by executing a SQL query to count and return the number of rows.

Resource Documentation

Date	Issue Description	Location of Resource	Result (did it solve the issue?)
12/1/23	We couldn't figure out the right class to use in the beautiful soup statement to collect the data from the table.	Office Hours	Yes, we were able to access the table data using the id instead of the class name.
12/7/23	We were confused about the number of rows we were supposed to be accessing/adding to the database and what the rubric meant when saying access 100 lines of code but only store 25 items at a time.	Office Hours	We now understand how we should fix our code to correctly follow the guidelines outlined. We need to access the data from the databases using queries rather than passing a list of lists between each of our apis/beautiful soup.
12/7/23	We needed to tell our code what 25 items to collect and add to the database. We did not know how to find the number of rows in the database.	https://www.freecodecamp.org/news/sql-distinct-statement-how-to-query-select-and-count/#:~:text=Conclusion-,In%20SQL%2C%20you%20can%20make%20a%20database%20query%20and%20use,as%20part%20of%20the%20count.	We found this information on using COUNT(*) to get the number of rows in the database. We were able to use this as a start variable and add 25 to it to get the range of rows to be gathered/added.
12/8/23	We wanted to add a trendline to our scatter plots. Also, we	ChatGPT	Yes, it provided us with information on adding a trendline to

	wanted to plot points on a map rather than a flat graph.		our plots as well as what packages to use for graphing on a map.
12/10/23	We wanted to see the function parameters for writing to a text file.	https://www.pythontutorial.net/python-basics/python-write-text-file/	Yes, we found information on .write() and .writelines() which we used in our code.