SI 206 - Final Project Report

Prof. Barbara Ericson

Team: AJS Force

Audrey Kowara (akowara@umich.edu)
Jessica Yang (jesyang@umich.edu)
Sydney Emuakhagbon (semuak@umich.edu)

Git Repository:

https://github.com/audreykowara/AJSForce206.git

Project Plan (Goals):

- We want to create a database that consists of data that help people to plan their holiday (look at the weather forecast, restaurant / food recommendations, attraction recommendations)
- Initial plan:
    - Use 3 API (Yelp, AccuWeather, TripAdvisor) to complete this project
    - Data to gather:
        - Weather: (in October)
            - City name: Chicago
            - Current temperature (°C or °F)
            - Weather condition (e.g., sunny, snowy)
            - Wind speed
        - Yelp:
            - Restaurant name
            - Cuisine type
            - Rating
            - Price range
            - Number of reviews
            - Location: Chicago
        - TripAdvisor:
            - Activity/attraction name
            - Category (e.g., sightseeing, adventure)
            - Rating
            - Number of reviews
            - Location : Chicago

Project Achievements:

- WeatherAPI:

- Achieved in gathering the forecast between 7 days (from current date, in my case, it was 13/12/2024).
- Data gathered (for several cities):
  - Temperature (°F)
  - Weather condition (e.g., sunny, snowy)
  - Wind speed (MPH)
  - Humidity
- Achieved to calculate the average temperature and wind speed of each cities and store it in a csv file (city_weather_averages.csv)
- Achieved to visualize the calculator into a dual-axis graph (dual_axis_chart.png)

- Yelp:
  - Actual gathered data (for restaurants in Chicago):
    - Business ID
    - Name
    - Location (address)
    - Rating
    - Price range
  - Successfully stored the gathered restaurant data in the database (AJSChicago_data.db).
  - Created a CSV file (price_percentages.csv) with price range percentages for restaurants.
  - Created a Pie Chart Visualization using matplotlib to represent the distribution of restaurants by price range.
- Google Places:
  - Gathered information about various activities in Chicago, such as restaurants, parks, and museums.
    - Place Name
    - Address
    - User Rating
    - Category
    - User Review Count
  - Successfully stored the gathered data in the database (AJSChicago_data.db).
  - Created a CSV file for specific data for the Itinerary Table, namely category, average rating, and count of attractions.
  - Created a dual-axis chart showing the average rating and attraction count for each category using matplotlib.

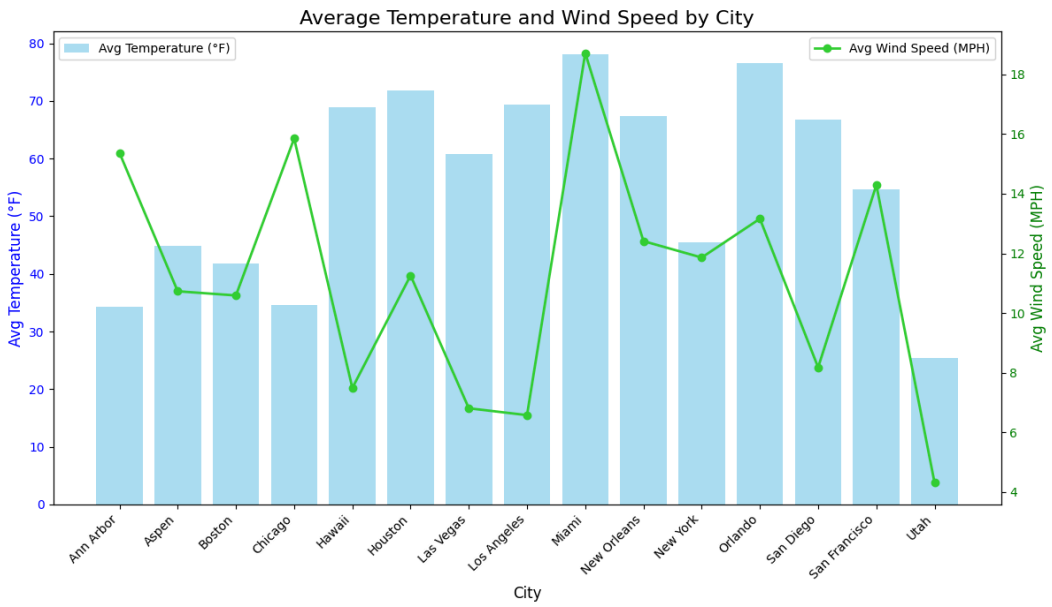
WeatherAPI - Audrey Kowara

a) Problems that I encountered:
- Many weather APIs require a paid subscription to access advanced features, such as long-term forecasts or historical weather data.
- Free-tier APIs often come with limitations, such as:
  - Limited number of API calls per day.
  - Restricted access to specific types of weather data (e.g., only current conditions or a 3-day forecast).
  - The range of forecast data offered by most APIs is limited to a few days (e.g., 7 days maximum in the free tier), making it difficult to analyze long-term trends or seasonal data
  - This restriction required us to adjust the project scope, as we initially planned to focus on one city (Chicago) with a comprehensive monthly forecast or long-term data analysis.

b) Calculation and Visualization:

city_weather_averages

City Name	Avg Temperature (°F)	Avg Wind Speed (MPH)
Ann Arbor	34.34	15.37
Aspen	44.93	10.73
Boston	41.74	10.59
Chicago	34.56	15.85
Hawaii	68.86	7.49
Houston	71.8	11.26
Las Vegas	60.85	6.81
Los Angeles	69.41	6.58
Miami	78.13	18.71
New Orleans	67.45	12.41
New York	45.43	11.86
Orlando	76.53	13.16
San Diego	66.71	8.18
San Francisco	54.59	14.29
Utah	25.33	4.31



c) Instruction to run my code:

Fetch and Store Data

1. Get WeatherAPI Key:
   - Sign up at [WeatherAPI] to get your API key. Once signed up, log in to your account and copy your unique API key.
2. Setting up your environment (Using VSCode and Github):
   - Clone the GitHub Repository:
     a) Open the terminal in VS Code
     b) Navigate to the directory where you want to store the project. In this project, Desktop > AJS
     c) Clone the GitHub repository (link is provided on top of this document)
     d) Navigate into the cloned repo, in this case, I used this command — cd AJSForce206
   - Set up the project in VS Code:
     a) Open the folder in VS Code by selecting File > Open Folder, then navigate to the cloned AJSForce206 folder
     b) Create the Python File (AJSFroce206.py)
     c) Write and save the code in a Python file code.
     d) Replace the API_KEY placeholder with your actual WeatherAPI key. In my case, my API_KEY is "8a5c7cf8930644dab7414532241412"
     e) Install the requests library — `pip install requests`.

     Note: SQLite3 and time module are a part of Python's standard Library, therefore, no installation is needed.
3. Running the code:

- Execute the code
- Ensure the database is created and data is fetched successfully. In this case, the database is “AJSChicago_data.db”

4. Commit and push changes to GitHub:

   Note: Ensure you are already inside the repository directory when working on the code. You can navigate to the repository directory using the cd command.
   For example:

   ```
   cd Desktop

   cd AJS

   cd AJSForce206
   ```

   - Git add any files that you want to be added to the repo. In this case, AJSFroce206.py and AJSChicago_data.db

     Example command: `git add AJSFroce206.py`

   - Commit changes with git commit

     Example command: `git commit -m”Create database”`

   - Push the changes to GitHub using “git push” command

5. Verify on GitHub
   - Open the GitHub repository in a browser
   - Navigate to the repo
   - Confirm that the changes are committed / visible in the repository

Data Calculation and Visualization

1. Create the python file:
   - In the same directory as the "Fetch and Store Data" script (AJSFroce206.py), create a new Python file named weathercalcandviz.py.
   - This file will handle calculations and visualization of the weather data gathered in the database.
2. Verify the database contains data
3. Write, save, and execute the code (weathercalcandviz.py)
4. Verify that the following files have been created:
   - city_weather_averages.csv: A CSV file containing average temperature and wind speed for each city.
   - dual_axis_chart.png: A visualization file showing the bar chart for temperature and a line chart for wind speed.
5. Commit and push changes to GitHub:
   - Git add any files that you want to be added to the repo. In this case, weathercalcandviz.py, city_weather_averages.csv, dual_axis_chart.png

Example command: `git add weathercalcandviz.py`

- Commit changes with git commit
  Example command: `git commit -m"Weather Calculation"`
- Push the changes to GitHub using "git push" command

Note: Ensure to commit every changes and add the files needed

6. Verify on GitHub
   - Open the GitHub repository in a browser and confirm that all the added files (weathercalcandviz.py, city_weather_averages.csv, dual_axis_chart.png) are now listed under your project files.

d) Functions Documentation:

1. setup_database():
   - This function initializes the SQLite database by creating two tables: `cities` and `weather`. The `cities` table stores unique city names, and the `weather` table stores daily weather data for each city. It also enforces unique constraints to prevent duplicate entries for the same city and date.
   - Input: None
   - Output:
     - Creates the database file (`AJSChicago_data.db`) if it does not already exist.
     - Ensures the schema includes foreign key relationships between the `cities` and `weather` tables.
     - Commits all changes and closes the database connection.

2. fetch_weather_data(api_key, city_name, start_day=0, days_per_run=7):
   - This function fetches weather forecast data for a specific city using the WeatherAPI. It retrieves weather details for multiple days starting from a specified day and parses the JSON response to extract relevant information.
   - The function validates the response format to ensure all required data is available.
   - Raises exceptions for HTTP errors or invalid responses.
   - Input:
     - 'API_KEY' (str): The API key required to authenticate with WeatherAPI.
     - 'city_name' (str):The name of the city for which weather data is being requested.
     - 'start_day' (int): The starting day for the forecast (default is 0, meaning today).
     - 'days_per_run' (int): The number of forecast days to retrieve (default is 7).

- Output:
    - ‘collected_data’ (list): A list of dictionaries where each dictionary contains:
    - ‘city_name’ (str): The city name.
    - ‘temperature’ (float): The maximum temperature in Fahrenheit.
    - ‘condition’ (str): A description of the weather condition (e.g., "Sunny").
    - ‘wind_speed’ (float): The maximum wind speed in miles per hour.
    - ‘humidity’ (int): The average humidity percentage.
    - ‘date’ (str): The forecast date in YYYY-MM-DD format.

3. store_weather_data(data, max_rows=25, current_total_rows=0):
    - This function inserts weather data into the SQLite database. It ensures that the city names are stored in the ‘citie’ table and corresponding weather data is stored in the ‘weather’ table. The function enforces a row limit to control the size of the database.
    - Input:
        - ‘data’ (list): A list of dictionaries, each containing weather data for a specific day.
        - ‘max_rows’ (int): The maximum allowed number of rows in the database (default is 25).
        - ‘current_total_rows’ (int): The current number of rows already present in the database (default is 0).
    - Output:
        - ‘rows_added’ (int): The number of rows successfully inserted into the `weather` table.
    - Details:
        - Each city's name is inserted into the ‘cities’ table if it does not already exist.
        - The corresponding weather data is inserted into the ‘weather’ table, avoiding duplicates based on the unique combination of ‘city_id’ and ‘date’.
        - Prints a message for any duplicate entries that are skipped.

4. verify_database():
    - This function verifies the integrity of the SQLite database by counting the total number of rows in the `weather` table and checking for duplicate entries. Prints the results of these checks to the console.
    - Input: None
    - Output: None
    - Details:
        - Counts and displays the total rows in the ‘weather’ table.

    - Identifies and lists any duplicate entries based on the combination of ‘city_id’ and ‘date’.
    - Closes the database connection after verification.

5. calculate_average_weather():
    - This function calculates the average temperature and wind speed for each city from the stored weather data (AJSChicago_data.db) and saves the calculation results to a CSV file (city_weather_averages.csv) for further analysis or visualization.
    - Input: (Implicit Input) SQLite database (AJSChicago_data.db)
    - Output:
        - Writes the results to a CSV file named `city_weather_averages.csv` in the format:
            - Columns: ‘City Name’, ‘Avg Temperature (°F)’, ‘Avg Wind Speed (MPH)’
            - Rows: One row per city, with averages calculated from the available weather data.
    - Details:
        - ‘output_file’ (str): The file path of the created CSV file.
        - Connects to the SQLite database to query the ‘weather’ and ‘cities’ tables.
        - Calculates the average temperature and wind speed for each city.
6. visualize_data_from_csv(csv_file):
    - This function reads weather data from the CSV file (city_weather_averages.csv) and visualizes it using a dual-axis chart. The chart shows average temperature as a bar plot and average wind speed as a line plot for each city.
    - Input:
        - ‘csv_file’ (str): The name of the CSV file containing weather averages. The file should have the following columns: ‘City Name`, ‘Avg Temperature (°F)’, ‘Avg Wind Speed (MPH)’.
    - Output:
        - The function does not return a value
        - Generates a chart image file named `dual_axis_chart.png` in the current working directory.
    - Details:
        - Extracts city names, average temperatures, and average wind speeds from the CSV file.
        - Creates a dual-axis chart using Matplotlib:
            - The primary y-axis (left) represents average temperatures as a bar plot.
            - The secondary y-axis (right) represents average wind speeds as a line plot.
        - Labels axes, adds legends, and ensures a clear and visually appealing layout.

- Saves the chart as ‘dual_axis_chart.png’ in the current working directory.
- Handles errors such as missing CSV files or unexpected issues during processing.

7. main():
   - The main function orchestrates the entire process of fetching, storing, and verifying weather data. It initializes the database, iterates through a predefined list of cities, fetches weather data for each city, and stores the data in the database. It also enforces a maximum row limit for each run and verifies the database at the end.
   - Input: None
   - Output: None
   - Details:
     - Defines a list of cities to fetch weather data for.
     - Call ‘setup_database’ to initialize the database.
     - Iterates through each city in the list:
       a. Fetches weather data using ‘fetch_weather_data’.
       b. Stores the data in the database using ‘store_weather_data’.
       c. Enforces a pause between API calls to avoid rate limits.
     - Calls ‘verify_database’ to validate the database at the end of execution.
     - After fetching and storing weather data for the predefined cities, it calculates average weather data using `calculate_average_weather` and visualizes it using `visualize_data_from_csv`.
     - The CSV file generated in the `calculate_average_weather` function is used as input for the visualization step.
     - Ensures all steps are executed sequentially to prepare and analyze the data effectively.

e) Resources Documentation:

Date	Issue Description	Location of Resource	Result (did it solve the issue)
12/10/2024	Finding out only offering limited historical data	AccuWeather Historical Current Conditions	AccuWeather's free tier does not provide extensive historical data. I have to find another API to work with.

12/11/2024	Finding a better API to use for a free tier account	ChatGpt List of API	Yes, give me recommendations on which API is better to use. Finally, decided to use WeatherAPI
12/13/2024	Merging Database in GitHub	ChatGPT	Yes, provided step-by-step guidance on merging file on the repository
12/13/2023	Matpotlib for visualization	Matpotlib Documentation	Yes, as a guidance for creating visualization successfully

Yelp API - Jessica Yang

Problems I encountered:

- API Pagination Complexity: Difficulty in systematically fetching all restaurant data without duplicates or missing entries
- Price Range Mapping: Converting Yelp's string-based price indicators ($ to $$$$) to a consistent numerical representation
- Error Handling: Implementing comprehensive error management for API requests, database insertions, and file operations
- Offset Tracking: Creating a reliable mechanism to track and resume data collection across multiple script runs

Calculations:

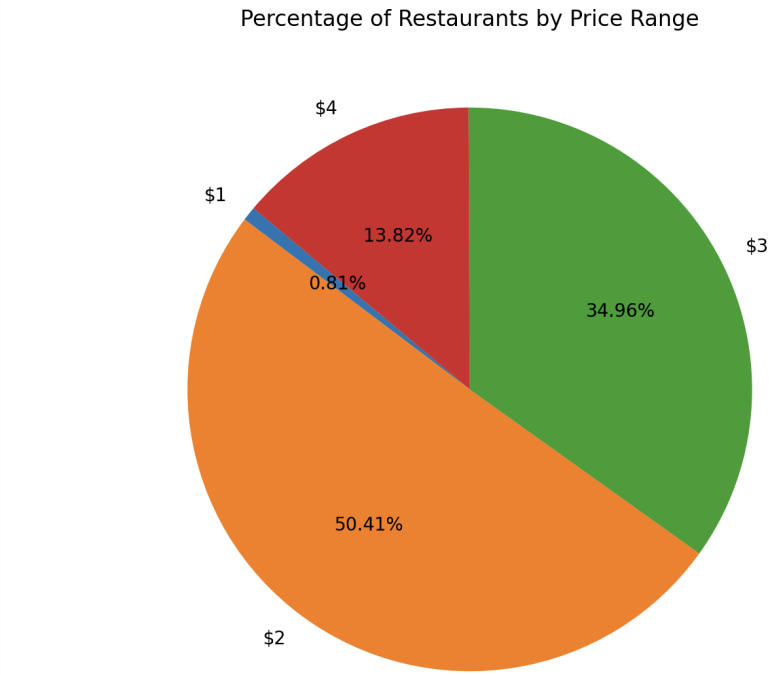
Price Range	Restaurant Count (Total: 123)	Percentage
1 ($)	1	0.81%
2 ($$)	62	50.41%
3 ($$$)	43	34.96%
4 ($$$$)	17	13.82%

price_percentages.csv

Users > jesyang > Desktop > 206project > AJS

```
1  Price Range,Count,Percentage
2  1,1,0.81%
3  2,62,50.41%
4  3,43,34.96%
5  4,17,13.82%
6
```

Visualization (Pie Chart):



Instructions for running the code:

Here are step-by-step instructions for running the code:

1. Prerequisites:
    - Ensure Python 3.x is installed on your computer
    - Required libraries:
        i. sqlite3 (built-in)
        ii. requests
        iii. matplotlib
        iv. csv (built-in)

2. Yelp API Setup:
    - Go to the Yelp Fusion Developer Portal
    - Create a developer account
    - Generate API credentials (Client ID and API Key)
    - Open `restaurants_db.py`
    - Replace the existing `CLIENT_ID` and `API_KEY` with your credentials:
      python
      Copy
      ```
      CLIENT_ID = "YOUR_CLIENT_ID_HERE"
      ```
    - `API_KEY = "YOUR_API_KEY_HERE"`

3. Run the scripts in sequence: First, run the database and data collection script (four or more times, since the code fetches restaurants in batches of 25):

```
python restaurants_db.py
```

    - This will create the database
    - Fetch restaurant data from Yelp API
    - Store data in `AJSChicago_data.db`
4. Next, calculate price percentages:

```
python calculate_prices.py
```

    - This will generate `price_percentages.csv`
5. Finally, create the visualization:

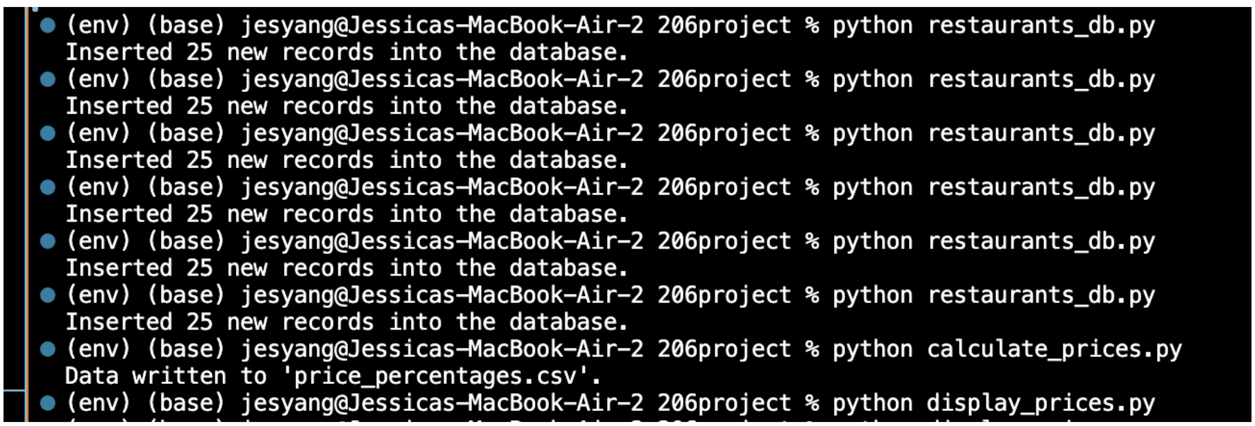
```
python display_prices.py
```

- This will show a pie chart of restaurant price distributions

6. Expected Outputs:
    - `AJSChicago_data.db` (SQLite database)
    - `price_percentages.csv`
    - Pie chart visualization

Notes:

- Each run of `restaurants_db.py` will fetch a new batch of restaurants
- The `offset.txt` file tracks progress between runs
- Duplicate restaurants are automatically prevented



Function documentation:

setup_db()	**Purpose**: Initialize SQLite database with Restaurant Price and Restaurants tables **Input**: None **Output**: Creates database tables **Side Effects**: • Creates 'AJSChicago_data.db' • Inserts default price ranges (1, 2, 3, 4)
get_current_offset()	**Purpose**: Read the current API fetch offset from 'offset.txt' **Input**: None **Output**: Integer offset value **Fallback**: Returns 0 if file not found

save_offset(offset)	**Purpose**: Save the current API fetch offset to 'offset.txt' **Input**: `offset` (integer): Current API fetch progress **Output**: None **Side Effects**: Writes offset to 'offset.txt'
get_yelp_restaurants (location, offset)	**Purpose**: Fetch restaurant data from Yelp API **Inputs**: ● `location` (string): City to search ● `offset` (integer): Starting point for API fetch **Output**: ● List of restaurant dictionaries if successful ● None if API request fails
insert_data(businesses)	● **Purpose**: Insert fetched restaurant data into SQLite database ● **Input**: ○ `businesses` (list): Restaurant data from Yelp API ● **Output**: None ● **Side Effects**: ○ Inserts restaurant data into Restaurants table ○ Converts Yelp price string to integer
main()	● **Purpose: Orchestrate database setup and data fetching** ● **Input: None** ● **Output: None** ● **Process:** 1. **Setup database** 2. **Get current offset** 3. **Fetch restaurants from Yelp** 4. **Insert data**
calculate_price_percentages()	**Purpose: Calculate percentage of restaurants in each price range**

	Input: None **Outputs:** ● **Prints confirmation message** ● **Creates 'price_percentages.csv' with:** ○ **Price Range** ○ **Count of Restaurants** ○ **Percentage of Total Restaurants**
create_pie_chart_from_csv(csv_file)	**Purpose: Create a pie chart visualization of restaurant price ranges** **Input:** ● `csv_file` **(string): Path to price percentages CSV** **Output:** ● **Matplotlib pie chart** ● **Displays chart showing percentage of restaurants by price range**

Resources used:

Date	**Issue Description**	**Location of Resource**	**Result (did it solve the issue?)**
12/12	Implementing a simple offset tracking mechanism for API data retrieval	Python File I/O Documentation, Chat GPT	Partially solved with offset.txt file
12/13	Converting Yelp's price strings to numerical values	Chat GPT	Resolved using a price_map dictionary
12/14	Generating a CSV with restaurant price range percentages	Chat GPT	Successfully implemented price percentage calculation
12/15	Adding comments to code for partners' ease of viewing	Chat GPT	Yes, added comments without changing any of the code itself

Google Places API - Sydney Emuakhagbon

1) Project Goals
    a) Main Objective: Gather and analyze data on various activities and attractions in Chicago to gain insight into popular attractions and their ratings.
    b) Planned API: The plan was to use the TripAdvisor API to gather information on tourist attractions, restaurants, and activities. However, due to monetary requirements and a limited trial that required inputting my card information, I was unable to fully use this API as expected.
    c) Goal Data to Gather:
        i) Attraction Name
        ii) Category (e.g., sightseeing, adventure)
        iii) Rating
        iv) Number of reviews
2) Project Achievements
    a) API Used: Despite initially planning to use the TripAdvisor API, I switched to the Google Places API due to the monetary restrictions of TripAdvisor. Although I still had to input my card information, the Google Places API effectively allowed me to gather data on various types of attractions in Chicago, such as museums, parks, and restaurants.
    b) Data Stats Gathered
        i) Attraction Name
        ii) Attraction Category
        iii) Average User Rating of Attraction
        iv) Attraction User Review Count: The total number of user reviews.
        v) Attraction Address
3) Problems
    a) Monetary restrictions in getting the data we needed for our vacation-themed data pool.
    b) Each API request returned a subset of the total results, and I had to manage pagination by handling the next_page_token, requiring me to make multiple API requests and collect the data incrementally. There was some difficulty in managing the timing of requests, as the next_page_token would only work after a short delay (2-3 seconds) to avoid issues with the rate-limiting of the API.
    c) To mitigate duplicate entries, I used INSERT OR IGNORE, helping to avoid inserting the same attraction multiple times.
    d) The data returned from the Google Places API was sometimes incomplete, such as not including user ratings. Therefore, my code required additional logic to handle these cases, ensuring the program didn't break and default values were set appropriately.

e) One of the most challenging aspects was ensuring that the program correctly inserted a maximum of 25 entries into the database. I had to repeatedly refine the code to handle different scenarios and ensure that each run of the program would stop when the total number of inserted entries exceeded 25, while ensuring each run returned at least one entry. It was frustrating to balance between fetching enough data and making sure the program didn't exceed the 25-entry limit, leading to multiple rounds of testing and debugging.

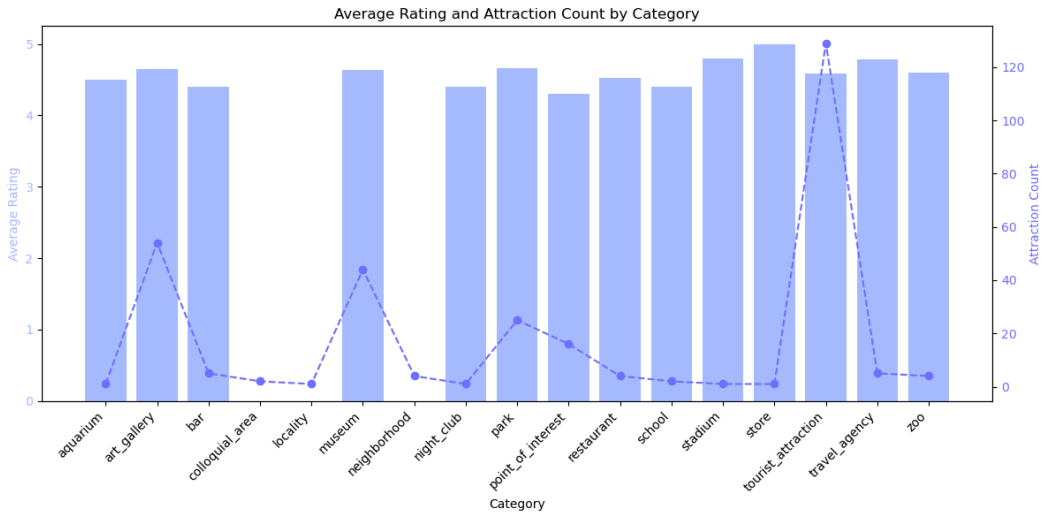
4) Calculation

Attraction Data Summary

Category	Average Rating	Attraction Count
aquarium	4.50	1
art_gallery	4.64	54
bar	4.40	5
colloquial_area	0.00	2
locality	0.00	1
museum	4.63	44
neighborhood	0.00	4
night_club	4.40	1
park	4.66	25
point_of_interest	4.31	16
restaurant	4.53	4
school	4.40	2
stadium	4.80	1
store	5.00	1
tourist_attraction	4.58	129
travel_agency	4.78	5
zoo	4.60	4

a)

5) Visualization



a)

6) Code Instructions
   a) Environment Requirements
      i) Ensure the following is installed:
         (1) Python 3
         (2) requests (for making API calls)
         (3) sqlite3 (for database interaction)
         (4) matplotlib (for data visualization)
         (5) csv (for handling CSV files)
   b) Obtain A Google Places API Key
      i) Sign up for a Google Cloud account (if you don't already have one) at Google Cloud Console and enable the Places API.
      ii) Obtain an API Key and place it where within the variable API_KEY in the code. My API key was "AIzaSyB--RDZw6LYLq0-23rngVX0cUt9idBObv4".
   c) Set Up the Database
      i) Ensure the database is created and data is fetched successfully, within the confines of this project, the database is "AJSChicago_data.db"
   d) Run the Main Program
      i) Execute the main() function to fetch and store data from the Google Places API
         (1) Functions of the main function:
            (a) Query various categories of places around Chicago.
            (b) Get data from the Google Places API.
            (c) Store the relevant information in the database.
   e) Storing the Data
      i) After fetching and processing the data, it will be stored in the AJSChicago_data.db database. SQLite browser tools can be used to inspect the stored data.

f) Data Visualization (OPTIONAL)
    i) Use the code for creating bar charts with matplotlib to generate visualizations based on the data stored in the database. The visual will provide insight into the average rating and count of attractions per category.
        (1) generate_bar_chart() will visualize the results
g) Function Documentation
    i) setup_db()
        (1) Purpose: Setting up the SQLite database and creating the Itinerary table, if it doesn't already exist. The table stores information about places to visit in Chicago.
        (2) Input: None required
        (3) Output: No return value since this function creates the table Itinerary in the database.
    ii) get_google_places_data(api_key, location="Chicago", page_token=None)
        (1) Purpose: Get data from the Google Places API for a specified location. Returns information about places based on the provided query, which is hardcoded to return Chicago-based results.
        (2) Input:
            (a) api_key (string): Your Google Places API key.
            (b) location (string): The location to search for places. Defaults to "Chicago".
            (c) page_token (string, optional): Used to fetch the next page of results in case of pagination. Default as None.
        (3) Output: Returns a dictionary containing the results from the API, which includes information about the places, or None if an error occurs.
    iii) store_data_in_db(data, remaining_entries)
        (1) Purpose: stores data gathered from the Google Places API into the Itinerary table in the database.
        (2) Input:
            (a) data (dictionary): The data returned from the Google Places API, containing information about places.
            (b) remaining_entries (int): The number of entries that have yet to be inserted into the database, ensuring no more than the specified number of entries are added (hardcoded to be 25).
        (3) Output: Returns the number of new entries successfully inserted into the database.
    iv) main()

(1) Purpose: Main function of the program, as it manages the gathering of data, storing it in the database, and generating visualizations.
(2) Input: No explicit inputs. The function initiates a series of queries to gather data from the Google Places API.
(3) Output: No return value. The function coordinates the entire process of querying, storing, and visualizing the data.

v) write_processed_data_to_csv()
(1) Purpose: Exports processed data, namely the average rating and attraction count by category, from the database to a CSV file.
(2) Input: No explicit inputs. The function retrieves data from the Itinerary table in the database.
(3) Output: Writes the processed data to a CSV file named places_data.csv

vi) plot_data()
(1) Purpose: Gathers data from the database and generates a dual-axis plot to visualize the average ratings and attraction count for each category. The plot displays the average ratings as bars and the attraction count as a line graph.
(2) Input: No explicit inputs. The function retrieves data from the Itinerary table in the database.
(3) Output: Displays a dual-axis plot with the average rating on the left y-axis (as bars) and attraction count on the right y-axis (as a line graph). No return value.

h) Resource Documentation

i)

Date	Issue Description	Location of Resource	Result
12-13-2024	Encountering monetary requirements and request limitations for the TripAdvisor API trial	TripAdvisor API Website	No, ultimately could not use the API due to cost and request constraints.
12-13-2024	Struggling to query and fetch data from Google Places API	Google Places API Documentation	Yes, helped in understanding how to use the API for retrieving place data.
12-15-2024	Correctly handling	ChatGPT	Yes, aided in

	pagination and data limits in API responses.		debugging ro correctly handle pagination in the API for gathering multiple results.
12-15-2024	Needed to gather only a specific number of records from the API (25).	ChatGPT	Yes, provided a solution on how to limit the number of records from an API query.
12-16-2024	Understanding how to work with the matplotlib library for creating data visualizations.	W3Schools	Yes, provided the necessary information to understand how to plot dual-axis charts and customize them.