Github link

1. The goals for your project, including what APIs/websites you planned to work with and what data you planned to gather (10 points)

The initial idea was to look at used car sale data to see if differing weather conditions in different parts of the country could have a noticeable effect on depreciation value by using a weather API and collecting car data. The weather API we were going to use initially was the Visual Crossing free weather API Visual Crossing. The plan was to use the Kelley Blue Book car API Kelly API to collect the data on used cars. Once we had collected data from both of these locations, we organized them into a database. Then we graphed them to test our hypothesis that more extreme weather conditions on either end of the scale would lead to higher levels of car depreciation. We had not fully decided when it came to the specific data we were going to gather, but we planned on starting with climate data like temperature, humidity, precipitation, and snow, along with car data from different models of cars from different cities around the country.

2. The goals that were achieved,d including what APIs/websites you actually worked with and
what data did youd gather (10 points)

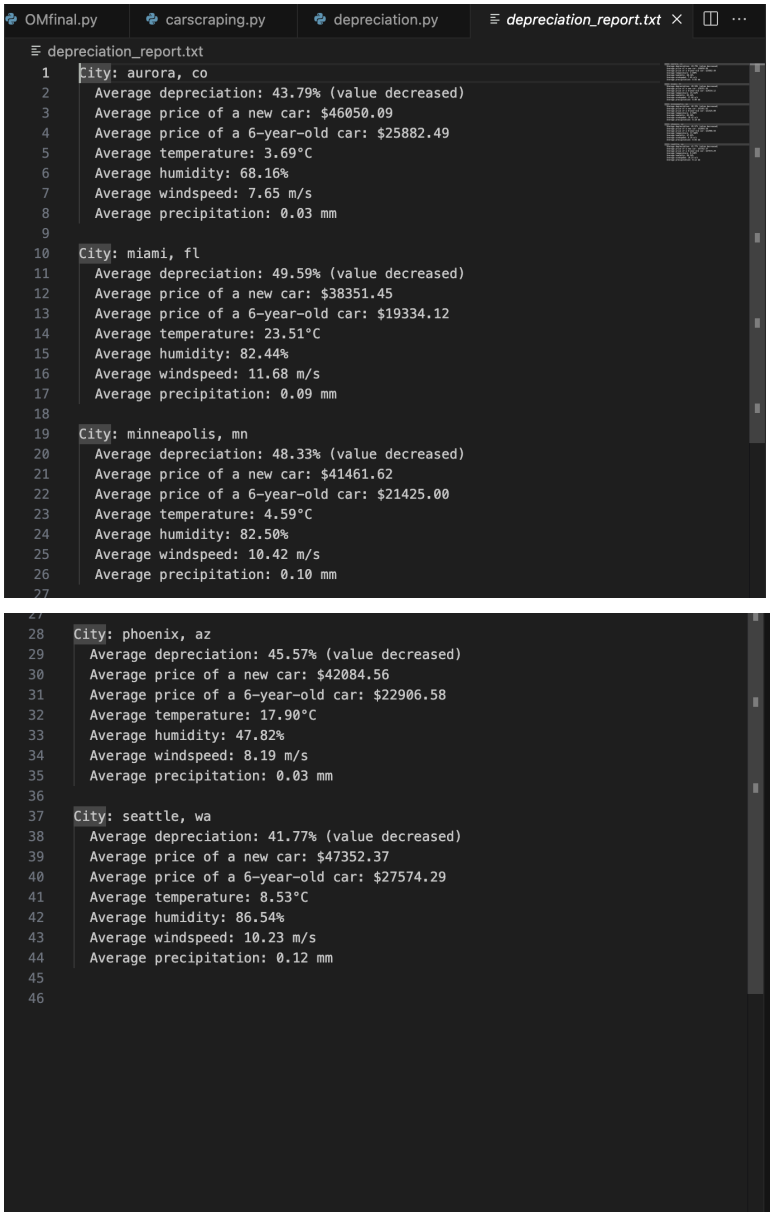
When enacting our plan, a few problems with where we would collect our data arose pretty quickly. Firstly, the Kelley Blue Book API seemed to require that you have a VIN number to gain any data on cars. To solve the car data problem, we used web scraping on the Kelley Blue Book website instead of the API to gather data. This was successful, and we gathered data on different car models around different cities in the country. When it came to the weather data, the visual Crossing API had unclear documentation on what the free and paid versions allowed you to gather, leading to us having to submit a ticket and then later learning that we couldn't access the exact data we needed. To pivot, we had to find a second API that would work to collect our weather data, and that's where we found https://open-meteo.com/, which served as a much easier source of our data that would have fewer issues than the previous API. With a working weather API, we gathered data on temperature, humidity, and precipitation through five cities.

Now, the car data and the weather data were collected in a database, sharing the locations where the data was collected and the time period, allowing for the data to be calculated and compared with itself. Once we had the data, we were able to calculate the averages on depreciation and for the weather; with the averages, we were able to finally make the three matplotlib graphs following average depreciation in all locations on one axis and then average temperature precipitation and humidity on the other access for the three graphs. Once we tweaked our graphs to be more presentable and easier to understand, we had our final output from our collected data. With this data, we were able to see that there were some patterns when it came to average temperature, humidity and precipitation, leading to places that shared in the extremes having higher depreciation values.

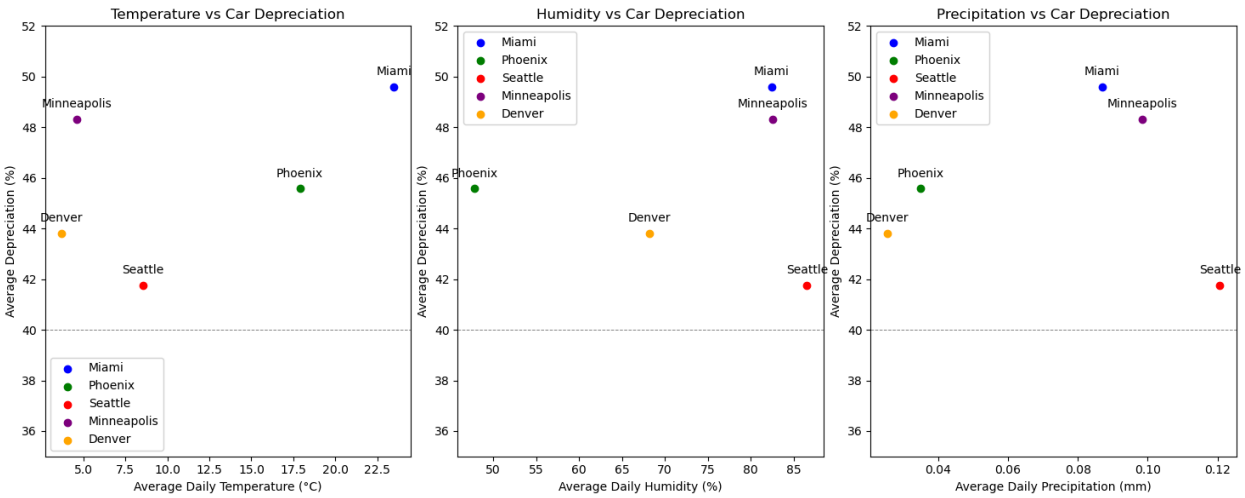
3. The problems that you faced (10 points)

1. The first issue we faced was the weather API not working, as mentioned, but once we moved on to the other API it functioned properly.
2. We ran into an issue where the car database collected data on Denver under the name Aurora while the weather database didn't, needing us to change how it calls in the data.
3. Gathering 25 data points at a time for the weather data was difficult ,but after some debugging we found a way to have it add with the proper limitations instead of throwing in all the data at once
4. When trying to create a plot for the precipitation and depreciation we realized that the weather data being collected was only from a couple days instead of the whole range we wanted. To fix this we implemented that it gathered all the data instead of only 125 rows and this allowed us to gather all of the weather data correctly.

4. The calculations from the data in the database (i.e. screenshot) (10 points)



5. The visualization that you created (i.e. screenshot or image file) (10 points)



6. Instructions for running your code (10 points)

1. Open a terminal window and run the following line to make the open metro API work.
    a. **pip install openmeteo-requests requests-cache retry-requests pandas**
2. Run carscraping.py
    a. Not ideal but it must run multiple times in order for all the data to be collected (our final database has 484 rows of prices)
3. Run OMfinal, py
    a. Run multiple times until roughly 12,000 lines are in the hourly_climate table
4. Run depreciation.py
5. Optionally check the database on DB
6. Run analysis.py
7. View the graphs

7. Documentation for each function that you wrote. This includes describing the input and
output for each function (20 points)

1. **OM final**

initialize_db()

Purpose: Creates a SQLite database and three tables (cities, hourly_climate, and city_averages).
Input: None
Output: None

get_city_id(city_name)

Purpose: Fetches the city_id of a city by its name from the cities table.
Input: city_name (string) - The name of the city whose ID is needed.
Output: city_id (integer) - The ID of the specified city.

insert_data_to_db(city_id, data)

Purpose: Inserts weather data into the hourly_climate table for a specific city identified by city_id. Ignores duplicate data based on city_id and date.
Input: city_id (integer), data (DataFrame) - The ID of the city and the DataFrame containing the weather data to be inserted.

Output: None

fetch_weather_data(latitude, longitude)

Purpose: Fetches weather data from the Open-Meteo API for a specific latitude and longitude range. The data includes temperature, humidity, wind speed, and precipitation.

Input: latitude (float), longitude (float) - The coordinates of the location for which weather data is required.

Output: hourly_data (DataFrame) - A DataFrame containing the fetched weather data.

main()

Purpose: Orchestrates the process of initializing the database, inserting city information, fetching weather data, and inserting climate data. It handles the overall flow of the script.

Input: None

Output: None

Carscraping.py

scrape_car_data()

Purpose: Utilizes BeautifulSoup to scrape car data, particularly prices, from the Kelley Blue Book website.

Input: Accepts parameters including the car's make, model, year, as well as the city, state, and zip code where the car is located. These parameters are used to construct the URL for accessing the specific car page within the designated city.

Output: Returns a dictionary containing the car's make, model, and year, along with a list of prices scraped for that car within the specific city.

setup_database()

Purpose: Sets up the database by creating several tables: cities for storing specified cities, cars for storing specified car details, prices for holding pricing data associated with cars and cities, and car_depreciation for storing depreciation calculations from depreciation.py.

Input/Output: This function does not take any inputs or produce any outputs.

store_car_and_city()

Purpose: Stores car and city information into their respective database tables. It takes in car_data from the scrape_car_data() function, as well as the city, state, zip code, latitude, and longitude. The function returns the shared integer keys car_id and city_id, which will be utilized by the store_prices() function.

Input/Output: Takes in car and city details, and outputs car_id and city_id.

store_prices()

Purpose: Separates the process of storing prices to facilitate batch insertion of up to 25 items at a time into the database. It accepts car_id and city_id from the store_car_and_city() function and stores them along with the list of corresponding prices.

Input: Takes car_id, city_id, and the list of prices.

Output: This function does not produce an output.

main()

Purpose: Integrates all the functions above and manages the specified cars and cities intended for processing. It limits database entries to 25 per run.

Input/Output: This function does not take any inputs or produce any outputs.

Depreciation.py

get_average_price_by_year_and_city()

Purpose: Fetches and calculates the average price of cars for a specific year and city.

Input: Takes the year of the car and city_id as input parameters.

Output: Returns a float representing the average price of cars for the specified year and city. If no prices are available, it returns None.

calculate_average_depreciation_by_city()

Purpose: Computes the average depreciation of cars across various cities based on their prices from different years (2018-2024).

Input: None.

Output: Outputs a dictionary where each key is a city, and each value is a tuple containing the depreciation percentage, average price of new cars, and average price of old cars.

store_depreciation_data()

Purpose: Stores the calculated depreciation data into the database for future use in visualizations.

Input: Accepts the dictionary outputted by the calculate_average_depreciation_by_city() function.

Output: This function does not produce an output.\

store_average_weather()

Purpose: Calculates and stores the average weather data, including temperature, humidity, wind speed, and precipitation for each city.

Input/Output: This function does not take any inputs or produce any outputs.

fetch_weather_data_by_city()

Purpose: Helper function

main()

Purpose: Integrates and runs all the functions to calculate and store the necessary data.

Input/Output: This function does not take any inputs or produce any outputs.

analysis.py

normalize_city_names()

Purpose: Simplifies city names in a data frame to lowercase strings to standardize and facilitate easier data processing when creating visualizations.

Input: Accepts a data frame and the column containing city names.

Output: Modifies the data frame in place by converting city names to lowercase.

map_city_names()

Purpose: Replaces suburb names with their better-known city counterparts in the dataset for improved readability in visualizations. For instance, replaces 'Aurora' with 'Denver' when dealing with Denver, Colorado.

Input: The data frame with city names.

Output: Modifies the data frame in place by replacing specific suburb names with their corresponding major city names.

fetch_combined_data()

Purpose: Pulls and combines the calculated data from different tables in the database.

Input: Accepts the path to the database.

Output: Returns a data frame containing the combined data, utilizing previously defined functions to format and structure the data appropriately.

plot_data()

Purpose: Generates scatter plots that compare temperature, humidity, and precipitation against the average depreciation rate in each selected city using the data from the database.

Input: None explicitly, but operates on the processed data fetched by fetch_combined_data().

Output: Produces and displays scatter plots for the specified comparisons.

8. You must also clearly document all resources you used. The documentation should be of
the following form (20 points)

Date	Issue description	Location of resource	Result
12/1/24	Visual crossing not allowing data collection	Open metro weather API	Working API for weather data
12/2/24	OMfinal not properly gathering 25 data points at a time	Had to consult UMgpt to fix it, gathering 25 for each city but doing it all in one run	Gathers data as requested in instructions
12/1/24	Needed a regex function to help clean data	Found regex sheet in canvas and used regex 101 website.	Was able to get a working regex function to clean pricing data
12/1/24	Had trouble making a loop that correctly inserted 25 rows of pricing data at a time	Consult UMgpt	Fixed my loop to correctly get 25 rows per run of the file carscraping.py
12/2/24	Becuase I had to use a 2023 tesla instead of 2024 like all other cars I had trouble calculating depreciation specifically with the loop.	Consult UMgpt	Added an if statement to my loop that allowed the function to work correctly for 2024 and 2023 car models

12/5/24	Realized we had 2 databases (one for each api/website) adn we needed to merge it into one while reducing duplicate string data. This was an issue because each database had a cities table so I needed to figure out how to merge these into one.	Consulted this stack overflow message chain https://stackoverflow.com/questions/80801/how-can-i-merge-many-sqlite-databases	Was able to get both databases into one and merge the cities table while keeping most of the code relatively the same
12/15/24	Although we stored the calculations from our data into our database I needed to get it into a txt file but couldn't remember how to do this.	Found this website that helped me get started with the txt file https://www.w3schools.com/python/python_file_write.asp	Now the calculations are correctly written to a txt file.