

Report Final Project

Project Name: APIs, SQL, and Visualizations

1. The goals for your project include what APIs/websites you planned to work with and what data you planned to gather.

The primary goal of this project was to analyze the relationship between weather conditions and air quality across specific locations, using APIs and databases to gather real-time and historical data. This analysis will provide insights into how weather patterns influence air quality and pollution levels in cities of Detroit.

We used the AirVisual (Air quality) API for the purpose of gathering real-time and historical air quality data for specific cities, we choose Detroit and pointed are attention on data such as Air Quality Index (AQI), main pollutants (e.g., PM2.5, PM10, O3), and AQI categories. Second, we worked with the weather API to retrieve real-time and historical weather data for the same cities as the air quality data, focused on temperature, humidity, wind speed, and weather conditions (e.g., sunny, rainy, cloudy). This project provides actionable insights by integrating air quality and weather data through APIs and visualizations, contributing to understanding environmental impacts on urban health.

Data plan to gather:

- AirQuality
 - Id
 - Date and time
 - Air Quality Index (AQI)
 - Main pollutant (e.g., PM2.5 and PM10)
 - Location (city, region, country)
- Weather API:
 - Id
 - Date and time
 - Temperature (°C)
 - Weather condition (e.g., sunny, cloudy, rainy)
 - Humidity
 - Wind speed
 - Location (city, region, country)

2. The goals that were achieved including what APIs/websites you actually worked with and what data you did gather

Goals that were achieved :

- We successfully created data from scratch, gathered and analyzed data to explore the relationship between weather conditions and air quality across specific locations.
- Created a structured SQLite database (WeatherAirQuality.db) to store the collected data, facilitate integration and retrieval for analysis and visualization.

APIs/Websites Worked with and gather data:

☐ API Air Quality:

- Hourly AQI values mapped to categories (Good, Moderate, etc.).
- Main pollutants contributing to air quality degradation (e.g., PM2.5, CO).
- Date and time
- Location
- Id

☐ API Weather

- Hourly temperature and humidity percentages.
- Wind speed and direction.
- General weather conditions (e.g., clear skies, rain).
- Date and time
- Location
- Id

3. The problems that you faced

- Rate limits: We encountered limitations on the number of API calls allowed per day, with the Air quality AP we had to pull twenty four raw by day and using five days we had in total one hundred twenty data in the database.
- Authentication Errors: Faced 403 Forbidden errors due to invalid or revoked API keys, necessitating key regeneration and verification (Also we had a challenge with the API key we used first was not pulling data, after changing another API and fixing the code we were able to run the code).
- Data Inconsistencies: Some hourly data was unavailable or incomplete from the APIs, leading to gaps in the dataset.
- Dysfunctioning of some APIs: API Spotipy did not work, so we changed to Air quality.

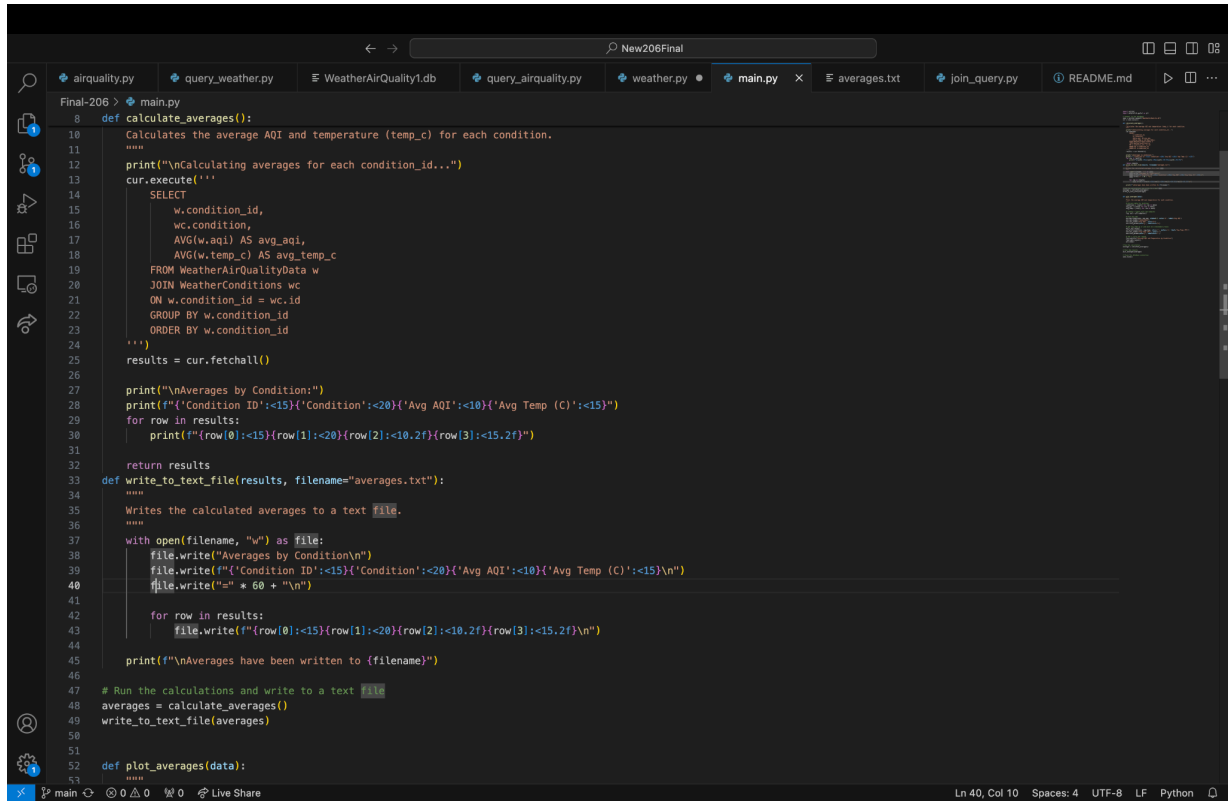
- File Path Errors: The SQLite database (WeatherAirQuality.db) was not found in the working directory at times, causing issues with data storage. This required specifying absolute paths to the database.
- Time Constraints: Managing the simultaneous demands of data collection, storage, integration, and visualization within the project timeline was challenging, especially when facing API problems and pulling data.

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

WeatherAirQualityData.db (WeatherAirQualityData)

id	hour	temp_c	wind_mph	humidity	aqi	main_pollutant	condition_id
1	1 00:00	1.8	7.8	97	65	p2	2
2	2 00:00	1.8	7.8	97	65	p2	2
3	3 00:00	1.8	7.8	97	65	p2	2
4	4 00:00	1.8	7.8	97	65	p2	2
5	5 00:00	1.8	7.8	97	65	p2	2
6	6 01:00	1.1	8.3	97	65	p2	2
7	7 01:00	1.1	8.3	97	65	p2	2
8	8 01:00	1.1	8.3	97	65	p2	2
9	9 01:00	1.1	8.3	97	65	p2	2
10	10 01:00	1.1	8.3	97	65	p2	2
11	11 02:00	0.8	8.9	97	65	p2	2
12	12 02:00	0.8	8.9	97	65	p2	2
13	13 02:00	0.8	8.9	97	65	p2	2
14	14 02:00	0.8	8.9	97	65	p2	2
15	15 02:00	0.8	8.9	97	65	p2	2
16	16 03:00	0.6	8.7	97	65	p2	2
17	17 03:00	0.6	8.7	97	65	p2	2
18	18 03:00	0.6	8.7	97	65	p2	2
19	19 03:00	0.6	8.7	97	65	p2	2
20	20 03:00	0.6	8.7	97	65	p2	2
21	21 04:00	1.2	8.6	97	65	p2	2
22	22 04:00	1.2	8.6	97	65	p2	2
23	23 04:00	1.2	8.6	97	65	p2	2
24	24 04:00	1.2	8.6	97	65	p2	2
25	25 04:00	1.2	8.6	97	65	p2	2
26	26 05:00	1.3	8.7	83	65	p2	3
27	27 05:00	1.3	8.7	83	65	p2	3
28	28 05:00	1.3	8.7	83	65	p2	3
29	29 05:00	1.3	8.7	83	65	p2	3
30	30 05:00	1.3	8.7	83	65	p2	3

- ★ This dataset WeatherAirQuality prints the calculation of average Air Quality Index (AQI) and temperature (temp_c).

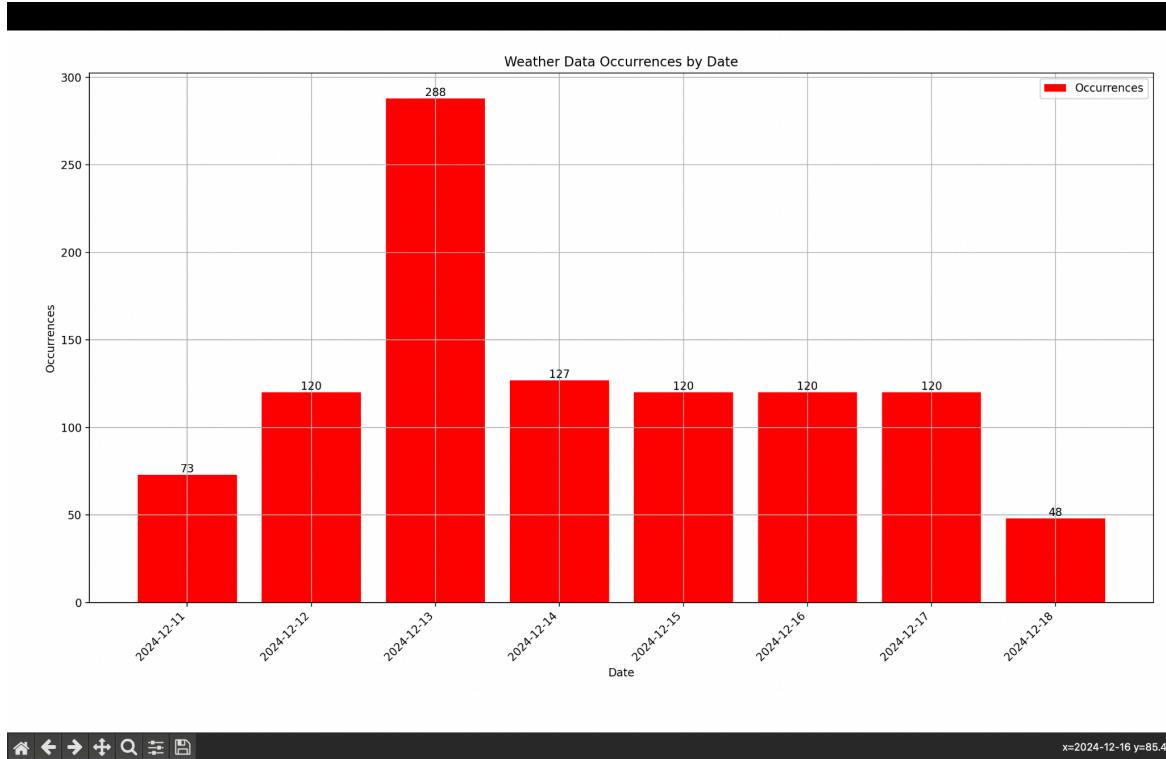


```
Final-206 > main.py
8 def calculate_averages():
9     """
10     Calculates the average AQI and temperature (temp_c) for each condition.
11     """
12     print("\nCalculating averages for each condition_id...")
13     cur.execute("""
14         SELECT
15             w.condition_id,
16             wc.condition,
17             AVG(w.aqi) AS avg_aqi,
18             AVG(w.temp_c) AS avg_temp_c
19         FROM WeatherAirQualityData w
20         JOIN WeatherConditions wc
21         ON w.condition_id = wc.id
22         GROUP BY w.condition_id
23         ORDER BY w.condition_id
24     """)
25     results = cur.fetchall()
26
27     print("\nAverages by Condition:")
28     print(f"{'Condition ID':<15}{'Condition':<20}{'Avg AQI':<10}{'Avg Temp (C)':<15}")
29     for row in results:
30         print(f"{row[0]:<15}{row[1]:<20}{row[2]:<10.2f}{row[3]:<15.2f}")
31
32     return results
33
34 def write_to_text_file(results, filename="averages.txt"):
35     """
36     Writes the calculated averages to a text file.
37     """
38     with open(filename, "w") as file:
39         file.write("Averages by Condition\n")
40         file.write(f"{'Condition ID':<15}{'Condition':<20}{'Avg AQI':<10}{'Avg Temp (C)':<15}\n")
41         file.write(" " * 60 + "\n")
42         for row in results:
43             file.write(f"{row[0]:<15}{row[1]:<20}{row[2]:<10.2f}{row[3]:<15.2f}\n")
44
45     print(f"\nAverages have been written to {filename}")
46
47 # Run the calculations and write to a text file
48 averages = calculate_averages()
49 write_to_text_file(averages)
50
51
52 def plot_averages(data):
53     """
```

- ★ Code that calculates the average Air Quality Index (AQI) and temperature (temp_c) for each weather condition, grouped by condition_id. (main.py)

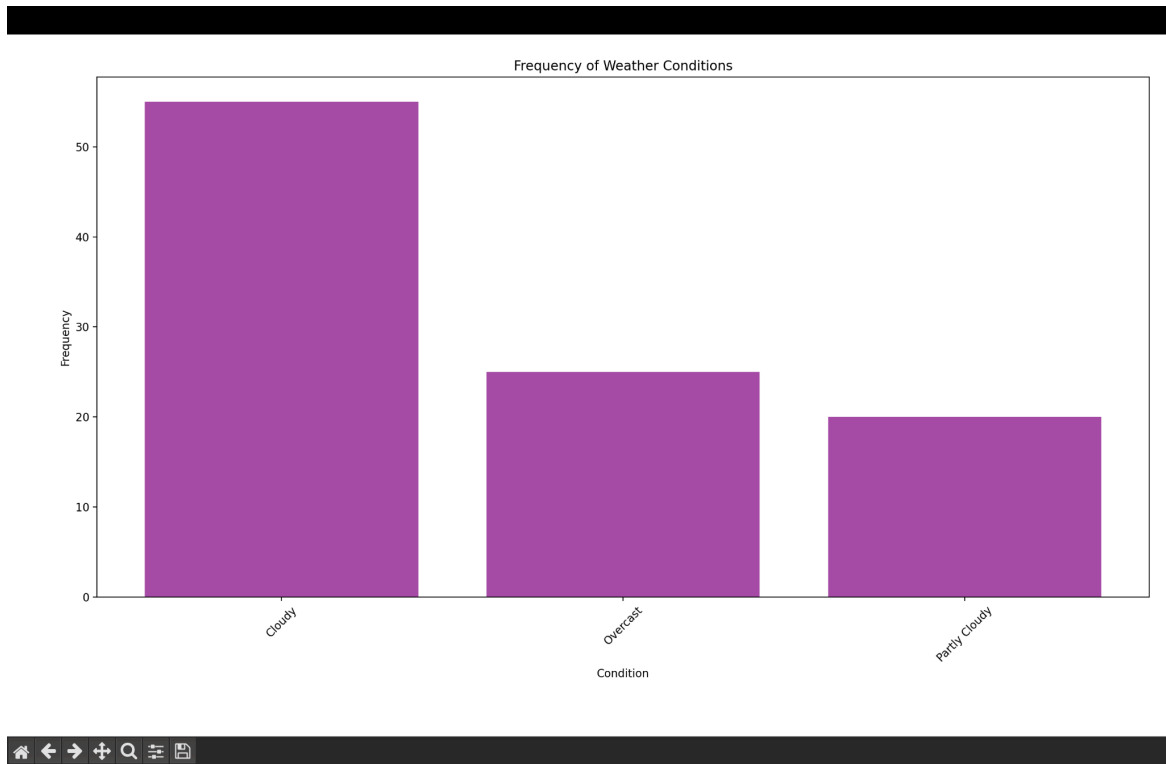
5. The visualization that you created (i.e. screenshot or image file)

1. Weather visualization (query_weater.py)



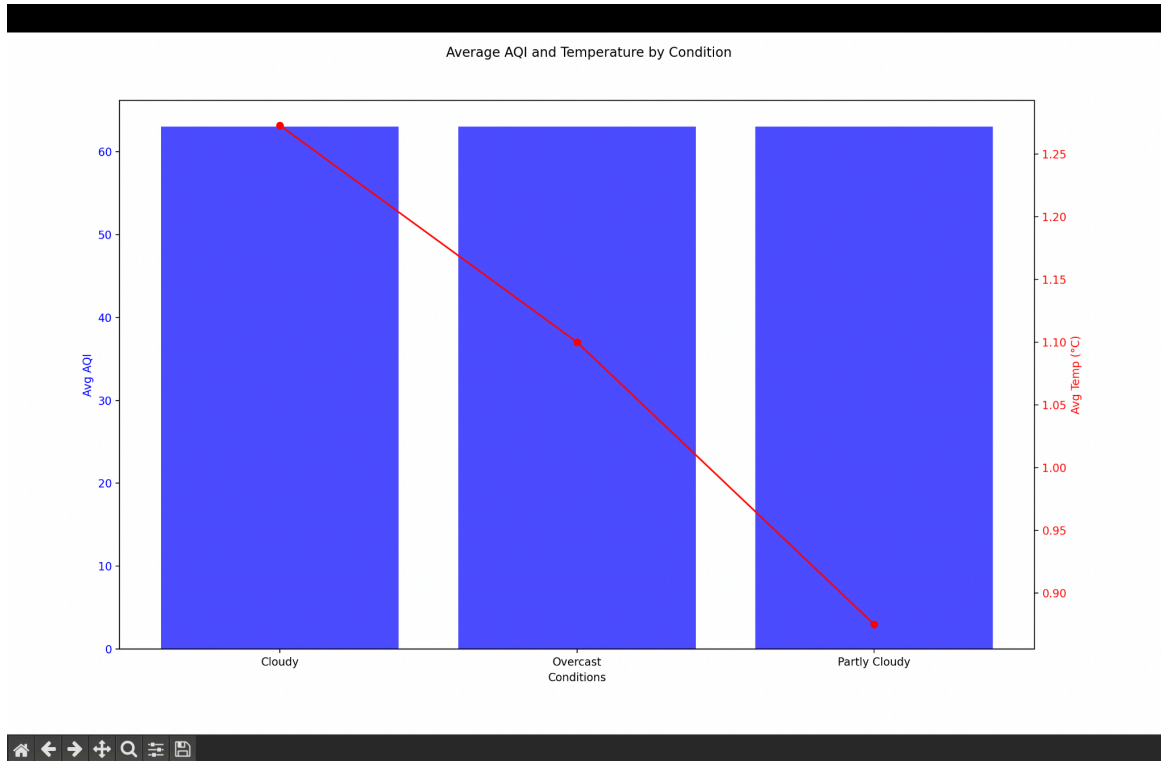
- ★ This code demonstrates how to count the occurrences of weather data by date from a SQLite database and visualize the results using a bar chart with Matplotlib.

2. Weather conditions (*query_airquality.py*)



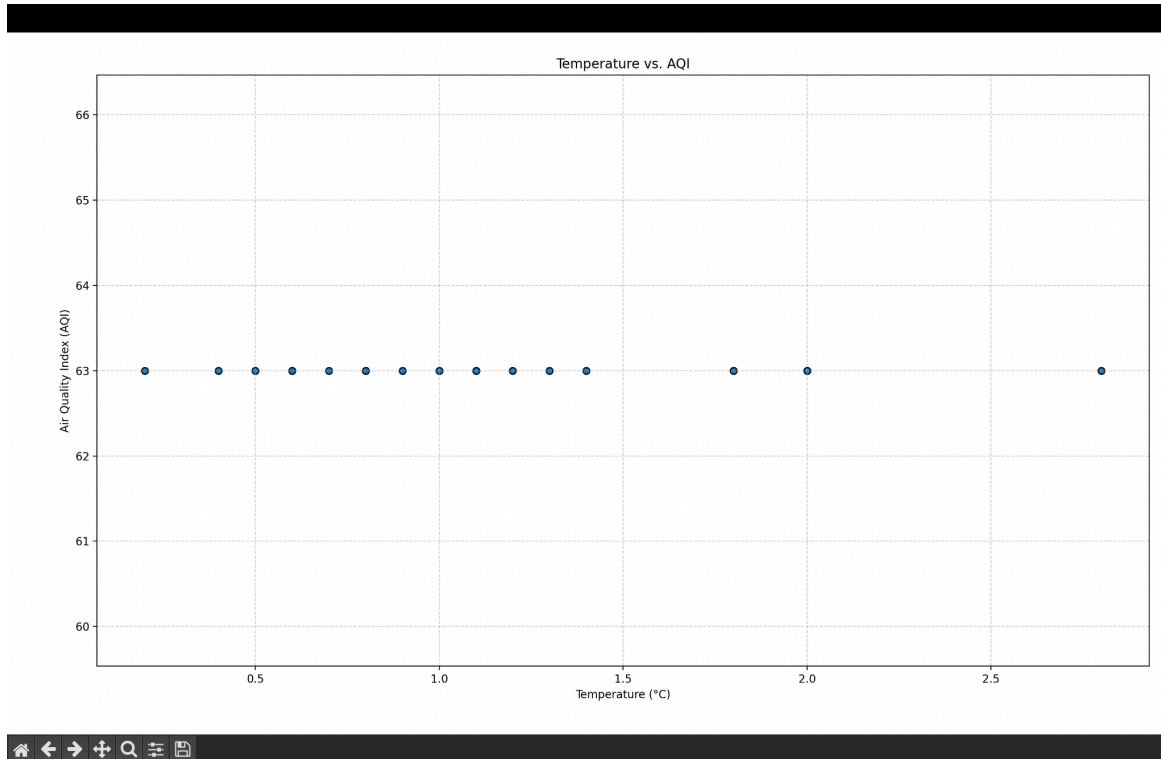
★ This plots a bar chart showing the frequencies of different weather conditions.

3. Temperature by condition (*WeatherAirQuality.db*)



- ★ The graph compares two metrics—average Air Quality Index (AQI) and average temperature (°C)—across different weather conditions. It provides a visual representation of how air quality and temperature vary for each condition.

2. Air quality visualization (*query_airquality.py*)



- ★ This code fetches the temperature and AQI (Air Quality Index) data from the WeatherAirQualityData table in the SQLite database.

6. Instructions for running your code

With the GitHub link submitted on Canvas, clicking the link will take you to the repository. Inside, you'll find eight Python files and one database dataset. To explore the project, clone the repository and open it in Visual Studio Code (VSCoDe). By running the Python scripts, you can see how the code works and observe the outputs directly.

File : Weather.py, Airquality.py, Query_airquality.py, Query_weater.py, Join_query.py, Join_query.py, Main.py, WeatherAirQuality.db.

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

1. **Weather.py:** This script fetches hourly weather data from the WeatherAPI and stores it in a SQLite database (WeatherAirQuality.db). It creates a separate table for each date in the specified range (2024-12-02 to 2024-12-06) and limits data insertion to 25 rows per day.

Function:

- **a. def create_date_table(date):** Creates a table in the SQLite database for a specific date if it doesn't already exist.
 - ☐ Description: A dynamically generates a table name based on the input date. Creates a table in the database with columns: id, hour, temp_c, condition, wind_mph, humidity. If the table already exists, it skips creation.
 - ☐ Inputs: date (str): A string representing the date in the format YYYY-MM-DD. This is used to create a table name and store weather data for that specific date.
 - ☐ Outputs: Returns the name of the table created or validated (e.g., "WeatherData_2024_12_02").
- **b. def fetch_weather_data(location, api_key, start_date, end_date):** Fetches hourly weather forecast data for a given location and date range and stores it in a SQLite database.
 - ☐ Description: Repeat through the date range from start_date to end_date and send an API request to WeatherAPI to fetch hourly weather data for the specified location and date. For each date we create a table for the date using create_date_table, inserting at least 25 rows of hourly weather data into the table. Each row contains: hour, temp_c, condition, wind_mph, and humidity.
 - ☐ Inputs: location, api_key, start_date, and end_date
 - ☐ Outputs: (None) The function writes data to dynamically created tables in the SQLite database. It does not return a value.

2. Airquality.py:

- **a. def create_date_table(date)**

Description:

Creates a table in the SQLite database for a specific date if it does not already exist. The table name is dynamically generated based on the given date.

Inputs: date (str): The date in YYYY-MM-DD format.

Outputs: Returns the name of the created or existing table (str).

- **b. def aqi_to_category(aqi)**

Description: Maps Air Quality Index (AQI) values to predefined categories based on air quality standards.

Inputs: aqi (int): The AQI value.

Outputs: Returns an integer (1-5) corresponding to the AQI category:

- 1: Good ($AQI \leq 50$)
- 2: Moderate ($51 \leq AQI \leq 100$)
- 3: Unhealthy for Sensitive Groups ($101 \leq AQI \leq 150$)
- 4: Unhealthy ($151 \leq AQI \leq 200$)
- 5: Very Unhealthy ($AQI > 200$)

- **c. def fetch_air_quality_data(city, state, country, api_key, current_date)**

Description: Fetches real-time air quality data from the AirVisual API for a given city, state, and country. Simulates hourly data storage in the SQLite database with a maximum of 25 entries per run. If a table for the date exists and has all 24 hourly entries, no further action is taken.

Inputs: city (str), state (str), country (str), api_key (str), current_date (datetime)

Outputs: Stores air quality data (hour, AQI category, main pollutant) in a SQLite table for the given date and prints log messages for each step and status of the operation.

3. Query_airquality.py:

Function:

a. def fetch_condition_data

Description: Fetches the frequency of different weather conditions from the WeatherAirQualityData table in the SQLite database.

Input: None

Output: List of Tuples: Each tuple contains a condition_id (int) and count (int): The frequency of the respective condition in the database.

SQL Query Explanation: the groups rows by condition_id and counts the number of occurrences for each condition_id and orders the results by condition_id.

b. def plot_condition_frequencies

Description: Plots a bar chart showing the frequencies of different weather conditions.

Input: data (list of tuples): The output of fetch_condition_data. Each tuple contains: condition_id (int) and count (int)

Output: None, Displays a bar chart.

Steps: Extracts condition IDs and their corresponding counts from the input data, maps condition_id to human-readable condition names using a predefined dictionary, and creates a bar chart with condition names as labels and counts as bar heights.

c . def fetch_temp_aqi_data

Description: Fetches the temperature and AQI (Air Quality Index) data from the WeatherAirQualityData table in the SQLite database.

Input: None

Output: List of Tuples: Each tuple contains (temp_c (float), aqi (int))

SQL Query Explanation: Selects rows where both temp_c and aqi are not NULL to ensure valid data for plotting.

d . def plot_temp_vs_aqi

Description: Plots a scatter plot of temperature (°C) versus Air Quality Index (AQI).

Input: data (list of tuples): The output of fetch_temp_aqi_data. Each tuple contains(temp_c,aqi (int))

Output: None: Displays a scatter plot.

Steps: Extracts temperatures and AQI values from the input data, creates a scatter plot with temperatures on the x-axis and AQI values on the y-axis, and enhances visualization with grid lines, axis labels, and a title.

4. Query_weather.py:

Description: Takes data from the weather table in the database and creates a bar graph, from the data.

Input : Data from the dataset in SQLite

Output: A bar graph

Steps: Initializes results to store data for visualizations, executes the query, and displays the visualizations, print exceptions if it does not work.

a.def count_weather_data_by_date

Description: This function retrieves and counts the number of weather data records for each date from the WeatherData table in the WeatherAirQuality.db SQLite database. It then prints the counts for each date and visualizes the data using a bar chart with a legend. The chart includes value labels on top of each bar for clarity.

Inputs:

This function does not take any external inputs. It operates directly on the SQLite database WeatherAirQuality.db.

Outputs:

- **Console Output:**
 - Prints the number of records (occurrences) for each date in the WeatherData table.
 - Displays any error messages in case of exceptions.
 - Provides logs about the connection status, query results, and function status.
- **Visualization:**
 - Generates a bar chart showing the number of occurrences for each date:
 - ☐ X-axis: Dates (rotated for readability).
 - ☐ Y-axis: Number of occurrences.
 - ☐ Bars labeled with the exact counts.
 - ☐ Legend to denote "Occurrences."
 - ☐ Title: "Weather Data Occurrences by Date."
- **Error Handling:**
 - If an error occurs (e.g., missing table, invalid database file), the function prints the error message to the console.

5. Join_query.py:

Functions:

- **a. def combine_weather_air_data()**

Description: This function combines weather and air quality data stored in the database using an SQL JOIN query. The combined data is inserted into the WeatherAirQualityData table.

Inputs: (None) The function fetches data directly from the SQLite database.

Outputs: Inserts rows into the WeatherAirQualityData table.

Print Statements: The number of rows combined from WeatherData and AirQualityData and success or failure messages indicating the operation's result.

- **b. def export_to_csv():**

Description: This function exports data from the WeatherAirQualityData table into a CSV file for further analysis and reporting.

Inputs: (None) The function fetches data directly from the SQLite database.

Outputs: CSV File by creating a file named WeatherAirQualityData.csv in the working directory.

Print Statements:

- Status of data export (success or failure).
- Path to the generated CSV file.

6. Join_query.py:

Description: Joins the weather and air quality data (WeatherAirQualityData) based on the hour column and changes the conditions to conditions id number based on the weatherConditions table in our database.

a. def initialize_database

Input: None

Output: Performs database operations to create tables and insert initial data.

Steps:

1. Creates the WeatherConditions table with columns (id, condition)
2. Creates the WeatherAirQualityData table with columns(id key, hour, temp_c, wind_mph, humidity, aqi, main_pollutant, condition_id.
3. Checks if the WeatherConditions table is empty and populates it with default weather conditions: "Cloudy," "Overcast," "Partly Cloudy," and "Mist."

b. Verify_weatherdata_structure

Description: Prints the structure of the WeatherData table to debug and identify missing or incorrect columns.

Input: None

Output: None: Prints the schema of the WeatherData table.

Steps:

1. Executes PRAGMA table_info(WeatherData) to retrieve the structure of the table.
2. Prints column details (name, type, constraints, etc.).

c. Combine_weather_air_data

Purpose: Combines data from WeatherData and AirQualityData tables by matching date and hour. Maps condition_id using WeatherConditions and inserts up to 100 rows into the WeatherAirQualityData table.

Input: None

Output: Inserts rows into the WeatherAirQualityData table.

Steps:

1. Prints a sample of data from the WeatherData table for debugging.
2. Executes an SQL query to join WeatherData, Dates, and AirQualityData on matching date and hour fields.
3. Inserts the joined data into WeatherAirQualityData with appropriate columns, limiting to 100 rows.
4. Commits the changes to the database.

d. Verify_table_structure

Purpose: Prints the structure of the WeatherAirQualityData table to verify that it matches the intended schema.

Input: None

Output: Prints the schema of the WeatherAirQualityData table.

Steps:

1. Executes PRAGMA table_info(WeatherAirQualityData) to retrieve the structure of the table.
2. Prints column details (name, type, constraints, etc.).

e. Verify_data

Purpose: Prints a sample of data from the WeatherAirQualityData table to verify the inserted rows.

Input: None

Output: Prints a sample of rows from the WeatherAirQualityData table.

Steps:

1. Executes an SQL query to select up to 10 rows from the WeatherAirQualityData table.
2. Prints the retrieved rows for verification.

7. Main.py:

Description: Calculates the averages on air quality and of weather for each of the days and then creates a graph for it

a. def Calculate_averages

Description: Calculates the average Air Quality Index (AQI) and temperature (temp_c) for each weather condition, grouped by condition_id.

Input: None

Output: List of tuples that each tuple contains condition_id (int), condition (str), avg_aqi (float), and avg_temp_c (float).

Steps:

1. Executes an SQL query that: Joins WeatherAirQualityData with WeatherConditions using condition_id, groups by condition_id to calculate averages of aqi and temp_c, and orders results by condition_id.
2. Prints a table of the calculated averages.
3. Returns the results as a list of tuples.

b. def Write_to_text_file

Purpose: Writes the calculated averages to a text file for future reference.

Input: results (list of tuples): The output of calculate_averages.

Each tuple contains condition_id (int), condition (str), avg_aqi (float), avg_temp_c (float)

Output: Writes the results to a text file and prints a confirmation message.

Steps:

1. Opens the specified file in write mode.
2. Writes a header and formatted table of the averages to the file.
3. Prints a message confirming the file has been written.

c. **def Plot_averages**

Purpose: Creates a visualization of the average AQI and temperature for each weather condition using Matplotlib.

Input: data (list of tuples): The output of calculate_averages. Each tuple contains:

- condition_id (int): The ID of the weather condition.
- condition (str): The name of the weather condition.
- avg_aqi (float): The average AQI for the condition.
- avg_temp_c (float): The average temperature in degrees Celsius for the condition.

Output: Displays a dual-axis plot:

- A bar chart for average AQI.
- A line plot for average temperature.

Steps:

1. Extracts the condition names, average AQI, and average temperature from the input data.
2. Creates a figure with two y-axes:
3. Enhances the plot with labels, legends, and a title.
4. Displays the plot.

8. **WeatherAirQuality.db:**

Description: The WeatherAirQuality.db is an SQLite database designed to store historical weather data for specific locations over multiple days. The database dynamically creates a separate table for each date to organize weather data in a structured manner.

- Tables are created dynamically for each date in the format WeatherData_YYYY_MM_DD (e.g., WeatherData_2024_12_02).
- Each table stores hourly weather data for that specific date.

Inputs: API Key and location (Detroit, MI), start and end dates (start_date=2024-12-02, end_date=2024-12-06)

Outputs

1. Database Updates:
 - A new table is created for each day in the date range (if it does not already exist).
 - A limit of 25 rows of hourly weather data are inserted into each day's table.
2. Console Logs:
 - Success messages indicating that data for a specific date has been successfully stored or error messages if the API call fails for any date
 - Warnings when the 25-row limit is reached for a specific date.

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 (did it solve the issue?)

Date	Issue Description	Location of Resource	Result (did it solve the issue?)
12/6/2024	Spotify API key problems	Spotify website	No, we change to another API.
12/15/24	Database Join	Tutoring, and peer help, Chatgpt	Yes, we assigned the duplicate data to unique and we joined using one column (hour) which worked.
12/08/2024	History Data, we couldn't gather history from our api from 2020 which our background of question need	Google	We ended up using dates from last week
12/09/2024	Git pull result to no change when sharing the code	Cours note, VsCode terminal update result	Cours note provided an explanation concernant git. Terminal update gave the exact instruction about what to add before

			pulling the code.
12/13/2024	Dysfunctioning of Air Quality API key of partner and sweet to another key	https://dashboard.iqair.com/personal/api-keys ChatGPT	Yes, ChatGPT discovered the problems and provided a solution to verify the key parameter and fix the code.
12/14/2024	WeathearAirQuality.db dataset is empty. No matching data found between WeatherData and AirQualityData.	Cours note, debugging,	Yes, the weather.py works first and permits us to fix other files.

Conclusion

This project shows how weather affects air quality in Detroit. Humid and still conditions lead to worse air quality, while wind and rain help clear pollutants. By combining weather and air quality data, we found key patterns and created visuals to explain them. This project highlights how data can help cities like Detroit improve air quality and protect public health.