Zeke Butler
Andrew Kagan
Isaac Abrams

Final Project Report - StormWatch Analytics

Github Link: https://github.com/zfbutler/FINAL-PROJECT-SI-201

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

Overall Research Question: How does rainfall relate to daytime car crash activity in New York City versus Chicago?

To answer this research question, the original plan was to use four APIs: New York City open-data API providing detailed crash records, a Chicago open-data API providing detailed crash records, a Weather open-data API for daily weather (especially precipitation) in NYC and Chicago, and a possible fourth API (initially air quality) to explore whether a second environmental factor might also relate to crash counts.

Then we wanted to build a SQL database that would store daily crash statistics (number of crashes, injuries, fatalities) for NYC and Chicago and also store daily precipitation totals for both cities. We also wanted it to limit every API-population function to adding at most 25 new dates per run, as to respect the requirements of the assignment.

After, we wanted to clean and aggregate raw API outputs into daily statistics, join weather and crash data by date, run simple summary calculations like averages or correlations, and create visualizations that compare crash counts on rainy vs. dry days and show the relationship between precipitation and crashes.

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

APIs we actually used:

1. New York City crash data
- Source: NYC Open Data - Motor Vehicle Collisions - Crashes
- Endpoint: https://data.cityofnewyork.us/resource/h9gi-nx95.json
- Used in NYC_API.py

2. Chicago crash data
- Source: City of Chicago Open Data - crash dataset
- Endpoint: https://data.cityofchicago.org/resource/85ca-t3if.json
- Used in chi_api.py

3. OpenWeather History API

- Source: OpenWeather “History” hourly endpoint
- Endpoint pattern: https://history.openweathermap.org/data/2.5/history/city?lat={lat}&lon={lon}&type=hour&start={start}&end={end}&appid={API key}
- Used in WEATHER_API.py.

4. We decided not to keep a fourth API (like air-quality) because the weather + crash pairing already produced a full project, additionally the air-quality API didn’t seem like it would work well with the data we were collecting on second inspection

Data we actually gathered -
Over multiple runs (6 new dates per run since we would get 4 entries per date so that each run was produced less than 25 new entries) we populated weather_crashes.db for each date in roughly the last 100 days with

NYC: total_crashes, total_injuries, total_fatalities
Chicago: total_crashes, total_injuries, total_fatalities
Weather: precip_mm (total rain + snow in millimetres) for NYC, precip_mm (total rain + snow in millimetres) for Chicago

Key files and roles -
WEATHER_API.py: Pulls hourly historical weather from OpenWeather, aggregates to daily precipitation, and inserts rows into NYCWeather and ChicagoWeather, helps populate weather_crashes.db (Andrew)

NYC_API.py: Calls the NYC crash API, filters to daytime (6:00–18:00), aggregates to daily counts, and helps populate weather_crashes.db (Zeke)

chi_api.py: Calls the Chicago crash API, performs similar cleaning, and helps populate weather_crashes.db (Isaac)

populate.py: Orchestrates the population steps, calling both the weather script and the city-specific crash scripts (Zeke and Andrew)

analysis.py: Joins crash and weather tables, runs calculations, and creates the visualizations saved as PNG files (Isaac)

results.txt: Text output file created by analysis.py, stores the numerical results of our analysis in a readable format (Isaac and Zeke)

3. The problems that you faced -

OpenWeather one call 3.0 subscription error (401): Initially we tried to use the One Call 3.0 day_summary endpoint, but our free account did not have the required “One Call by Call” plan so every request returned a 401 error. To fix this problem we switched to the OpenWeather History API (/data/2.5/history/city) and rewrote WEATHER_API.py to query 24 hours of hourly data per date and sum rain["1h"] and snow["1h"] for each hour to get daily precipitation.

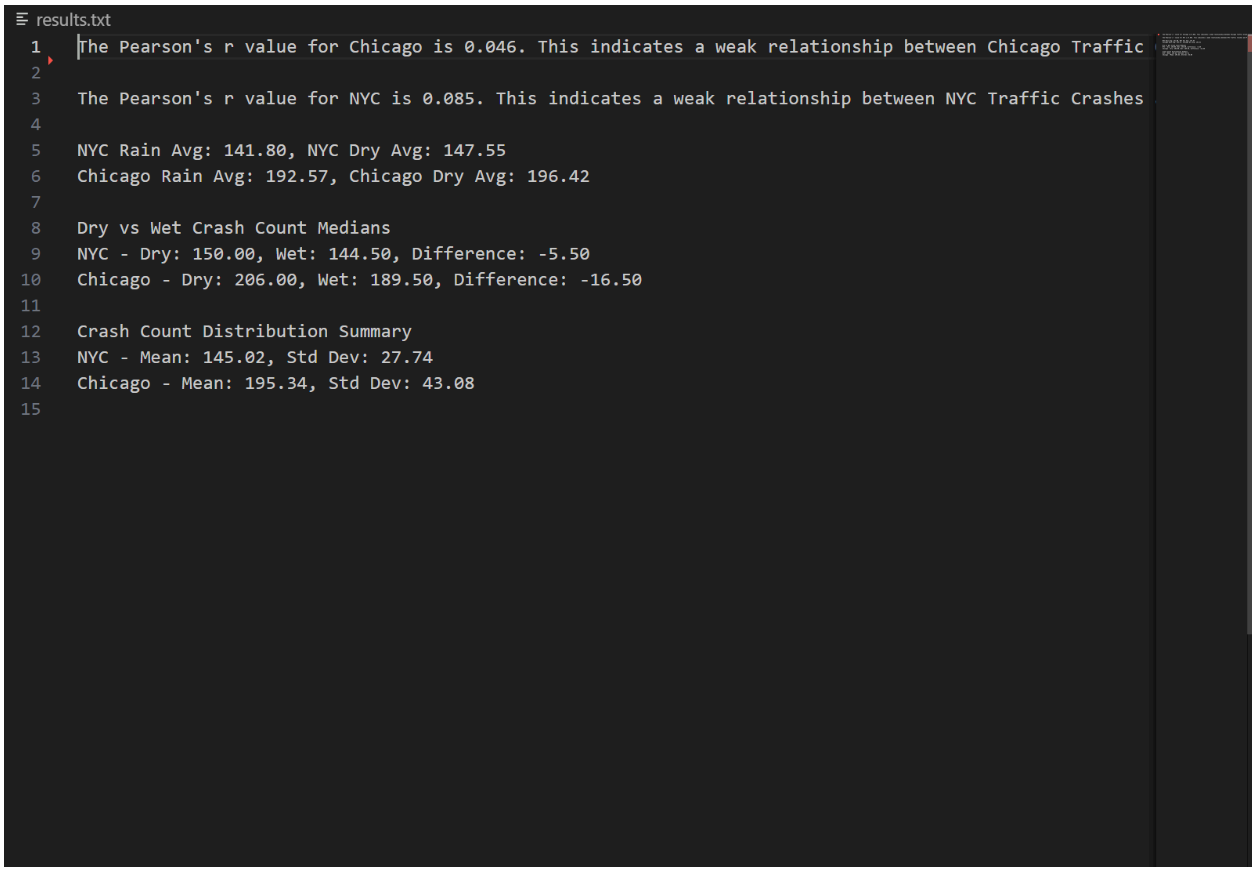
Database design & Shared keys: We experimented with separate tables with different ID schemes and a shared DayIndex table with a common ID as the format for organizing the data. Ultimately, for simplicity, we settled that our final Weather API code uses autoincrement IDs per table but keeps date unique in each weather table. The crash tables also store a date column, and we join everything on date in analysis.py. This however required dropping old tables, clearing data, and rebuilding the DB a few times which was definitely challenging.

25 rows per run requirement: Each API script had to respect a “25 new days per run” limit. We implemented this by tracking existing dates in each table and incrementing a counter and breaking the loop once max_new_dates / max_days reached 25. However, once we had our grading session we realized that it was 25 max data entries per call not rows, so we changed our program so that for each call it only added 6 new dates as each date created 4 new rows in the database so that it satisfied the requirement.

Cleaning messy crash data: Our crash datasets seemed to contain missing fields, strange timestamps, and night-time crashes that we wanted to exclude. So to address this problem, we built helper functions (check_crash_data in NYC, collect_crash_data in Chicago) that skip rows with missing dates/times, convert string timestamps to datetime objects, filter to daytime crashes only, and sum injuries and fatalities, ignoring bad values using try/except.

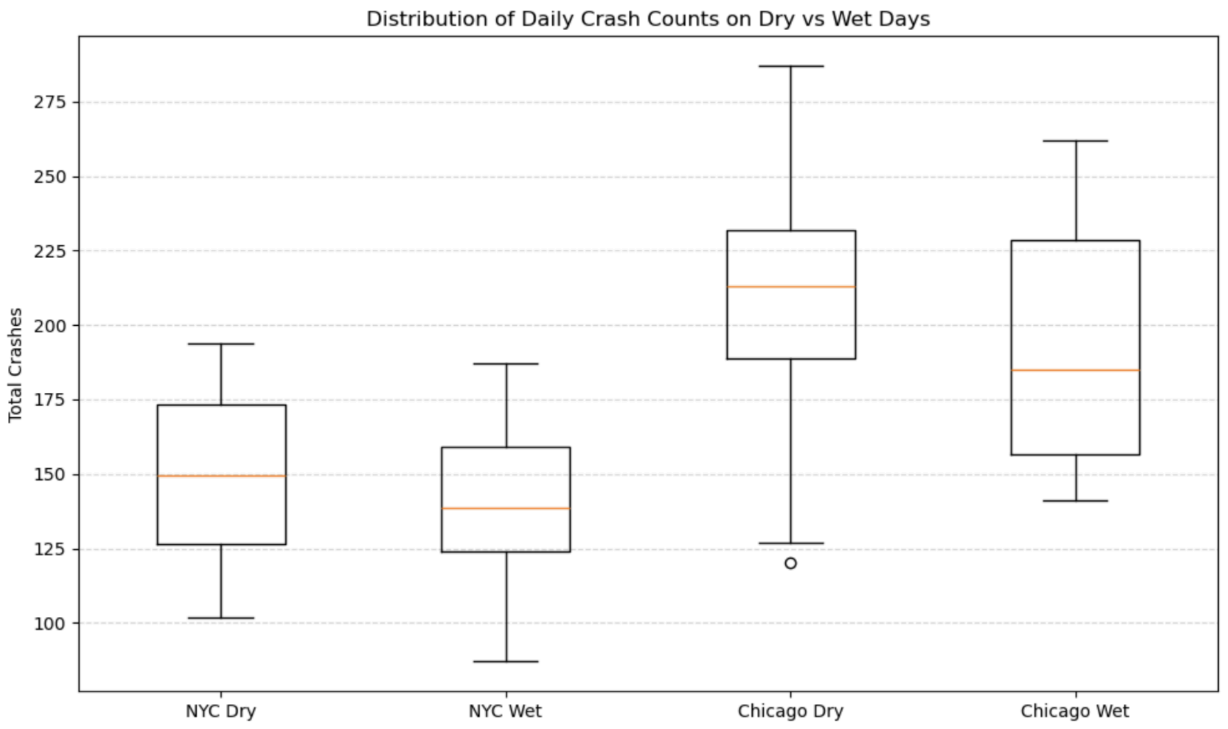
Git merge conflict on the database file: When pulling from GitHub we often got a binary merge conflict on weather_crashes.db. To fix this we learned to resolve by removing the local DB, choosing one version (or regenerating it), and then committing a fresh DB or excluding it from version control to solve the issue.

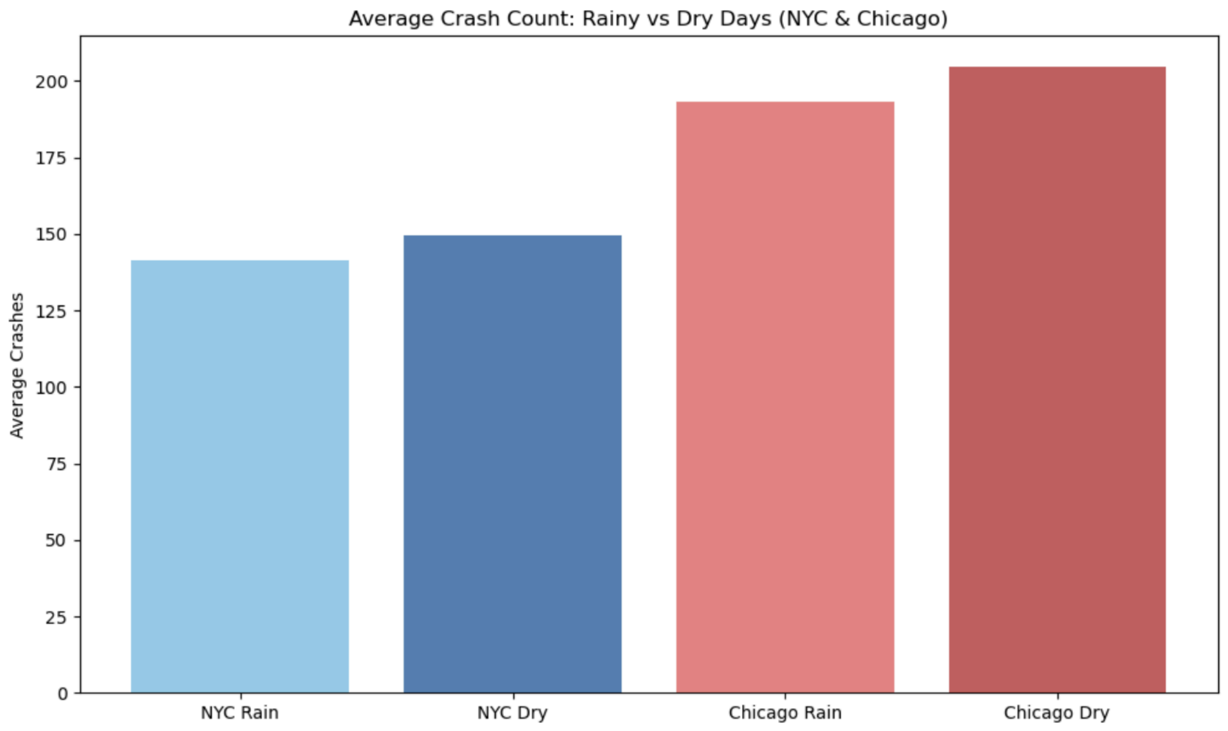
4. The calculations from the data in the database (i.e. a screen shot) -

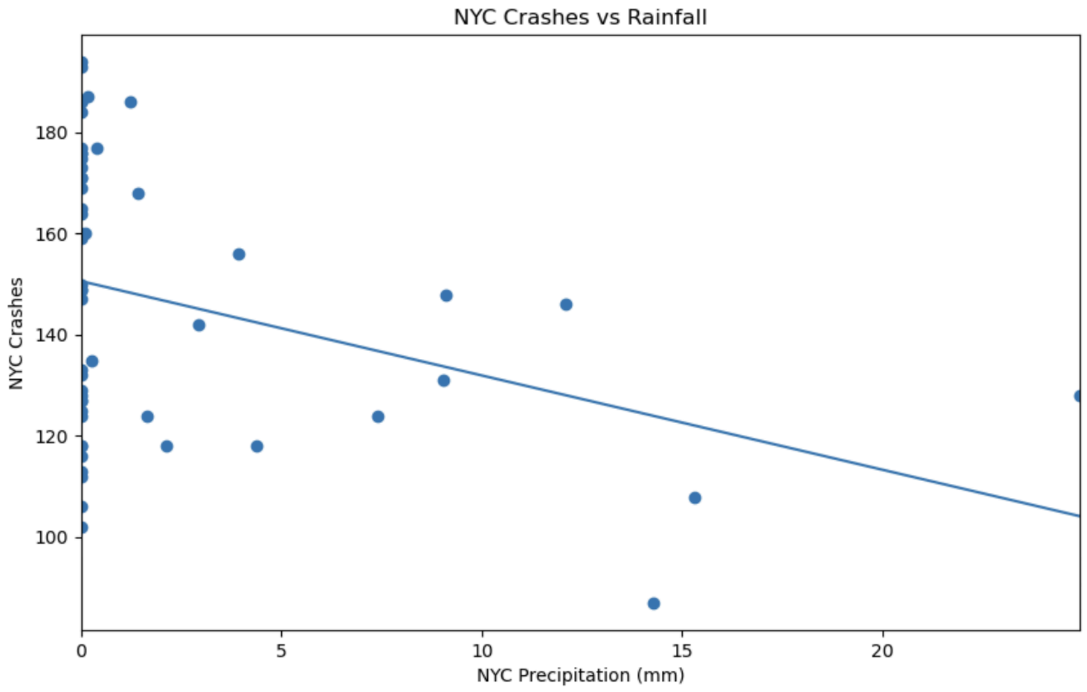


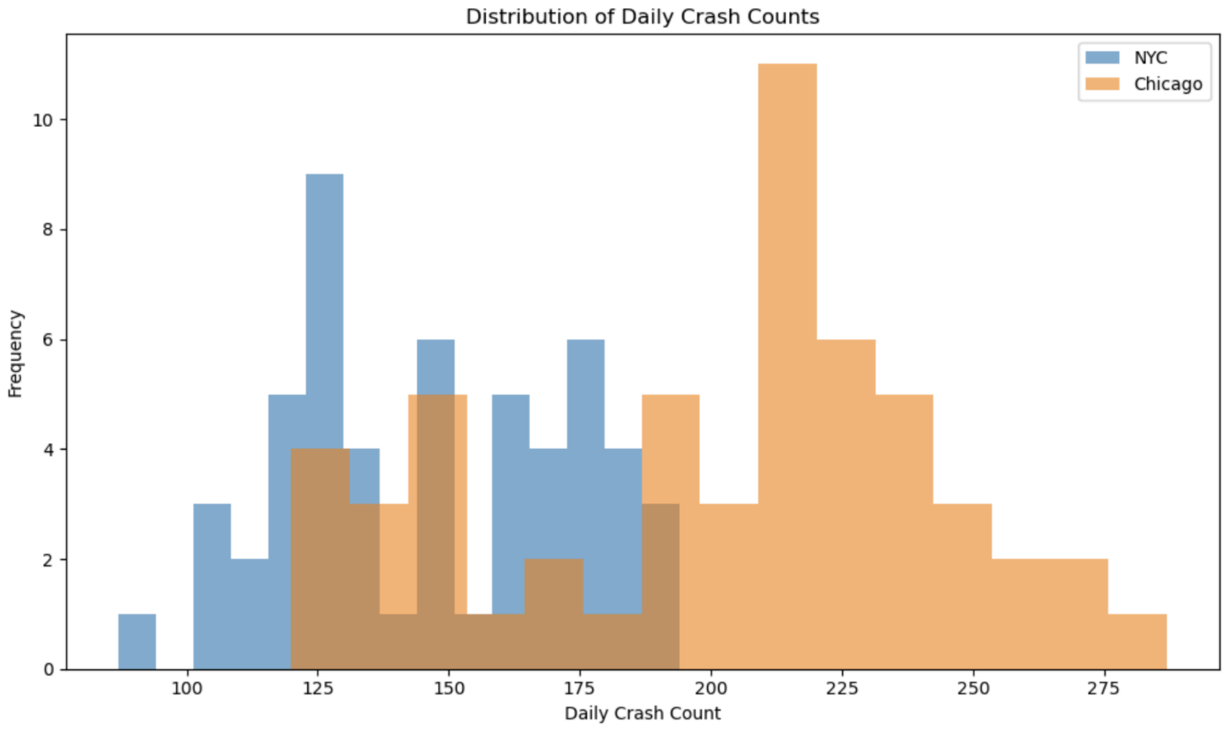
```
The Pearson's r value for Chicago is 0.046. This indicates a weak relationship between Chicago Traffic

The Pearson's r value for NYC is 0.085. This indicates a weak relationship between NYC Traffic Crashes

NYC Rain Avg: 141.80, NYC Dry Avg: 147.55
Chicago Rain Avg: 192.57, Chicago Dry Avg: 196.42

Dry vs Wet Crash Count Medians
NYC - Dry: 150.00, Wet: 144.50, Difference: -5.50
Chicago - Dry: 206.00, Wet: 189.50, Difference: -16.50

Crash Count Distribution Summary
NYC - Mean: 145.02, Std Dev: 27.74
Chicago - Mean: 195.34, Std Dev: 43.08
```

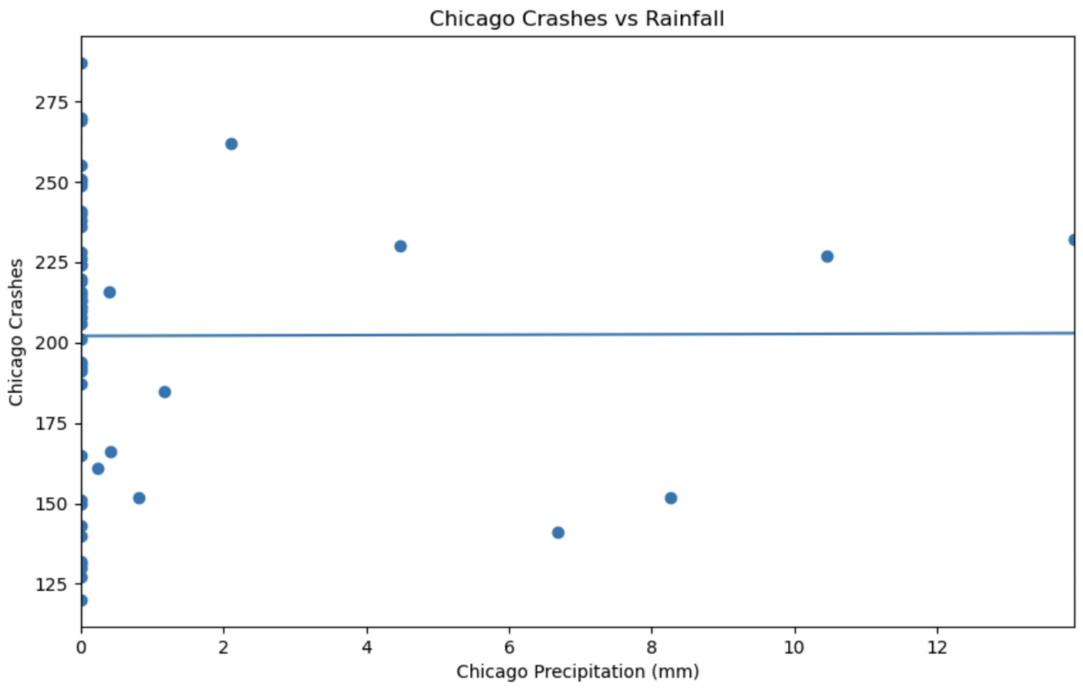
5. The visualization that you created (i.e. screen shot or image file) -


Distribution of Daily Crash Counts on Dry vs Wet Days
Total Crashes
275
250
225
200
175
150
125
100
NYC Dry
NYC Wet
Chicago Dry
Chicago Wet


Average Crash Count: Rainy vs Dry Days (NYC & Chicago)
Average Crashes
200
175
150
125
100
75
50
25
0
NYC Rain
NYC Dry
Chicago Rain
Chicago Dry


NYC Crashes vs Rainfall
NYC Crashes
NYC Precipitation (mm)
100
120
140
160
180
0
5
10
15
20


Distribution of Daily Crash Counts
NYC
Chicago
Frequency
Daily Crash Count
0
2
4
6
8
10
100
125
150
175
200
225
250
275



6. Instructions for running your code -

First make sure to first see if there is data in the database (or there is no database or the database is empty). If there is data in the database, go to line 60 in populate.py: main(run_clear=False) and change the False to a True to clear the tables before populating the database. If there is no database yet or the database tables are empty there is no need to change anything. Then run the populate.py file, it will generate 6 dates into the weather_crashes.db database of crash and weather data in nyc and chicago. To generate more dates or rows (say 100+ for every table) just keep rerunning populate.py as it will keep adding 6 dates at a time per call (make sure that when you rerun populate.py we have that line 60: main(run_clear=False) so that we don't clear tables on every iteration). Keep rerunning populate.py until you have the desired amount of data. Once this is the case, run analysis.py which connects to weather_crashes.db, joins weather and crash tables by date, computes summary statistics and writes results to results.txt, and also produces the five visualizations listed above and saves them as PNG files in the project directory.

Keep in mind, analysis.py will not work unless there is data populated in the tables already, a sufficient amount of it. So make sure to populate the database a bunch of times before you try to run the code in the file. Additionally, for better results, delete result.txt between uses. The code appends the text to the file, it does not write it all at once. Deleting this file will just make the analysis written to the file looked better.

7. An updated function diagram with the names of each function, the input, and output and who was responsible for that function -

WEATHER_API.py (Andrew) -

create_weather_tables()
Input - None
Output - None
Description - Connects to weather_crashes.db and creates (if missing) the weather-related tables (ex: NYCWeather, ChicagoWeather). Commits and closes the DB connection

days_between(start, end)
Input - start (datetime.date), end (datetime.date)
Output - list[datetime.date]
Description - Builds and returns a list of every date from start to end inclusive by incrementing one day at a time

day_unix_range(d)
Input - d (datetime.date)
Output - tuple[int, int] = (start_ts, end_ts) Unix timestamps (UTC)
Description - Converts a calendar day into a UTC timestamp window, which is midnight at the start of the day through midnight of the next day

fetch_history_for_day(lat, lon, d)
Input - lat (float), lon (float), d (datetime.date)
Output - dict (JSON parsed into a Python dictionary)
Description - Calls OpenWeather History API for hourly data over the UTC day window for the given lat/lon. If the response isn't 200, prints debug info and raises an HTTP error, otherwise returns the JSON

precip_from_history_json(history_json)
Input - history_json (dict)
Output - float (total precipitation in mm for the day)
Description - Iterates through history_json["list"] (hourly entries) and sums rain["1h"] + snow["1h"] for each hour (defaulting missing values to 0). Returns the daily total.

populate_weather_for_dates(date_list, max_days)
Input - date_list (list[datetime.date]), max_days (int) ("store at most 25 new days per run" cap)
Output - list[str] (date strings "YYYY-MM-DD" actually inserted this run)

Description - Loads existing dates from the DB, then loops through dates (sorted). For each new date, it fetches NYC + Chicago hourly history, converts each to daily precip totals, and writes to the DB. Stops once processed >= max_days.

drop_legacy_weather_tables()
Input - none
Output - none
Description - Drops old tables if they exist (ex: WeatherDaily, DailyWeather) from the database to keep everything clean.

NYC_API.py (Zeke) -

clear_tables()
Input - none
Output - none
Description - Connects to weather_crashes.db and drops NYC-related tables. Used to “reset” before repopulating.

create_nyc_table()
Input - none
Output - none
Description - Creates the NYC crashes table (if not already created). Uses date as a UNIQUE field so duplicate days don’t get duplicated.

fetch_nyc_crashes(limit=1000, offset=0, where=None, select=None, order=None)
Input - Socrata query controls, limit, offset, optional where, select, order
Output - list[dict] (JSON rows from the API)
Description - Calls the NYC Open Data endpoint with the provided $where, $select, $order, etc. Returns parsed JSON rows (or empty list if error).

check_crash_data(api_data)
Input - api_data (list[dict])
Output - tuple[int, int, int] = (total_crashes, total_injuries, total_fatalities)
Description - Filters crash rows to daytime hours and accumulates totals (crash count, injured, killed). Returns the three totals.

insert_weather_stats(date_str, total_crashes, total_injuries, total_fatalities)
Input - date_str (str, "YYYY-MM-DD"), total_crashes (int), total_injuries (int), total_fatalities (int)
Output - None
Description - Inserts (or replaces) one daily summary row into the NYC crash stats table for that date

date_already_processed(date_str)

Input - date_str (str)
Output - bool
Description - Queries the NYC table for that date. Returns True if already present, else False. Used to prevent duplicates across runs.

populate_nyc_crashes(date_list, max_new_dates=6)
Input - date_list (list[str] of dates), max_new_dates (int)
Output - int (number of new dates added this run)
Description - Loops through dates and inserts crash summaries for at most max_new_dates new days (cap logic uses new_dates_added >= max_new_dates: break). Skips dates already in the DB.

main()
Input - none
Output - none
Description - Driver function it creates the NYC table and calls populate_nyc_crashes(...), then prints how many dates were processed.

chi_api.py (Isaac)

clear_chi_tables()
Input - none
Output - none
Description - Drops Chicago-related tables in weather_crashes.db so the Chicago pipeline can rebuild from scratch.

create_tables()
Input - none
Output - none
Description - Creates the main Chicago crash summary table (if it doesn't exist).

fetch_chi_crashes(limit=1000, offset=0, where=None, select=None, order=None)
Input - Socrata query controls, which are limit, offset, optional where, select, order
Output - list[dict]
Description - Calls Chicago's Open Data endpoint with filters and returns parsed JSON rows (or [] on error).

collect_crash_data(api_data)
Input - api_data (list[dict])
Output - tuple[int, int, int] = (total_crashes, total_injuries, total_fatalities)
Description - Parses each record, filters to daytime hours, counts crashes, and sums injuries + fatalities. Returns totals.

insert_chi_crashdata(date_str, total_crashes, total_injuries, total_fatalities)

Input - date_str (str), total_crashes (int), total_injuries (int), total_fatalities (int)
Output - none
Description - Inserts one daily Chicago summary row (date + totals) into chi_crash_data (uses INSERT OR IGNORE in your file).

date_already_processed(date_str)
Input - date_str (str)
Output - bool
Description - Checks whether the given date already exists in the Chicago table. Used to avoid duplicates.

nycweather_id_already_processed(date_str)
Input - date_str (str)
Output - int or none (returns an id if found, otherwise None)
Description - Looks up the NYCWeather table by date and returns the matching id. This is used as a shared key bridge so Chicago rows can reuse the NYCWeather row id for the same date.

populate_chi_tables(target_dates)
Input - target_dates (list[str])
Output - none
Description - Loops through target dates, limits inserts to 25 new dates per run via max_new_dates = 25 and if new_dates_added >= max_new_dates: break. For each date, pulls Chicago crash rows, aggregates totals, then inserts into the database.

main()
Input - none
Output - none
Description - Calls populate_chi_tables(target_dates) (and optionally clear_chi_tables() depending on what's commented/uncommented).

populate.py (Zeke and Andrew)
main()
Input - none
Output - none
Description - Kind of the project pipeline runner. This file is the orchestrator that imports/executes the different modules so the DB gets populated in the right order (weather first to get dates/keys, then NYC + Chicago crash tables, and so on).

analysis.py (Zeke and Isaac)

load_data_for_analysis(db_path="weather_crashes.db")
Input - db_path: path to the SQLite database (default "weather_crashes.db")
Output - data (dict) with structure: "dates" → list[str] (ordered date strings), "nyc" → dict with "precip", "crashes", "injuries" lists, "chi" → dict with "precip", "crashes", "injuries" lists

Description - Connects to the SQLite database and runs a SQL JOIN across NYCWeather, ChicagoWeather, nyc_crash_stats, and chi_crash_data using the shared id fields. It fetches the joined rows ordered by date, then builds and returns a single dictionary holding aligned per-day arrays for precipitation/crashes/injuries for NYC and Chicago.

nyc_crash_weather_corr(data)
Input - data (dict), the dictionary returned by load_data_for_analysis()
Output - none
Description - Pulls NYC precipitation and crash counts from data, converts them to NumPy arrays, computes Pearson correlation (r) using np.corrcoef, and writes a formatted sentence about the correlation strength to results.txt (append mode). Then it creates a scatter plot of NYC precipitation vs NYC crashes, optionally overlays a regression line using np.polyfit, saves the figure as nyc_crash_weather_scatter.png, and displays it.

chi_crash_weather_corr(data)
Input - data (dict), the dictionary returned by load_data_for_analysis()
Output - none
Description - Same analysis pattern as the NYC function, but for Chicago, computes Pearson's r between Chicago precipitation and Chicago crash counts, appends the message to results.txt, creates and saves a scatter plot with optional regression line, saves as cho_crash_weather_scatter.png, and displays it.

rainy_vs_dry_barchart(data)
Input - data (dict), the dictionary returned by load_data_for_analysis()
Output - none
Description - Splits crash counts into rainy vs dry days separately for NYC and Chicago using the rule: rainy if precip > 0, else dry. Computes average crashes for each group (NYC rainy, NYC dry, Chicago rainy, Chicago dry), appends the averages to results.txt, then creates a bar chart comparing those four averages, saves it as rainy_vs_dry_bar.png, and displays it.

rainy_vs_dry_boxplot(data)
Input - data (dict), the dictionary returned by load_data_for_analysis()
Output - None
Description - Builds four lists of crash counts: NYC dry, NYC wet, Chicago dry, Chicago wet (again using precip > 0 as wet). Computes medians for each group using np.median, writes median values and differences to results.txt, then creates a boxplot comparing the distributions, saves it as rainy_vs_dry_boxplot.png, and displays it.

crash_histogram(data)
Input - data (dict): the dictionary returned by load_data_for_analysis()
Output - none
Description - Uses NYC and Chicago daily crash count lists to compute mean and standard deviation for each city using np.mean and np.std. Appends those summary stats to results.txt.

Then creates an overlaid histogram (NYC and Chicago) to compare distributions of daily crash counts, saves it as crash_histogram.png, and displays it.

main()
Input - none; Output - none
Description - Calls load_data_for_analysis() to get a merged dataset, then runs all plot functions
8.
You must also clearly document all resources you used. The documentation should be of the following form

DATE	ISSUE	LOCATION OF RESOURCE	RESULT
Dec 7th	The given API key is not working for NYC weather.api	ChatGPT	Chat pointed out that for this request, I should use the app token, not the key.
Dec 9th	Issue with figuring out which weather tower served to report weather data for NYC and Chicago.	ChatGPT	Chat found good locations to use as markers for the weather api.
Dec 10th	Issue with pulling out dates as standardized strings for the sake of creating cohesion across the folder	ChatGPT, OpenWeather, NYC city	The result we found was that we had to strip the times and then turn them into YYYY MM DD, from unix time
Dec 11th	Issue with changing the format of the tables because of feedback from the grading session.	ChatGPT, Homework 8, Lecture	We had sorted by using date as the shared key, but because it was split across multiple tables, we had to change it. It took a lot of reworking in the code; rather than just changing what got inputted, we had to change the way the data was collected.
Dec 13	From our initial grading session, we found out that we	Google, and other homework.	We were able to find more visualizations to add to the folder to

	were going to lose points from splitting tables, but it is inherent in our structure so we had to add more visualizations to get the points back.		get back on track. Isaac and Andrew have other classes where they worked with similar data, so we checked.