AEROTRACK-Real Time Air Quality Monitoring

PYTHON ESSENTIALS

[VITyarthi]

SLOT:B11+B12+B13

By

Ms.JIGYASA

Reg No. :25BCY10166

Submitted to:

Ms. Murugeswari K.

TABLE OF CONTENTS

INTRODUCTION

AeroTrack is a real-time Python-based air quality monitoring system for providing immediate AQI and pollutant-level information of any location. The system connects to the OpenWeatherMap API for fetching environmental data, then shows it in a structured, easy-to-understand format in the console.

It aims to make users aware of the environmental conditions within their surroundings so that better decisions regarding health and lifestyle can be made.

PROBLEM STATEMENT

Air pollution is increasing rapidly, and many people do not have easy access to current air quality data. Various existing solutions may be complex, paid, or installed.

There is a need for a simple, accessible, and lightweight tool that provides:

Real-time AQI

Concentration levels of the pollutants

Health-based interpretation

Easy location-based monitoring

AeroTrack addresses this by offering a CLI-based program that generates up-to-date air quality reports with minimal effort.

FUNCTIONAL REQUIREMENTS

Functional requirements describe what the system must do.

1 Input Requirements

It should take a place name as input from the user.

2 API Interaction

The system must call the Geocoding API to convert place names into coordinates.

The system must call the Air Pollution API to fetch AQI and pollutant data.

3 Output Requirements

Must demonstrate:

AQI (1–5)

Detailed pollutant concentrations (CO, NO_2, O_3, etc.)

Latitude and longitude

AQI category interpretation (Good / Fair / Poor etc.)

4 Error Handling

Must detect invalid place names.

Must handle unavailable data or API issues.

Must display appropriate error messages.

NON FUNCTIONAL REQUIREMENTS

1 Performance

API responses must be processed within 2–4 seconds.

Lightweight execution: low consumption of memory.

2 Usability

Console output should be clean, readable, and well formatted.

Users mustn't require technical knowledge.

3 Reliability

Should respond correctly even in case of missing pollutant data.

Should check API response formats.

4 Scalability

Can be extended to multiple cities, historical results, or dashboards.

5 Security

API keys should be kept safe, preferably via environment variables.

6 Portability

This should run on any OS with Python installed.

SYSTEM ARCHITECTURE

This solution follows a three-layer architecture:

1. Presentation Layer

Accepts user input: place name

Displays formatted air quality report

2. Business Logic Layer

get_coordinates(place_name)

fetch_air_quality(place_name)

interpret_aqi(aqi)

3. Data Access Layer

Calls external APIs using requests

Parses JSON response

Data Flow:

User → Program → Geocoding API → Coordinates → AQI API → Report Output

DESIGN DIAGRAMS

1 Use Case Diagram

Actors:User

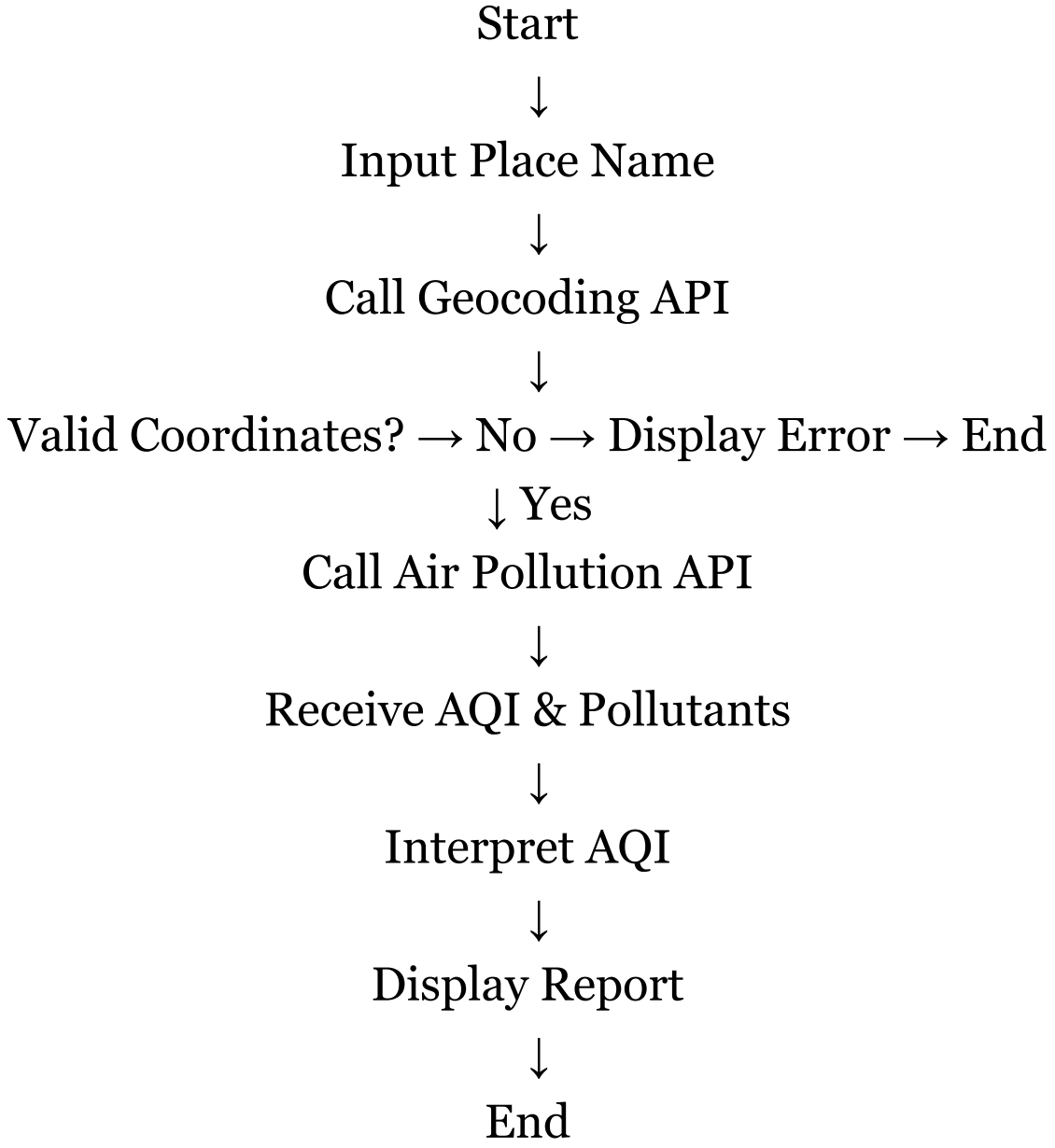
Use Cases:

- Enter place name
- View AQI report
- See pollutant details
- Receive interpretation
- Handle errors

User → [Enter Location] → System → [Fetch & Display AQI]

User → [Populate Pollutant Information]

2 Workflow Diagram



3 Sequence Diagram

User → Program: Input the place name

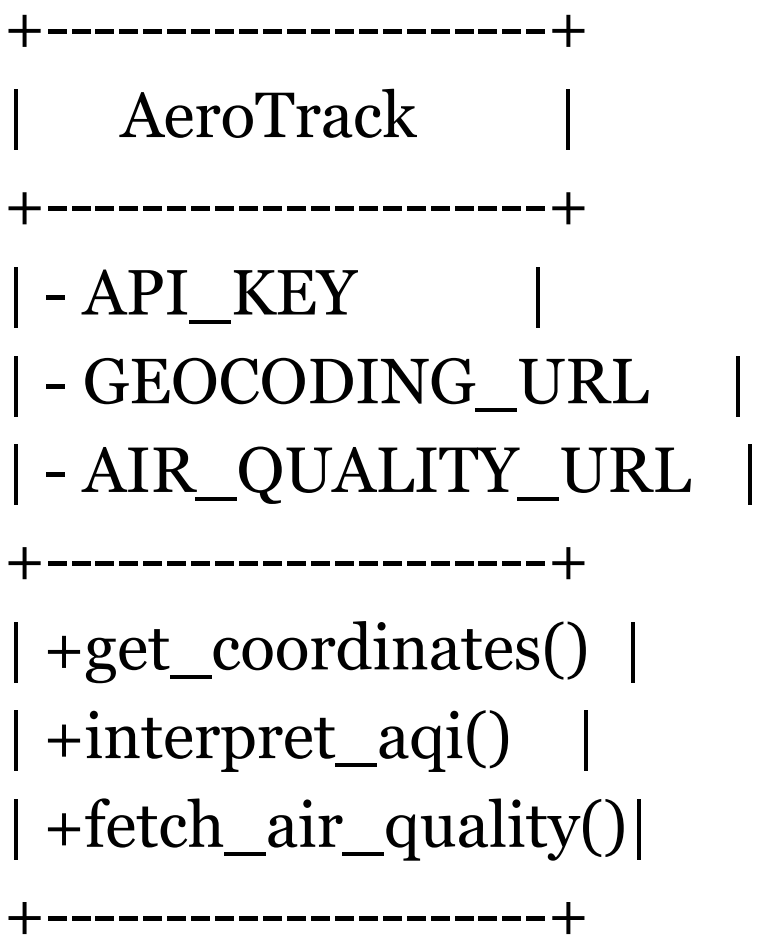
Program → Geocoding API: Request coordinates

Geocoding API → Program: Return lat/lon

Program → AQI API: Request air quality data

AQI API → Program: Return AQI & components Program → User: show formatted report

4 Class / Component Diagram



5 ER Diagram

(Not applicable — no local storage/database used)

Design Decisions & Rationale

DECISIONS	RATIONALE
CLI-based program	Simple and lightweight; no GUI dependency
OpenWeatherMap API	Free, reliable, global coverage
requests library	Clean and easy HTTP handling
AQI Interpretation dictionary	Faster lookup and cleaner code
Separation of functions	Improves maintainability and readability.

Implementation Details

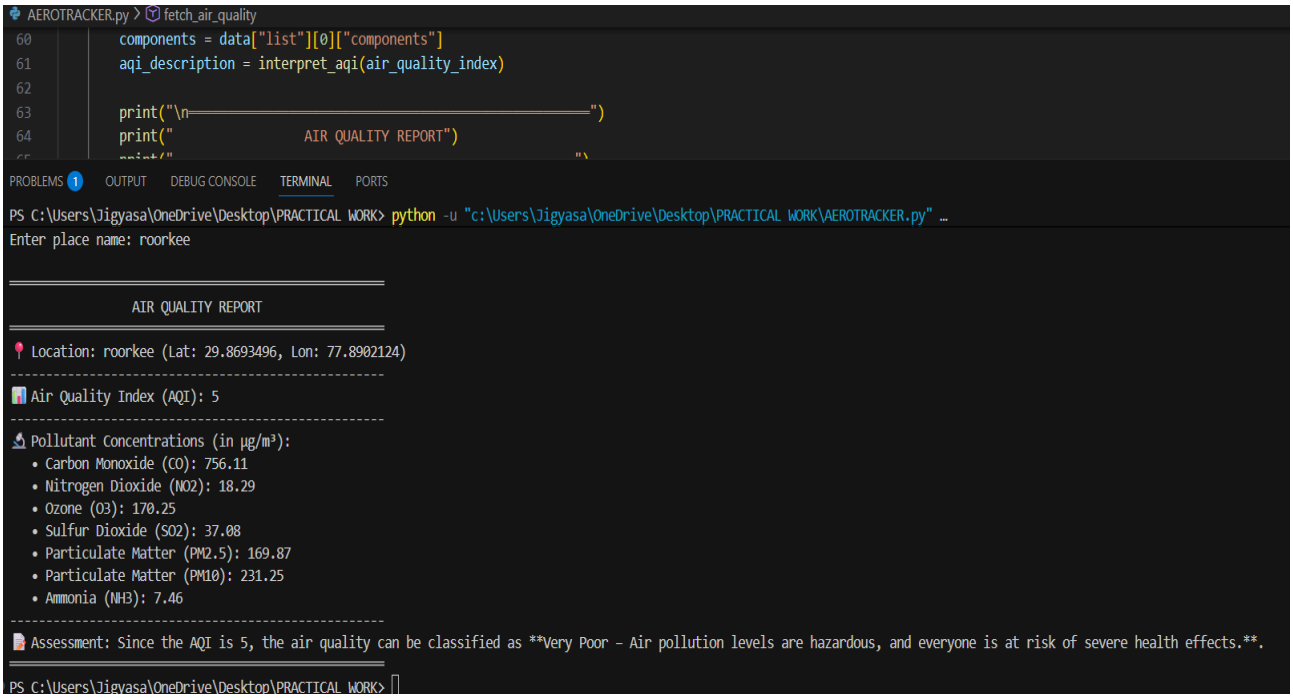
1.Implemented in Python 3.11

2.Three major functions:
get_coordinates() → Fetch latitude & longitude
fetch_air_quality() → Retrieve AQI + pollutants

interpret_aqi() → Convert AQI to readable text

3. Uses JSON parsing, conditional checks, and formatted printing.
4. Error handling ensures smooth user experience.

SCREENSHOT/OUTPUTS



TESTING APPROACH

Test Types:

1. Functional Testing

- Enter valid/invalid place names
- Properly parse API data

2. Negative Testing

- Network failure
- Invalid API key
- Empty responses

3. Boundary Testing

- Cities with multiple words (“New York”)
- Small villages
- Locations with missing AQI data

4. Usability Testing

- Readability and clarity of console output

Challenges Faced

- Handling wrong or misspelled location names
- Dealing with inconsistent API responses
- Managing API key security
- Ensuring user-friendly formatting
- Internet dependency in testing

Learnings & Key Takeaways

- Understanding REST APIs & HTTP GET requests
- Working with real-time JSON data
- Error handling and input validation
- Readable UI/UX is important, even for CLI programs.
- Fundamentals of categories of environmental pollutants
- Structured program design using functions

FUTURE ENHANCEMENTS

- Add historical AQI trend graphs
- Create a web or mobile UI
- Add database to store past results
- Provide notifications in case of poor air quality
- Visualization of maps using libraries like Folium
- Multicity comparison analytics
- Support for Offline Caching

REFERENCES

In order to work on this project websites are referred by me during the various phases of development of the project.

1) www.youtube.com

2) www.python.com

OTHER THAN THE MENTIONED THING I HAVE ALSO SEEKED HELP AND INFORMATION FROM MY TEACHERS WHO MADE ME UNDERSTAND EACH AND EVERY DETAIL OF THE PROJECT.