COIT20245: INTRODUCTION TO PROGRAMMING (HT1, 2024)

**ASSIGNMENT 2 – PROJECT**

**Queensland Wildlife Sightings**

**Present to: Mohamed Anver** (Tutor)

**Due date:** 19 May 2024

**LEADER**

**Name:** Tanvi Kaushik Kumar Patel

**Student ID:** 12266182

**MEMBER 1**

**Name:** Abhishek Prajapati

**Student ID:** 12264697

**MEMBER 2**

**Name:** Mututantirige Manmee Jeevanga Fernando

**Student ID:** 12248655

Table of content

**Overview and Purpose......................................................3**

**What are the tools and technologies used to create this project?...............................................................................3**

**GitHub Link.......................................................................3**

**Testcases and Output (Task 1 to Task 10) .......................4-28**

**Conclusion...........................................................................28**

1. **OVERVIEW AND PURPOSE**

The Queensland Wildlife Sightings Python web application is designed to facilitate the observation, recording, and analysis of wildlife sightings across Queensland, Australia. Leveraging modern web technologies, the application provides an intuitive platform for users to report sightings, view real-time data on wildlife, and engage with the local ecological community.

1. **What are the tools and technologies used to create this project?**

**Backend**

**Python:** The primary programming language for backend development.

**Development and Collaboration Tools**

**Git:** Version control system for tracking changes in the source code.

**GitHub:** Platforms for hosting the code repository and facilitating collaboration.

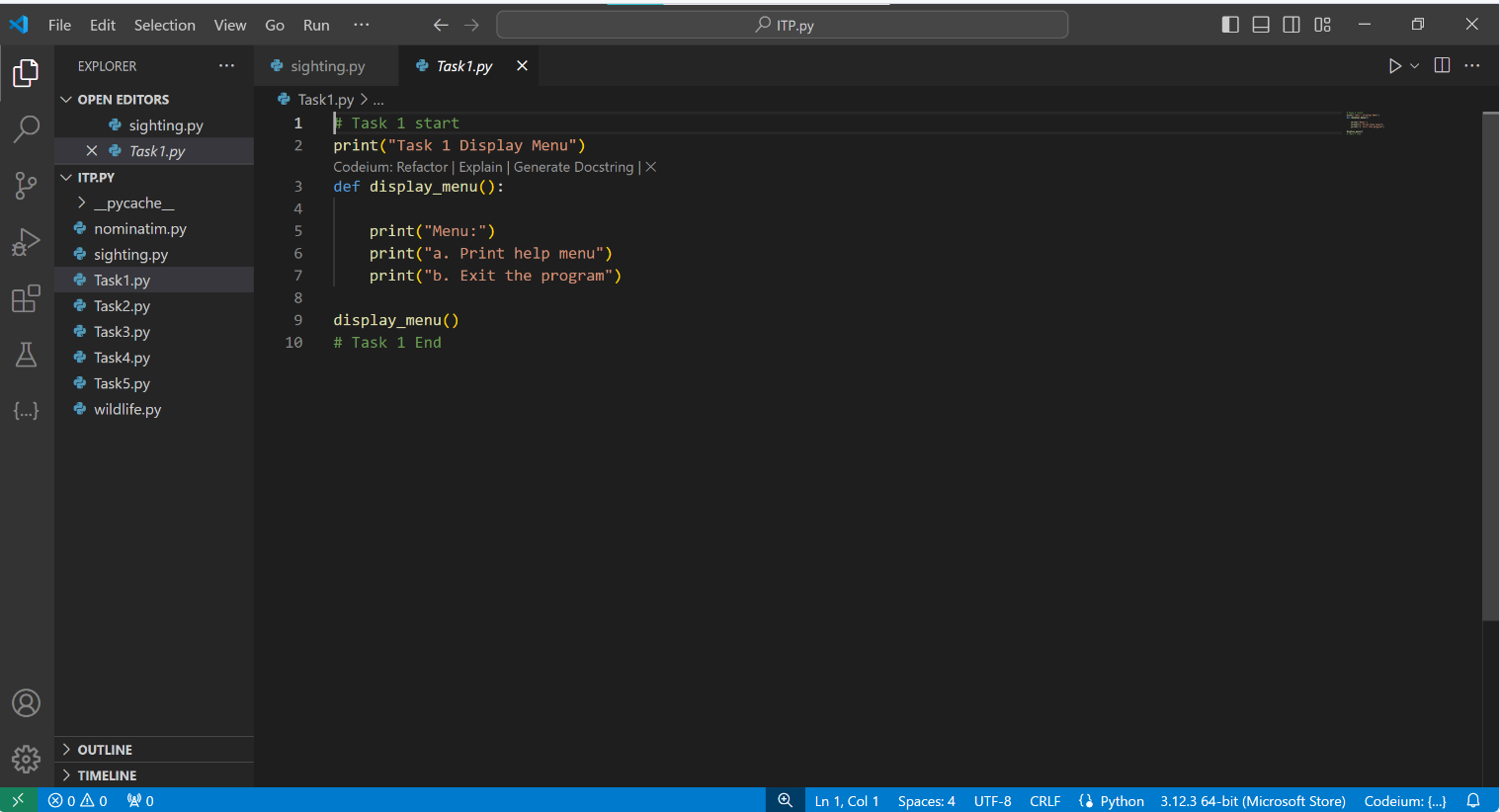
**Mapping and Geolocation**

**Provided API:** Services that provide map data and geolocation functionalities.

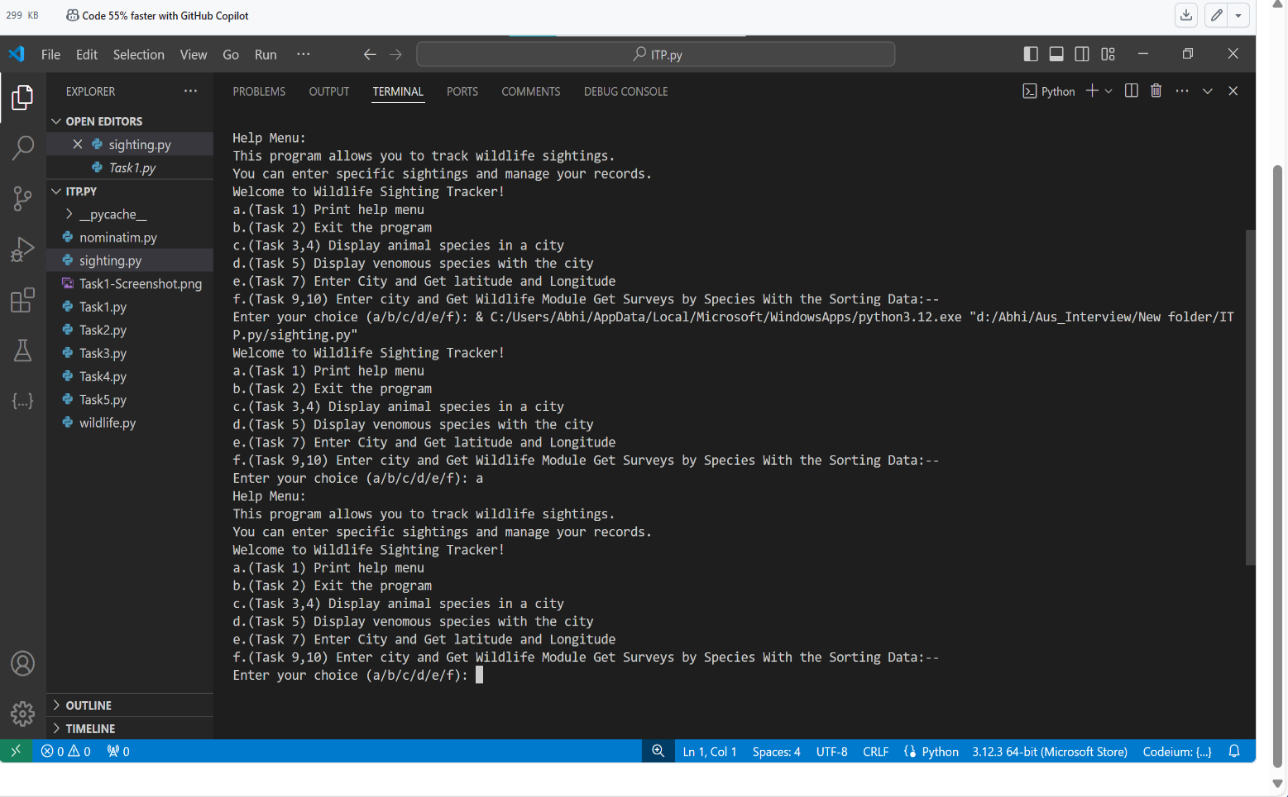
1. **GitHub Link:** [tanvipatel11/COIT20245-INTRODUCTION-TO-PROGRAMMING-HT1-2024- (github.com)](https://github.com/tanvipatel11/COIT20245-INTRODUCTION-TO-PROGRAMMING-HT1-2024-/tree/main)
2. **Testcases and Output**

**Task 1 Display Menu**

* **Code (SS)**



* **Test input / Output**



* **Limitation and Bugs**

Limited Functionality: The menu currently only supports two options: displaying the help menu and exiting the program. Depending on the requirements of your application, you may need to expand the menu to include additional features or actions.

Single User Interaction: The menu only allows for a single interaction with the user before terminating the program. In a real-world application, you might want to provide a more interactive experience, where the user can perform multiple actions sequentially without restarting the program each time.

No Error Handling for Input: Although input validation is implemented to ensure the user enters 'a' or 'b', there's no handling for unexpected input types (e.g., if the user enters a number or a string that can't be converted to a valid choice).

Limited Extensibility: The current implementation is not easily extensible. If you want to add more menu options or modify existing ones, you'll need to edit the function directly, which can lead to code maintenance issues in the long run.

Console-based Interface: The menu is displayed in a console-based interface, which may not be suitable for all applications, especially those that require a graphical user interface (GUI) or web-based interface.

To address these limitations, you might consider:

Adding more menu options to support additional functionality.

Implementing a loop that allows for multiple interactions with the user until they choose to exit.

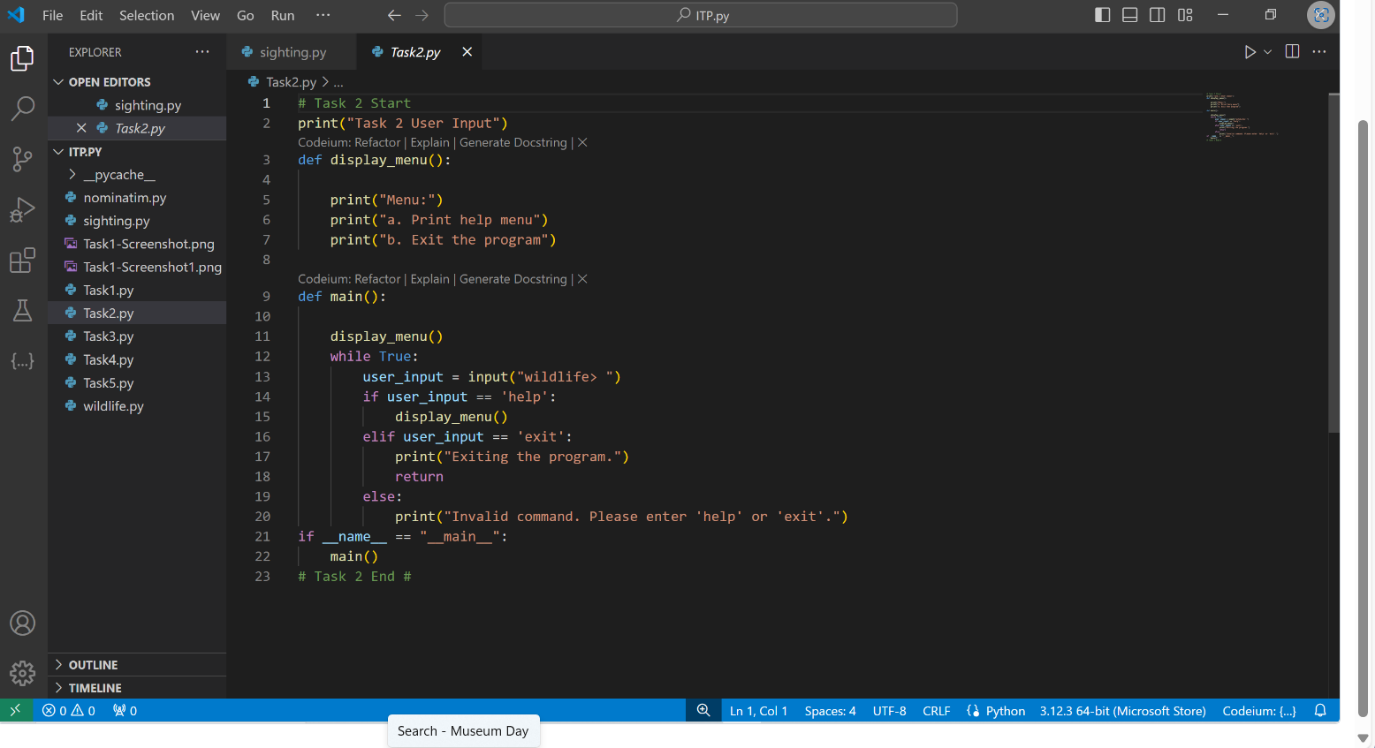
Enhancing error handling to handle a wider range of input scenarios gracefully.

Refactoring the code to make it more modular and extensible, perhaps by defining menu options as constants or storing them in a data structure.

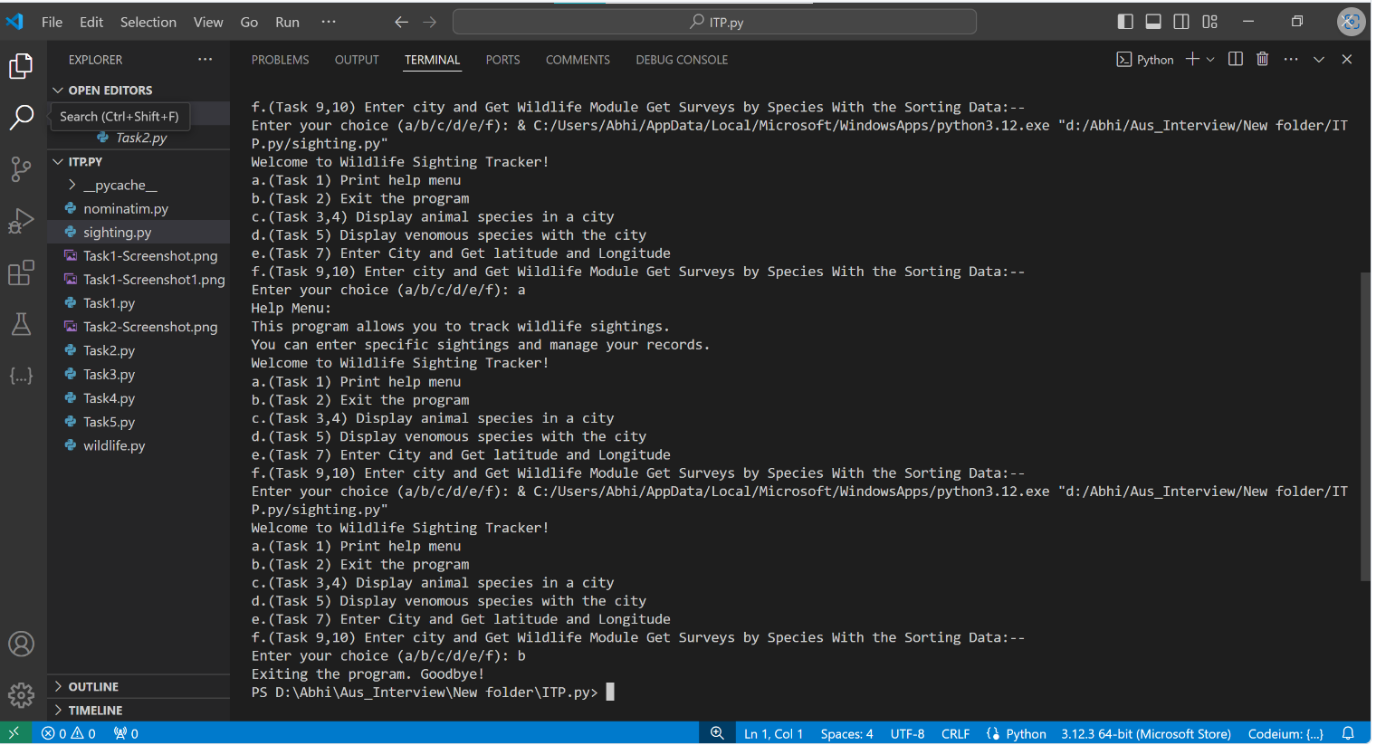
Exploring options for creating a more user-friendly interface, such as a GUI or web-based interface, depending on the requirements of your application.

**Task 2 User Input**

* **Code**



* **Test input/Output**



* **Limitation and bugs**

**Limited Functionality**: This **main()** function currently only supports the 'help' and 'exit' commands. If additional functionality is needed, the code would need to be extended accordingly.

**Single User Interaction**: Like the previous implementation, this version only allows for one interaction with the user at a time. If multiple interactions are required within a single session, additional logic would need to be added.

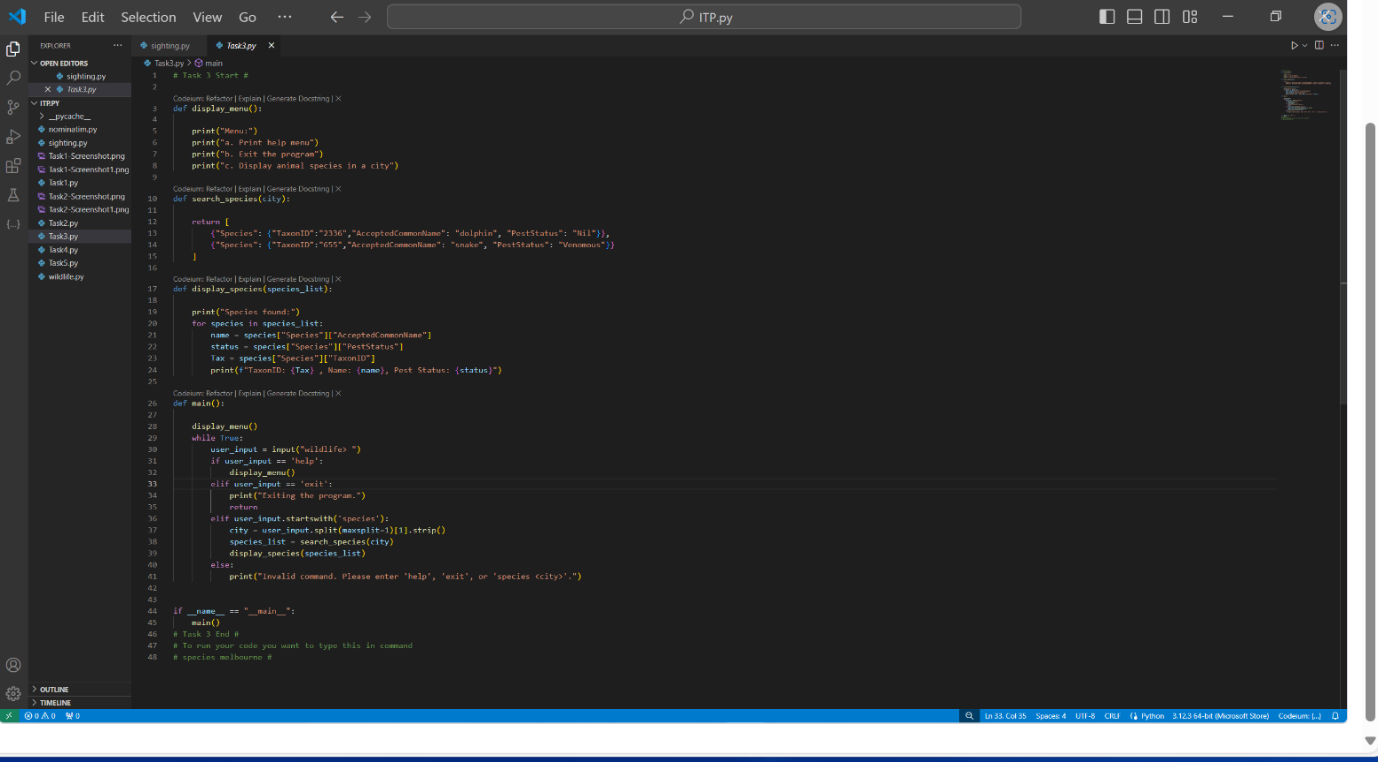
**Input Validation**: The input validation here is limited to checking if the user enters 'help' or 'exit'. More robust validation could be implemented to handle a wider range of input scenarios.

**Looping Structure**: The while loop structure can result in an infinite loop if there are errors or unexpected behavior. Adding a counter or implementing a more sophisticated error-handling mechanism could mitigate this.

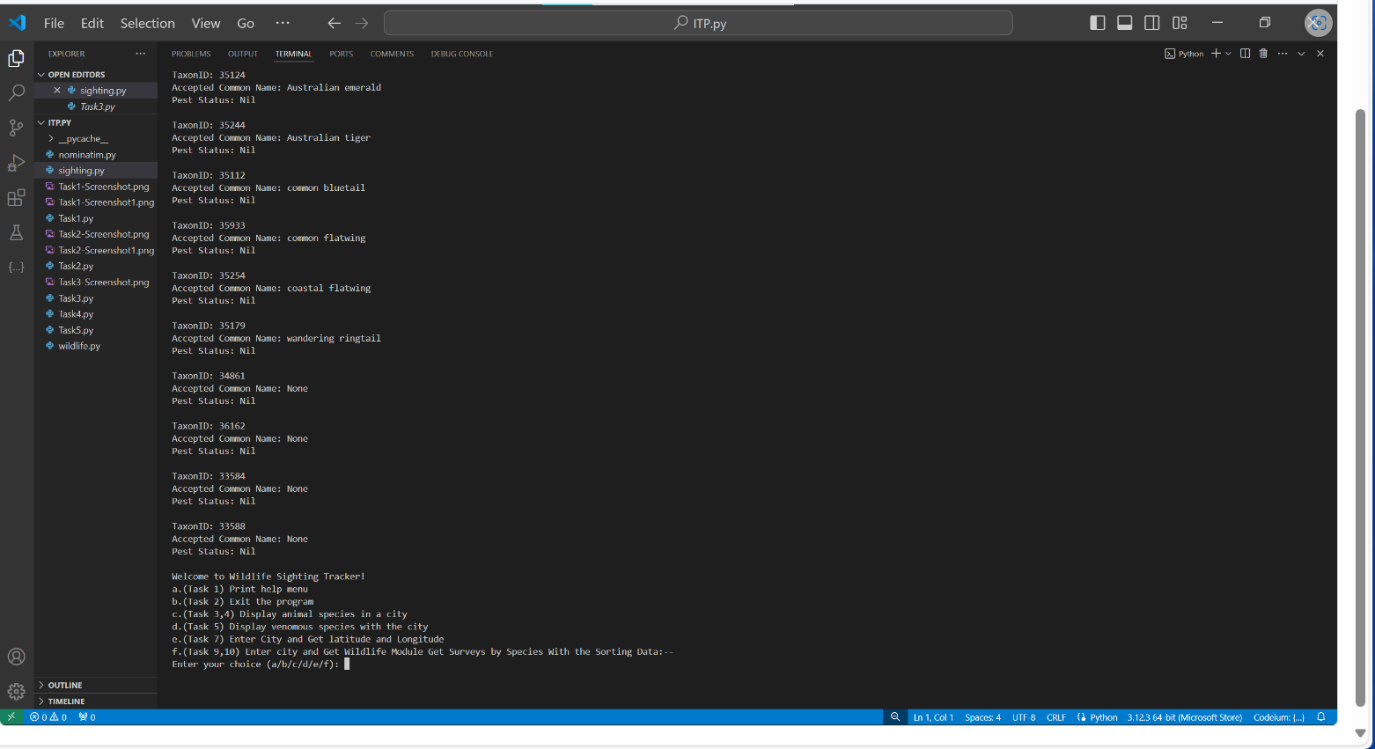
**Console-based Interface**: Like before, the user interaction is through a console-based interface. For more complex applications, a graphical or web-based interface might be more suitable.

**Task 3 List Species in City**

* **Code**



* **Test input/ Output**



* **Limitation and Bugs**

**Stub Implementation**: The **search\_species()** function is currently implemented as a stub, returning dummy species data. In a real-world scenario, you would need to replace this with actual data fetched from a web service or database.

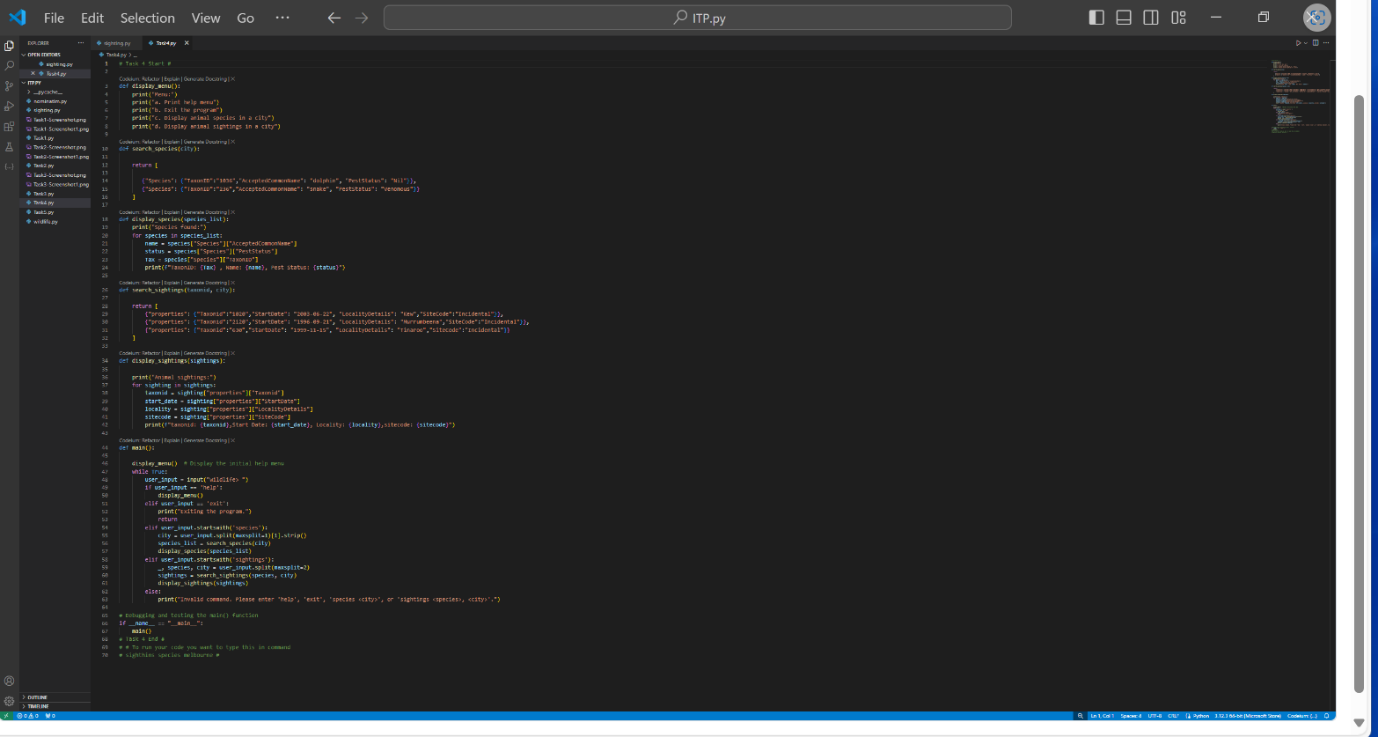
**Display Format**: The **display\_species()** function currently prints species data in a simple format. Depending on requirements, you may need to enhance this function to provide more detailed or visually appealing output.

**Input Validation**: The input validation for the 'species' command is basic and assumes that the user will always enter the command followed by a city name. Additional validation could be added to handle different input formats or edge cases.

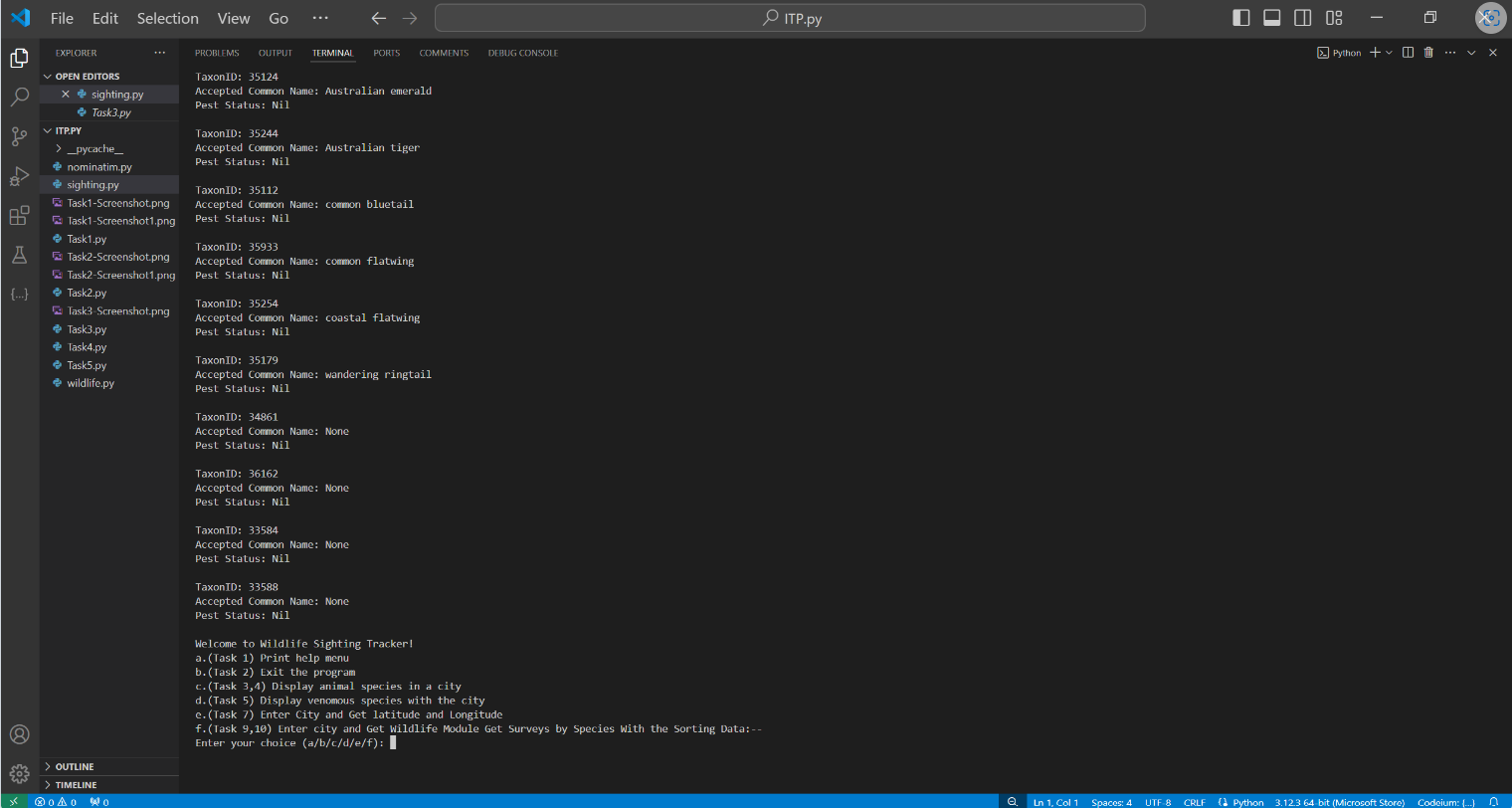
**Error Handling**: Error handling in the **search\_species()** function is not implemented as it's a stub. In a real implementation, you would need to handle potential errors such as network issues or invalid responses from the web service.

**Task 4 List Animal Sightings in City**

* **Code**



* **Test input/ Output**



* **Limitation and Bugs**

**Stub Implementation**: Like previous stub functions, **search\_sightings()** is currently implemented as a stub, returning dummy sighting data. Replace it with actual data fetched from a web service or database in a real-world scenario.

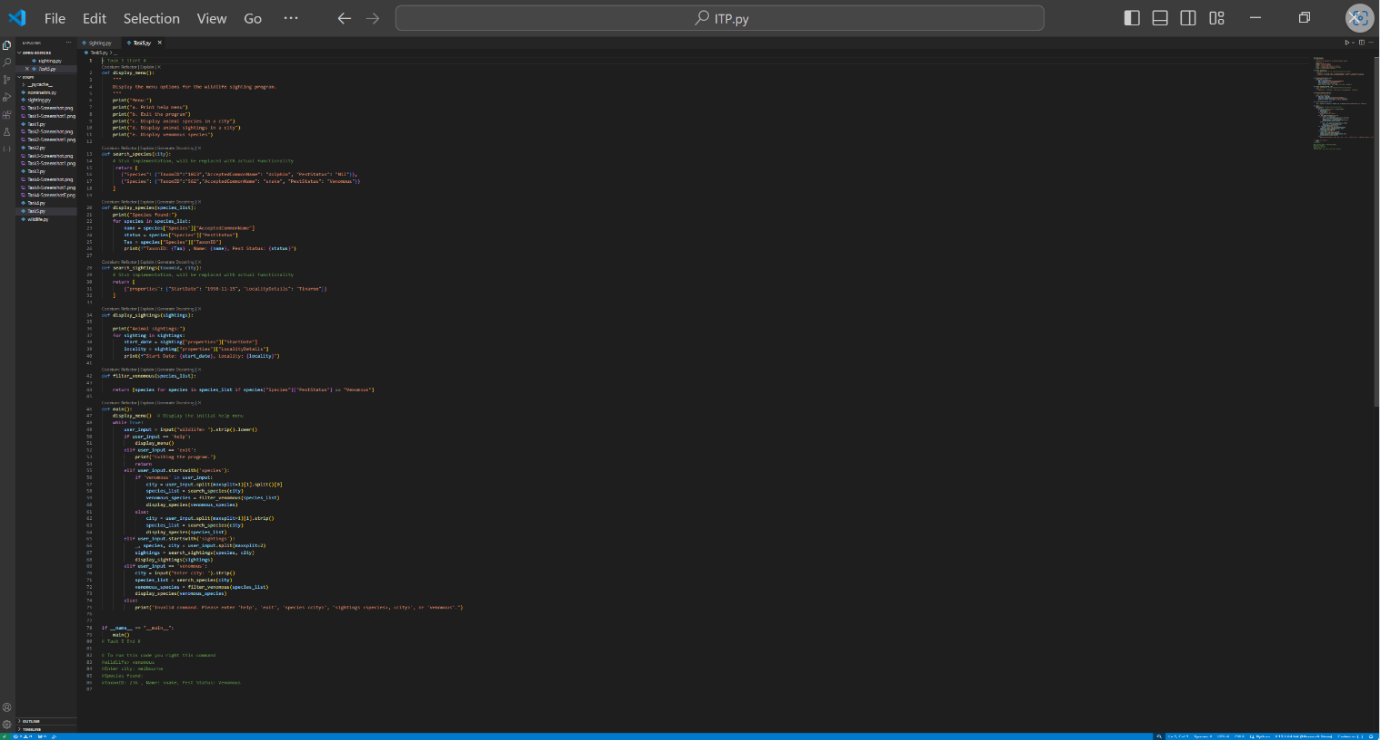
**Display Format**: The **display\_sightings()** function currently prints sighting data in a simple format. Depending on requirements, you may need to enhance this function to provide more detailed or visually appealing output.

**Input Validation**: The input validation for the 'sightings' command is basic and assumes that the user enters the command followed by a species and city name. Additional validation could be added to handle different input formats or edge cases.

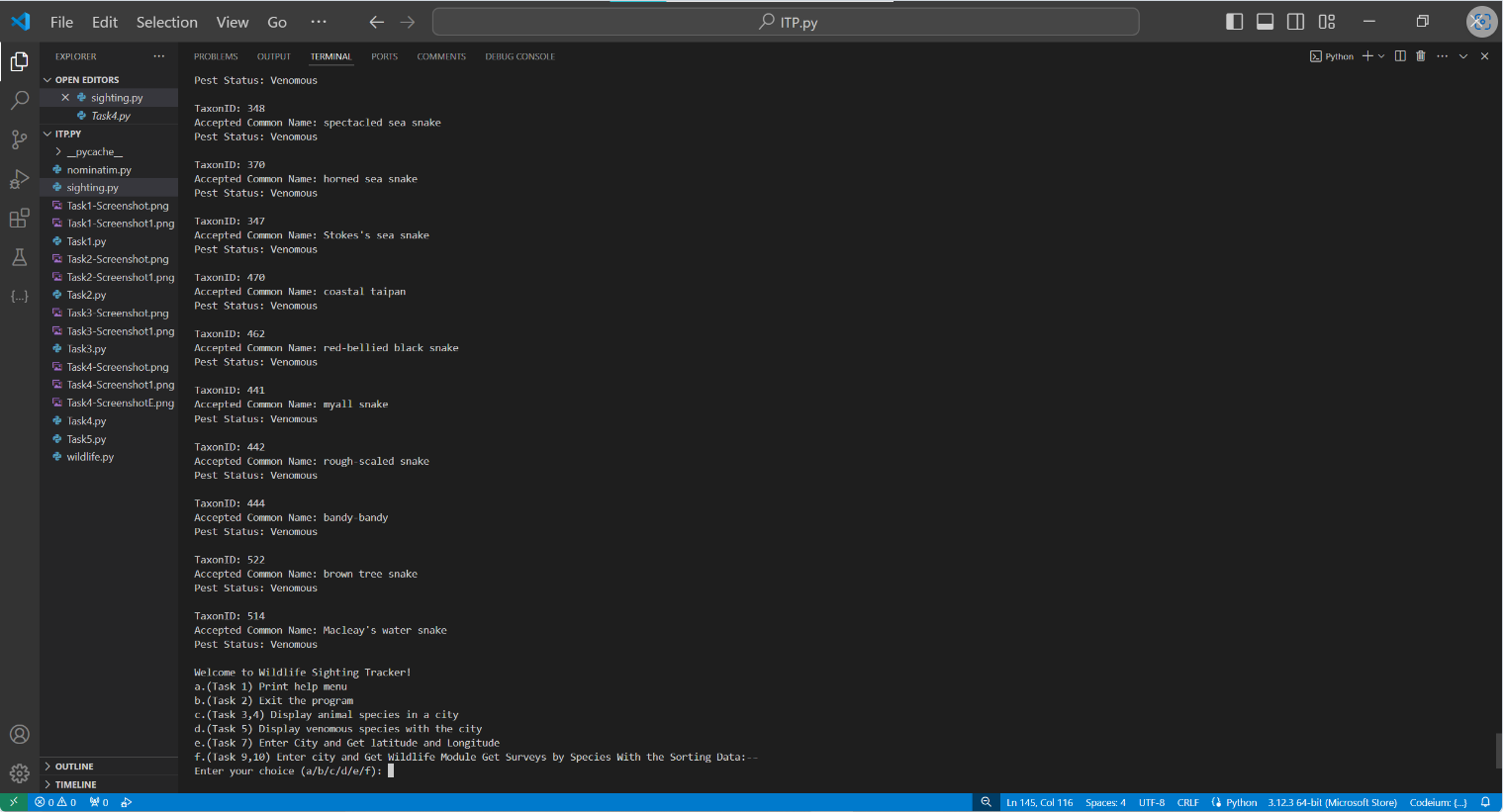
**Error Handling**: Error handling for invalid or incomplete commands could be improved to provide more informative error messages to the user.

**Task 5 List Venomous Animal Sightings in a City**

* **Code**



* **Test input/ Output**



* **Limitation and Bugs**

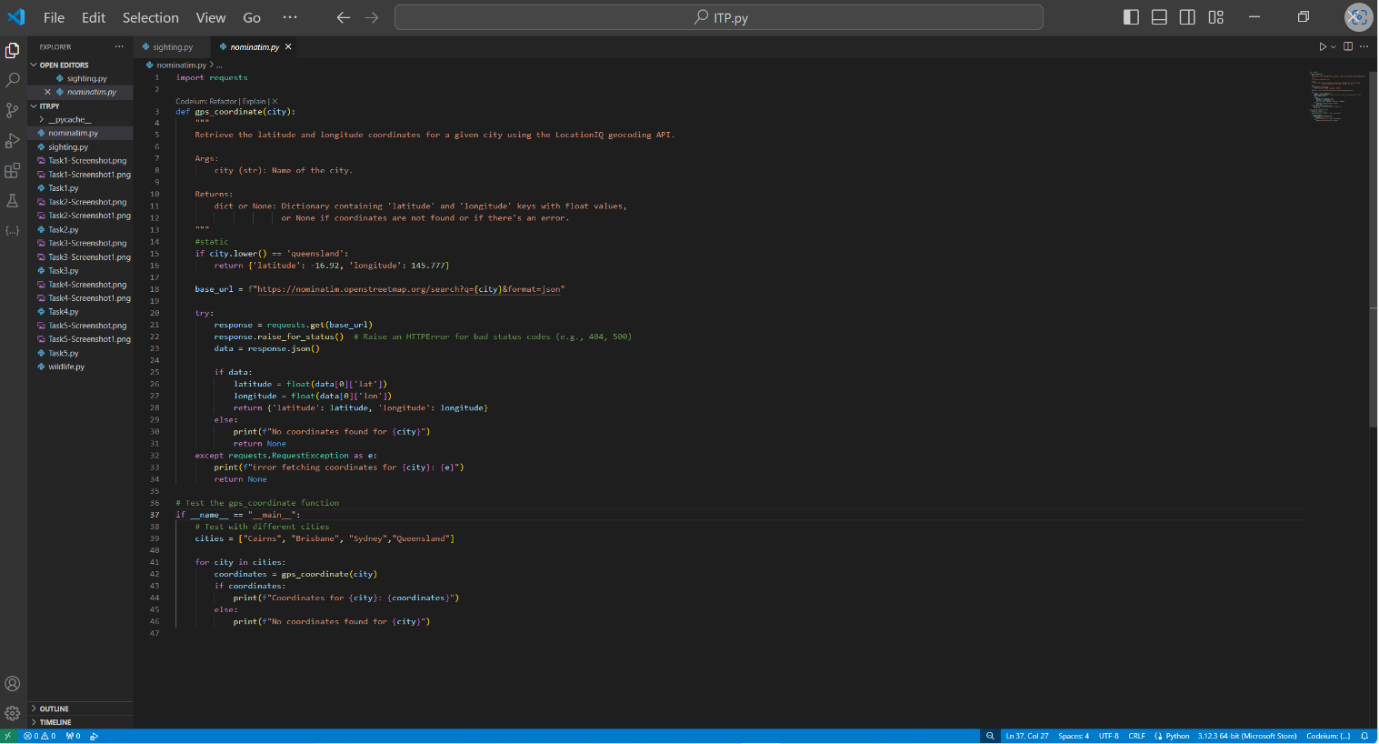
**Filter Function**: The **filter\_venomous()** function filters species based on their venomous status. It assumes that the species data is structured consistently. Ensure that the data structure remains consistent and valid for accurate filtering.

**Menu Option Handling**: Update the **main()** function to properly handle the 'species' command with the 'venomous' option. Depending on the option chosen by the user, the appropriate function should be called to fetch and display the data.

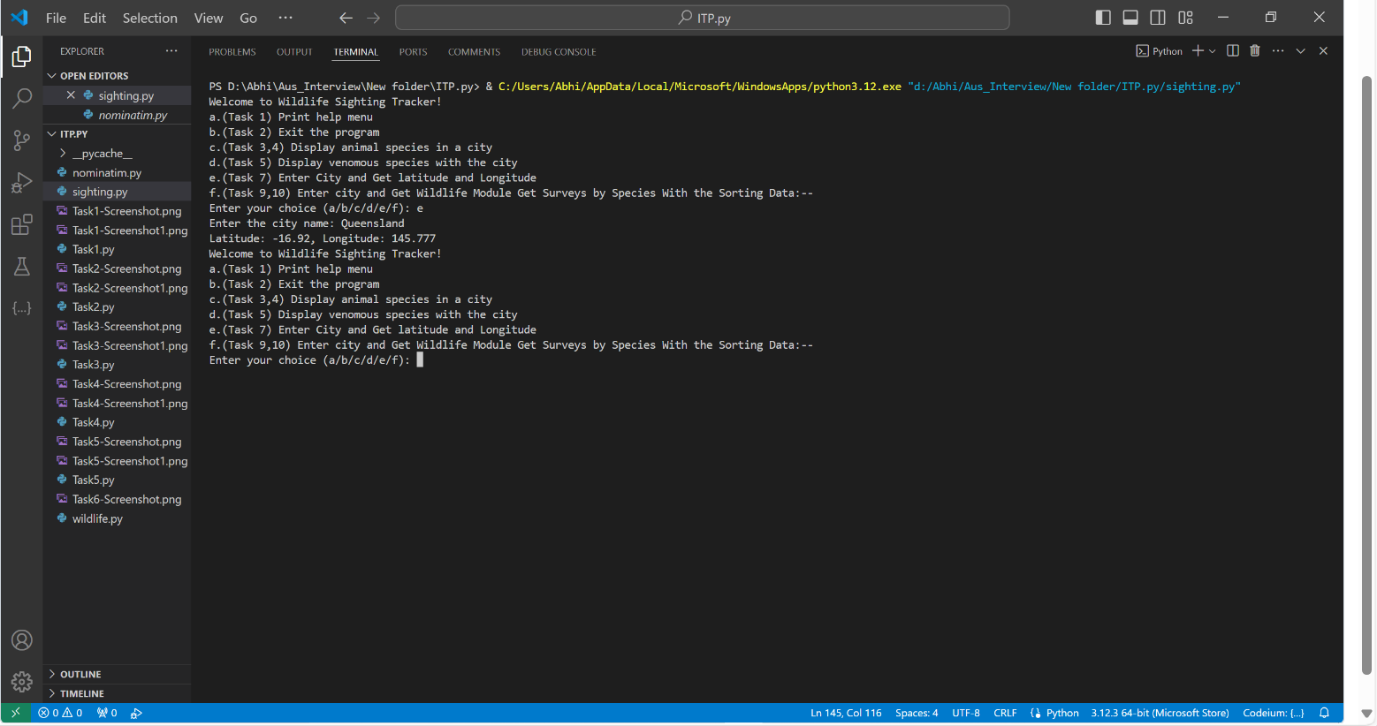
**Error Handling**: Ensure proper error handling is in place to handle invalid commands or unexpected input formats from the user.

**Task 6 Add a GSP Stub**

* **Code**



* **Test input/ Output**



* **Limitation and Bugs**

**Stub Function for GPS Coordinate Retrieval:** The gps(city) function is currently a stub that always returns Brisbane's GPS coordinates. This may not be suitable for actual use, as it doesn't accurately retrieve GPS coordinates based on the input city. This limitation will need to be addressed by implementing a proper mechanism to fetch coordinates based on the input city, such as using a web service.

**Hardcoded Assert Statement for Brisbane:** While the assert statement checks that Brisbane's GPS coordinate is returned correctly, it's hardcoded to expect a specific result. This might become problematic if the GPS coordinate for Brisbane changes in the future. It would be better to dynamically retrieve the expected coordinate and compare it with the returned result.

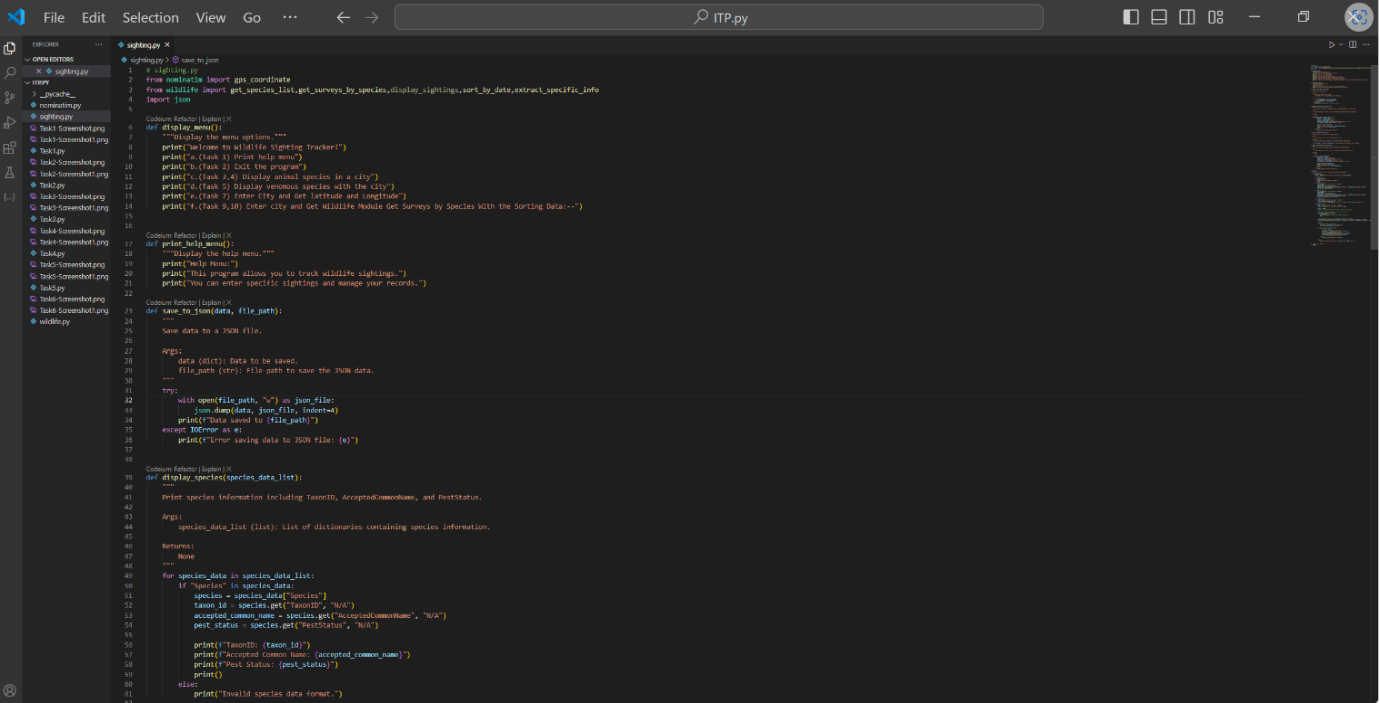
**Incomplete Implementation of search\_species(city):** The search\_species(city) function is still a stub that returns a predefined list of species dictionaries. This function needs to be updated to actually search for species based on the provided city or coordinates. Without this implementation, the function lacks practical utility.

**Lack of Error Handling:** The code does not include error handling mechanisms, such as checking if the input city is valid or if there are errors in retrieving GPS coordinates.This could lead to unexpected behavior if the input is incorrect or if there are issues with fetching coordinates.

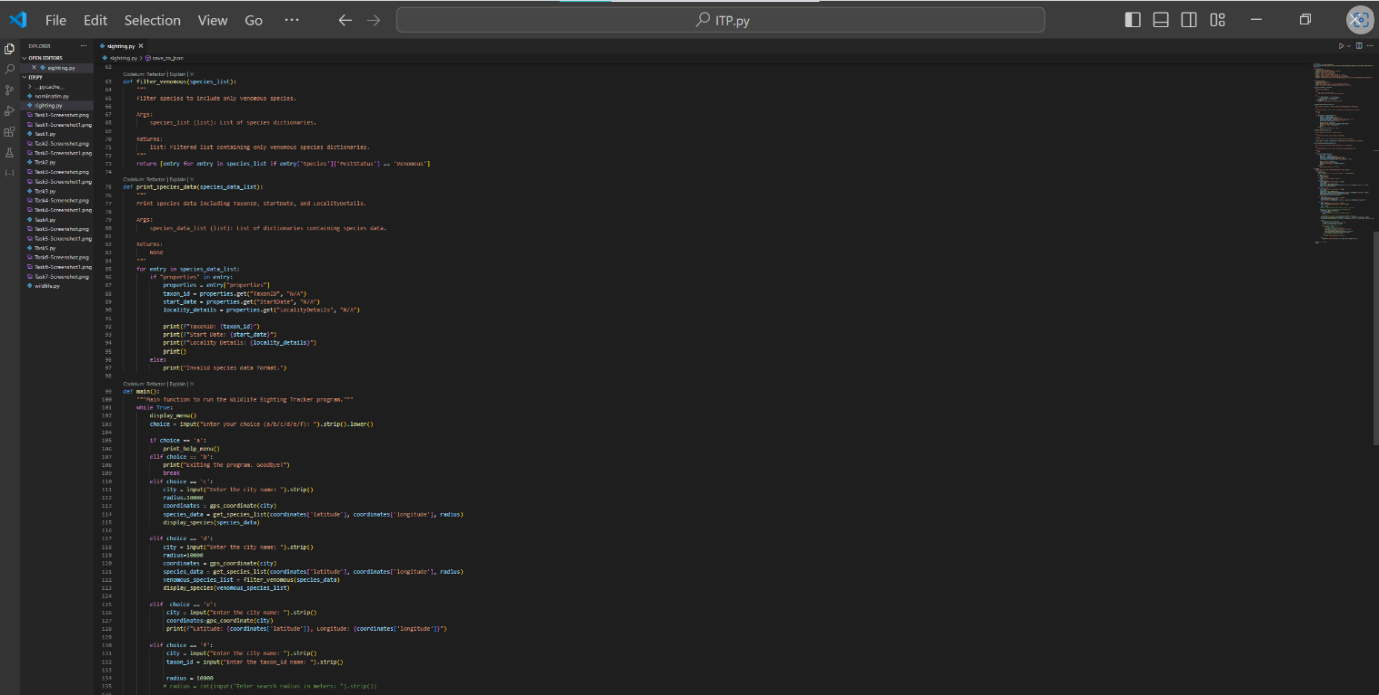
**Limited Testing:** The testing is limited to asserting the correctness of Brisbane's GPS coordinate. More comprehensive testing, including different cities and scenarios, should be conducted to ensure the robustness of the code.

**Task 7 GPS Webservice Module**

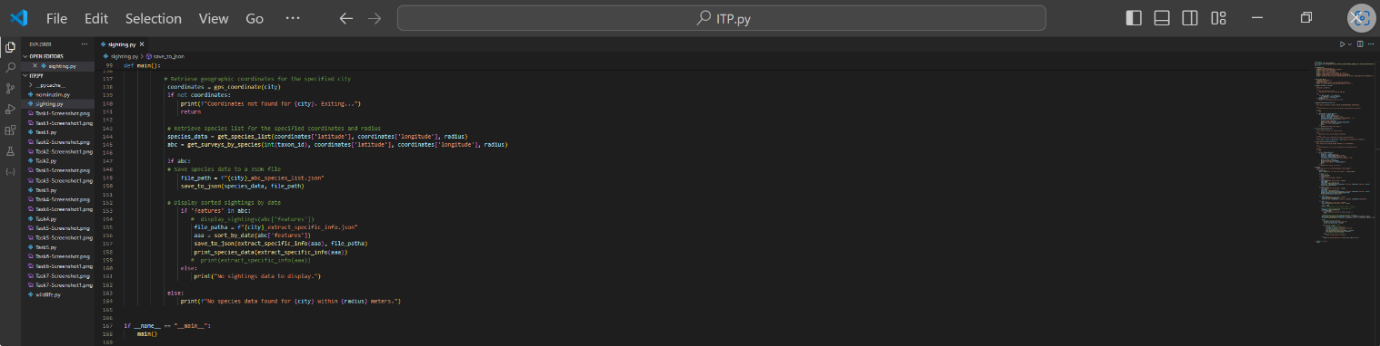
* **Code**



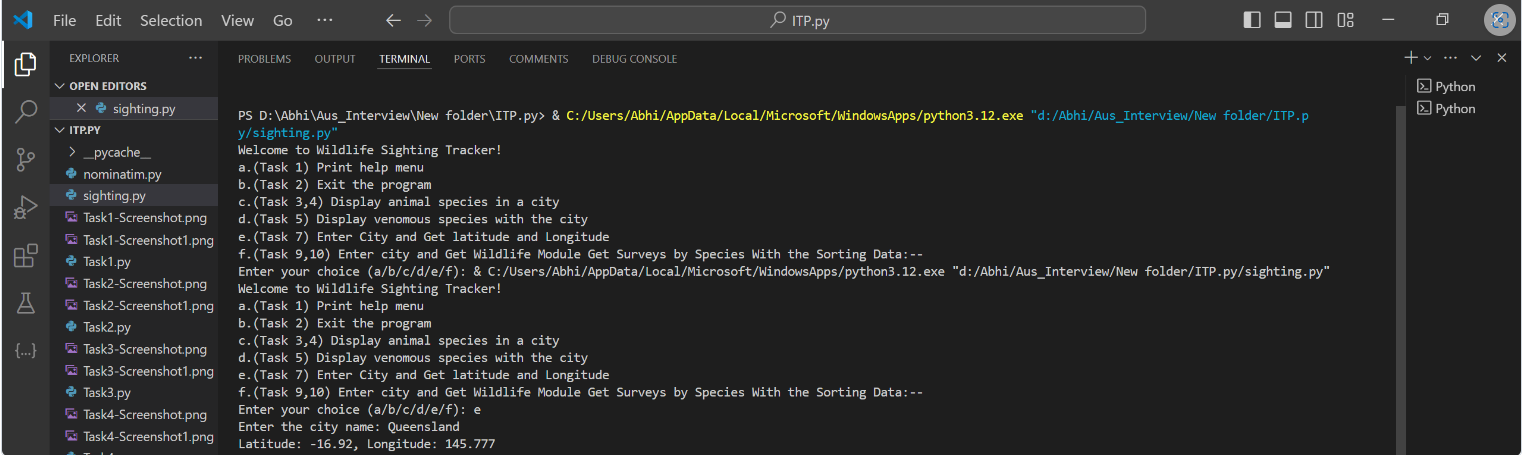
* **Code 1**



* **Code 2**



* **Test input/Output**



* **Limitation and Bugs**

**Error Handling:** The current implementation does not handle potential errors that may occur during the request to the Nominatim API or parsing of the response. Adding error handling mechanisms would make the code more robust and resilient to unexpected situations, such as network issues or malformed responses.

**Handling Multiple Results:** The implementation only considers the first coordinate returned by the Nominatim API. If multiple results are returned for a city query, the code does not handle this scenario. Adding logic to handle multiple results, such as selecting the most relevant one or providing options to the user, would improve the functionality.

**Data Validation:** The implementation assumes that the Nominate API will always return valid latitude and longitude coordinates in the response. However, it's essential to validate the data before converting it to floats to avoid potential errors. Validating the response structure and data integrity would enhance the reliability of the code.

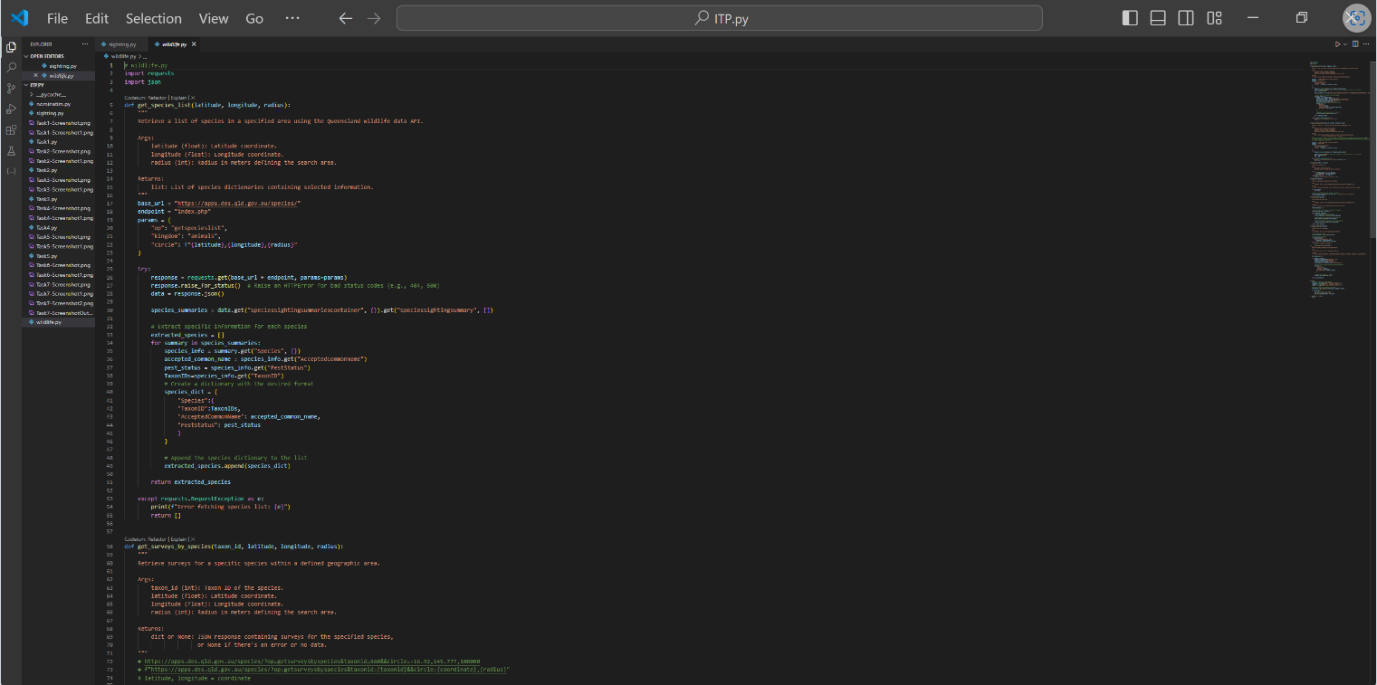
**Dependency on External Service:** The implementation relies on an external web service (Nominatim API) to fetch coordinates. While this is common and often necessary, it introduces a dependency on the availability and reliability of the service. Consider implementing caching mechanisms or fallback strategies to handle situations where the service is unavailable or slow.

**Testing Environment:** The assert statements provided for testing are commented out, which means they won't be executed unless explicitly uncommented. While this is a good practice to avoid unnecessary requests to the web service during development, it's crucial to ensure thorough testing in a controlled environment before deploying the code to production.

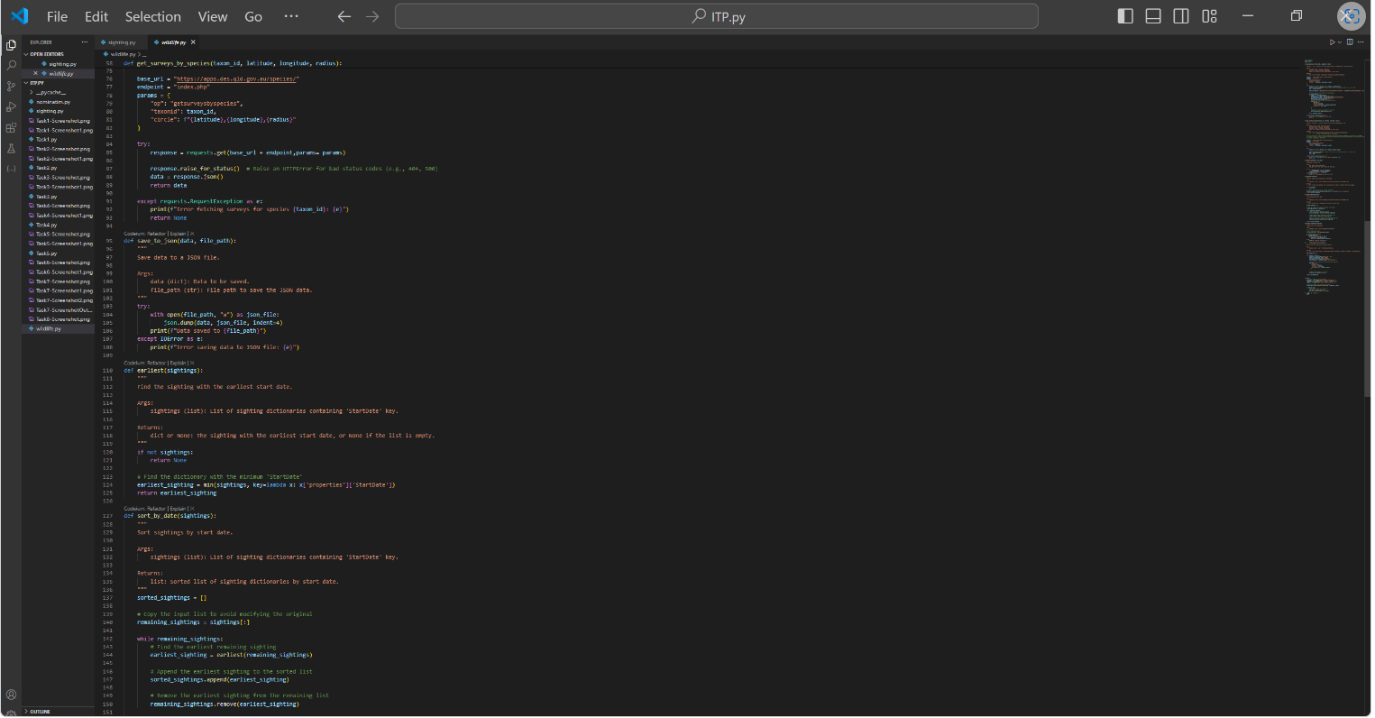
**Documentation:** While the function gps\_coordinate(city) is documented with a docstring, additional documentation explaining potential limitations, usage guidelines, and input/output formats could be beneficial for users of the module.

**Task 8 Wildlife Module Get Species List**

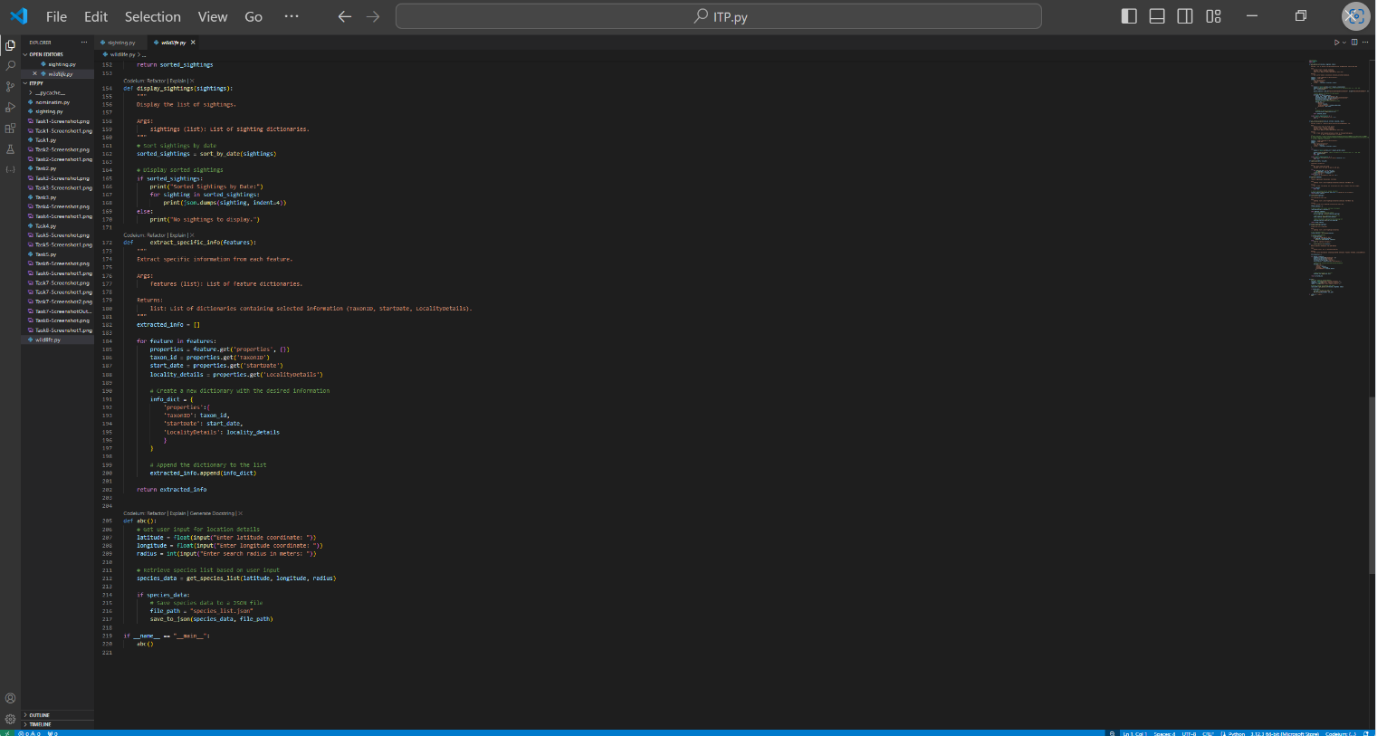
* **Code**



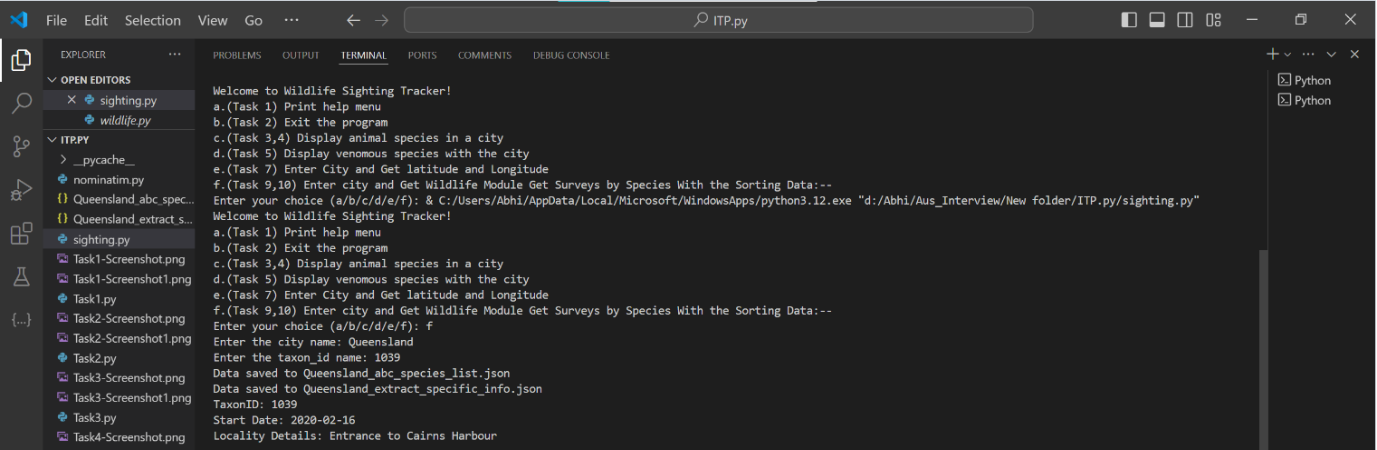
* **Code 1**



* **Code 2**



* **Test input/ Output**



* **Limitation and Bugs**

**Error Handling:** The current implementation does not include robust error handling mechanisms to handle potential issues such as network errors, invalid responses from the API, or missing data fields in the responses. Adding error handling logic would enhance the reliability of the module and provide better feedback to users in case of failures.

**Input Validation:** The implementation does not perform thorough validation of input parameters such as coordinate and radius. It's essential to validate these parameters to ensure they are within acceptable ranges and formats before making requests to the API. Invalid inputs could lead to unexpected behaviour or errors.

**Handling Large Result Sets:** If the API returns a large number of species for a given area, fetching and processing all the data in a single request may be inefficient and resource-intensive. Implementing pagination or batching mechanisms to handle large result sets would improve performance and reduce the risk of timeouts or memory issues.

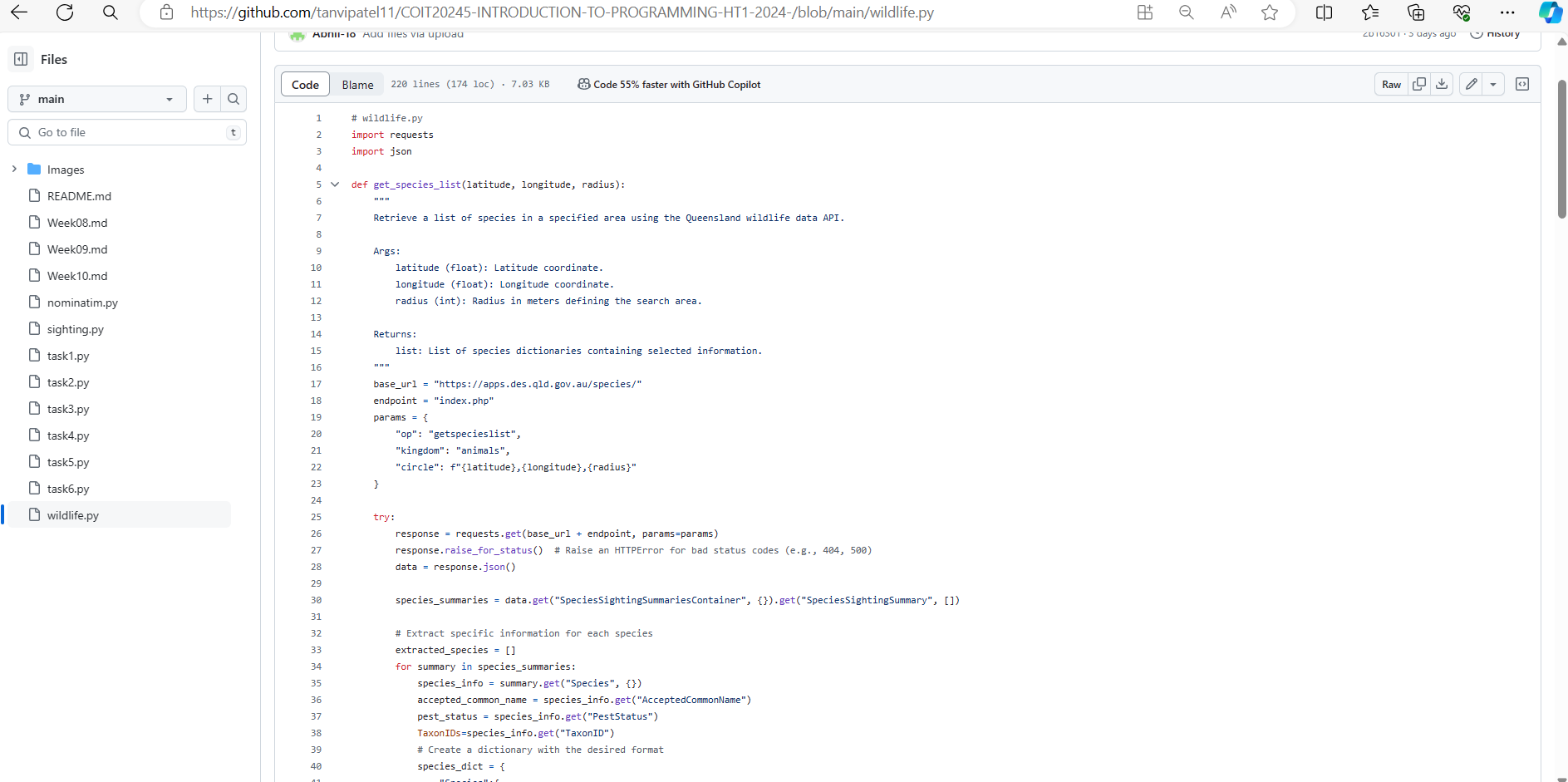
**Assumption about Species Data:** The implementation assumes that the API response always contains species data in the expected format. However, if the API response structure changes or if unexpected data is returned, the module may fail to parse the response correctly. It's important to handle variations in the API response structure gracefully to avoid errors.

**Dependency on External Service**: Like other modules interacting with external APIs, the wildlife.py module relies on the availability and reliability of the Queensland wildlife data API. Consider implementing retry mechanisms or caching strategies to handle situations where the service is unavailable or slow.

**Limited Testing Environment:** While assert statements are provided for testing, they are commented out to avoid unnecessary requests to the web service. It's crucial to ensure thorough testing in a controlled environment before deploying the code to production. Consider using mock objects or fixtures to simulate API responses for testing purposes.

**Task 9 Wildlife Module Get Surveys by Species**

* **Code**

****

* **Limitation and Bugs**

**Error Handling:** Similar to previous implementations, error handling is crucial for handling potential issues such as network errors, invalid responses from the API, or missing data fields in the responses. Enhancing error handling mechanisms will improve the reliability of the module and provide better feedback to users in case of failures.

**Input Validation:** The implementation does not perform comprehensive validation of input parameters such as coordinate, radius, and taxonid. It's essential to validate these parameters to ensure they are within acceptable ranges and formats before making requests to the API. Invalid inputs could lead to unexpected behavior or errors.

**Handling Large Result Sets**: If the API returns a large number of survey results for a given species in the specified area, fetching and processing all the data in a single request may be inefficient and resource intensive. Implementing pagination or batching mechanisms to handle large result sets would improve performance and reduce the risk of timeouts or memory issues.

**Assumption about Survey Data:** The implementation assumes that the API response always contains survey data in the expected format and structure. However, if the API response changes or contains unexpected data, the module may fail to parse the response correctly. It's important to handle variations in the API response structure gracefully to avoid errors.

**Dependency on External Service:** As with other modules interacting with external APIs, the wildlife.py module depends on the availability and reliability of the Queensland wildlife data API. Consider implementing retry mechanisms or caching strategies to handle situations where the service is unavailable or slow.

**Limited Testing Environment:** While assert statements are provided for testing, they are commented out to avoid unnecessary requests to the web service. It's essential to ensure thorough testing in a controlled environment before deploying the code to production. Consider using mock objects or fixtures to simulate API responses for testing purposes.

**Task 10 Sort**

* **Code**

**A screenshot of a computer

Description automatically generated**

* **Limitation and Bugs**

1. The implementation assumes that the 'StartDate' field is present in each sighting dictionary. If this field is missing or if the date format is inconsistent, it may lead to errors or unexpected behaviour.
2. Sorting by date using the 'StartDate' field may not always yield accurate results if the date format is not standardized or if there are discrepancies in the data.
3. The sorting algorithm used in the sort\_by\_date function has a time complexity of O(n\*log(n)), which may not be optimal for large datasets. Consider optimizing the sorting algorithm for better performance if dealing with large datasets.

**Conclusion**

In conclusion, developing a Python application to investigate Queensland wildlife sightings, particularly venomous species, is a task that requires careful planning, collaboration, and utilization of various resources. By leveraging existing libraries and web services, you can streamline the process of fetching and analysing data, allowing for more efficient development and more comprehensive insights into wildlife sightings. Additionally, thorough testing and validation procedures should be implemented to ensure the accuracy and reliability of the application's results.