C950 Task-2 WGUPS Write-Up

(Task-2: The implementation phase of the WGUPS Routing Program).

(Zip your source code and upload it with this file)

Logan Arguello

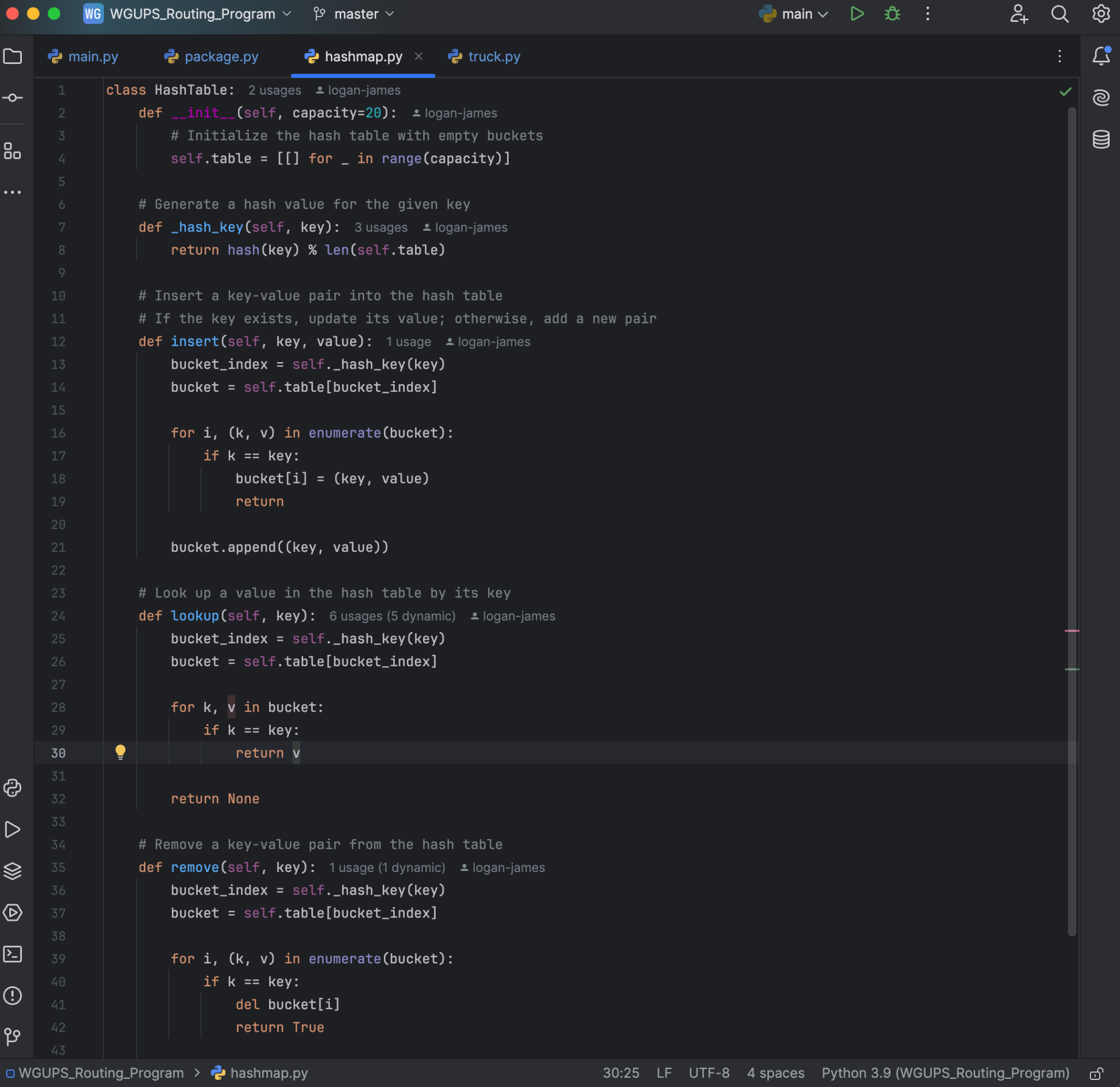
ID #012053764

WGU Email: largu10@wgu.edu

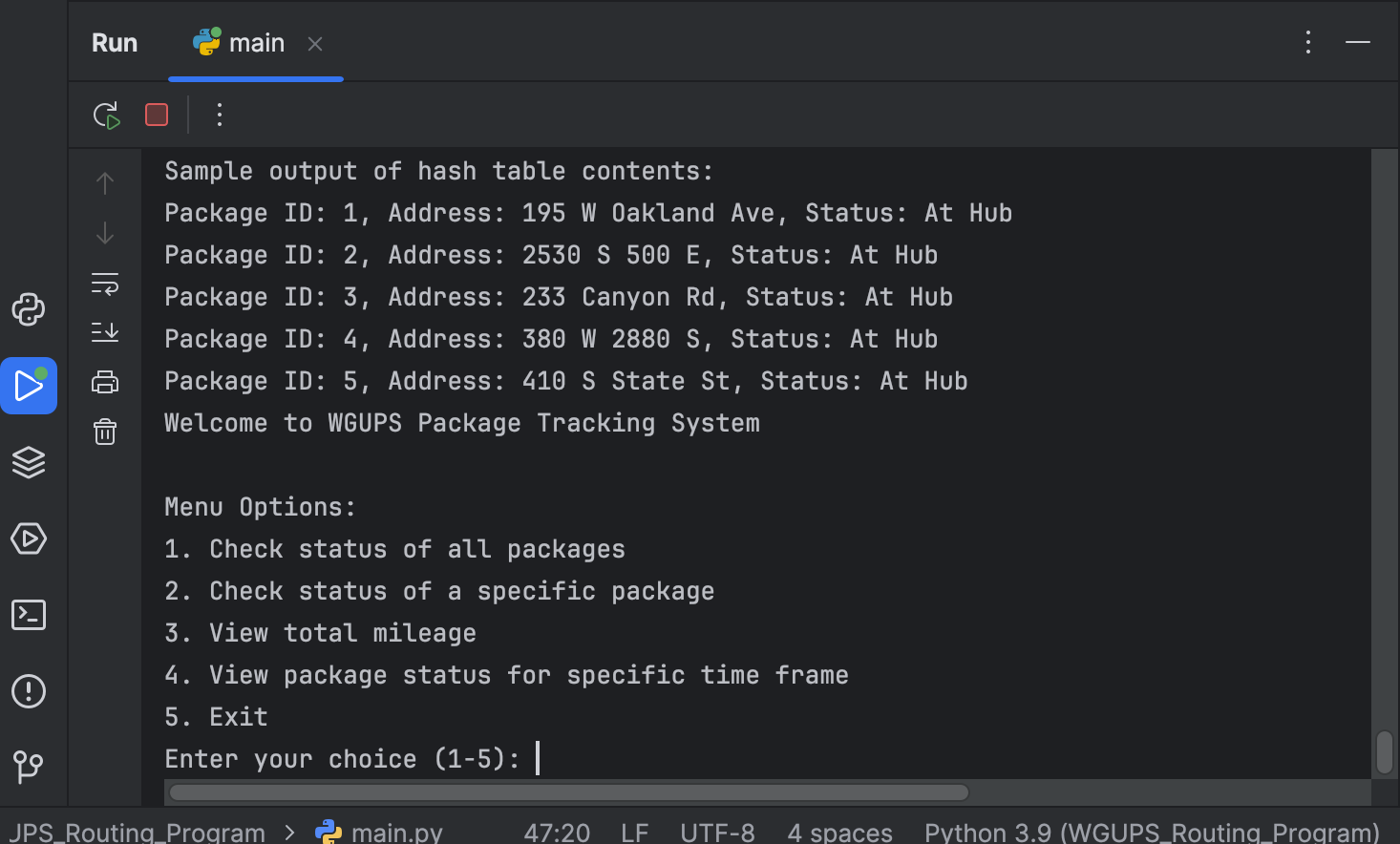
09/18/2024

C950 Data Structures and Algorithms II

# A. Hash Table

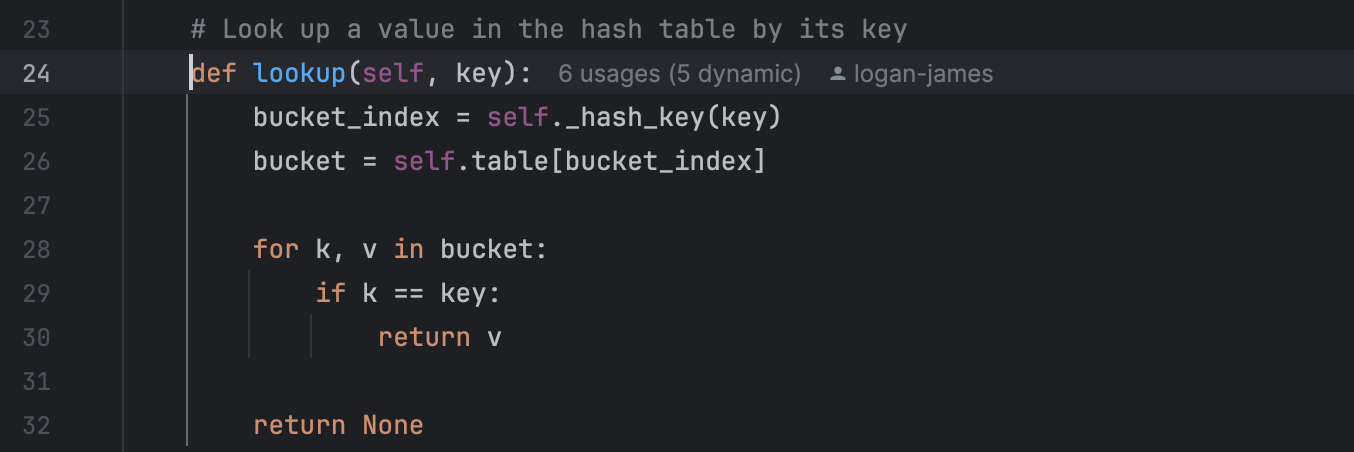


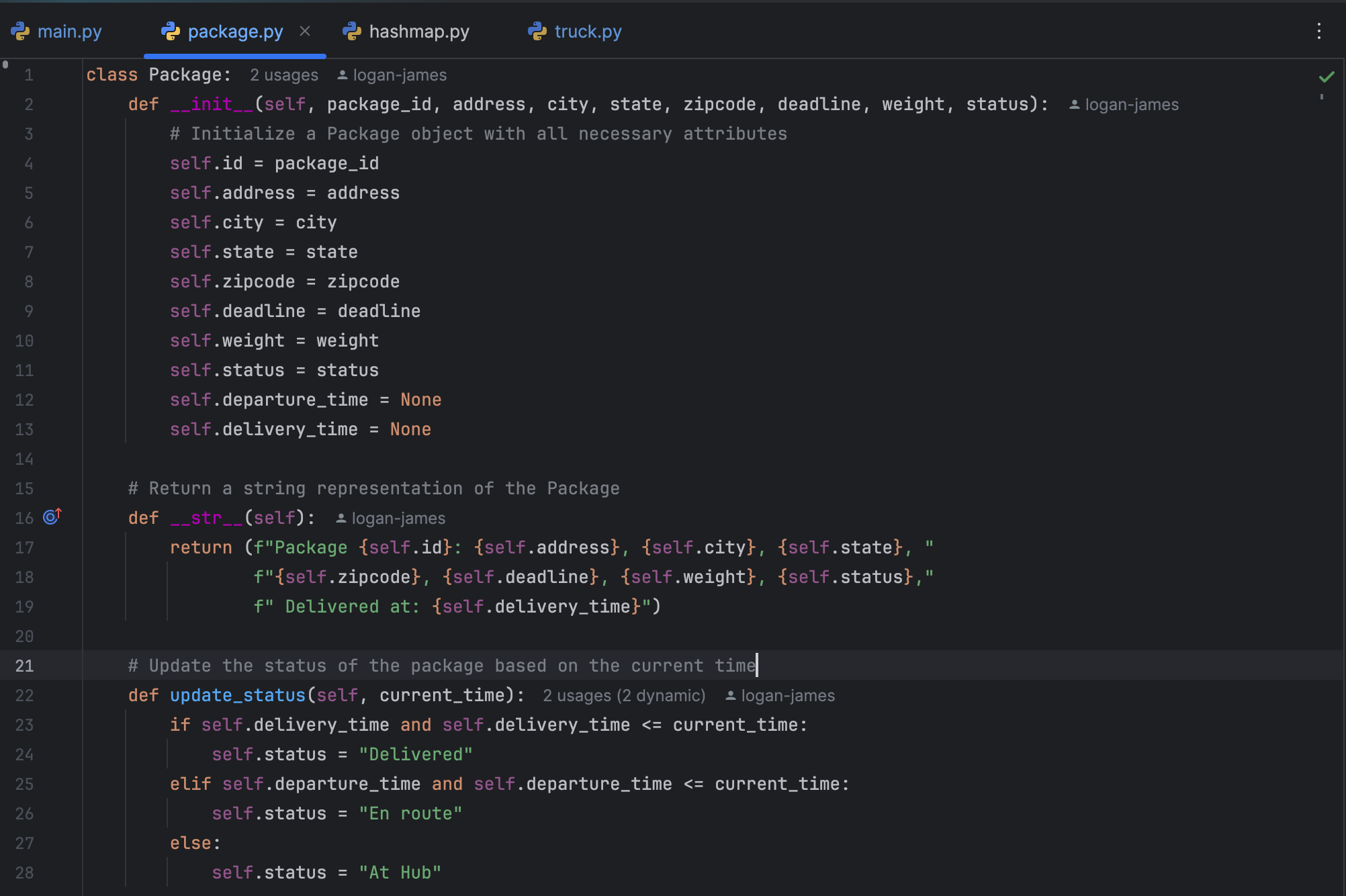
This screenshot shows my custom HashTable implementation. The class includes methods for insertion, lookup, and removal of key-value pairs. The insert method (lines 12-21) is capable of handling all required package data components. This implementation uses chaining for collision resolution and does not rely on any external libraries.



This screenshot demonstrates the creation and use of the custom hash table within the WGUPS package delivery system. The load\_packages() function initializes the hash table and inserts package data using the package ID as the key. To clearly present the functionality of the hash table, the function was temporarily modified with debugging code to print a sample output of the first few packages, displaying their ID, address, and status.

# B. Look-Up Functions





These screenshots demonstrate the look-up functionality of the WGUPS system:

Image 1 shows the lookup method in the HashTable class (hashmap.py). This method efficiently retrieves a Package object from the hash table using a given key (package ID).

Image 2 displays the Package class (package.py), which encapsulates all required data components for each package:

* delivery address (self.address)
* delivery deadline (self.deadline)
* delivery city (self.city)
* delivery zip code (self.zipcode)
* package weight (self.weight)
* delivery status (self.status)
* delivery time (self.delivery\_time)

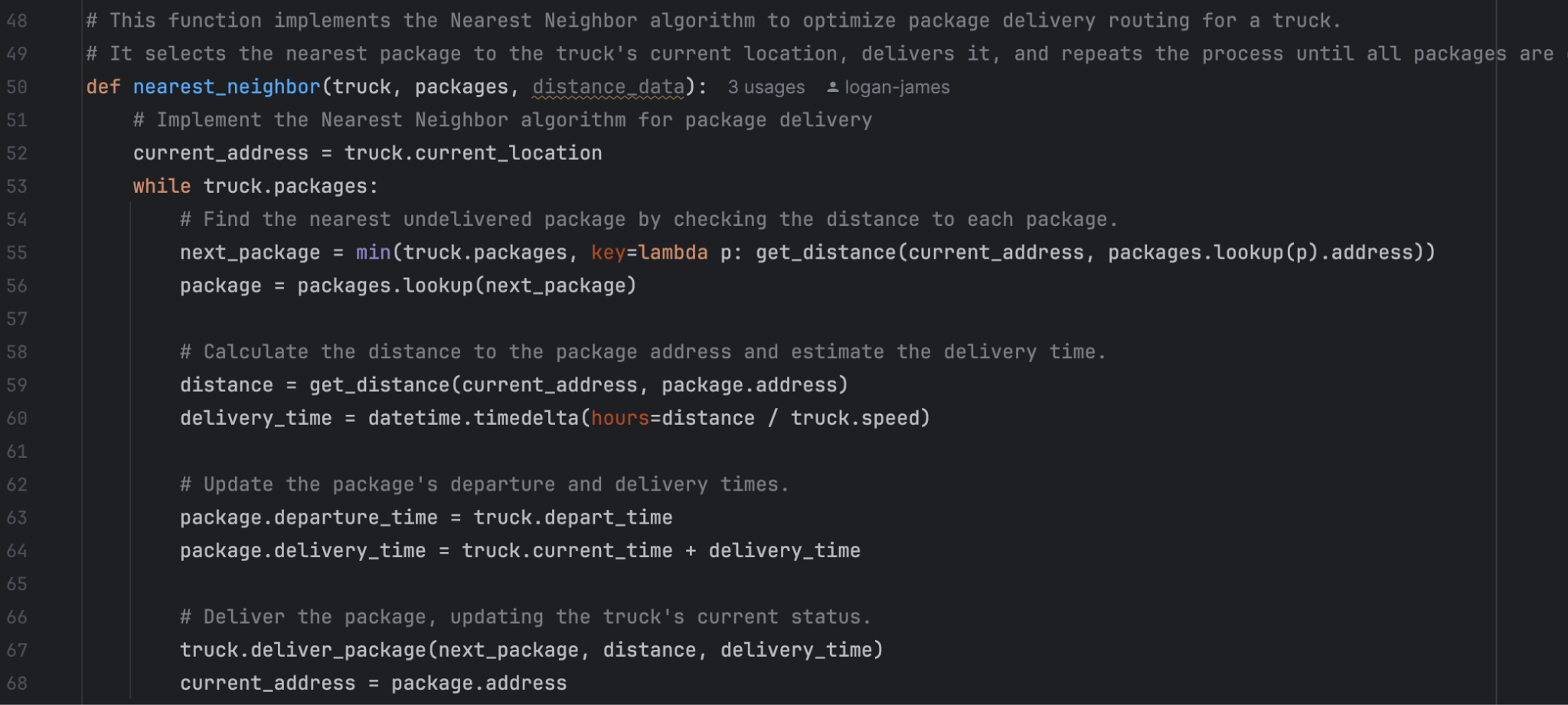
The combination of these two components allows for a complete look-up function:

1. The HashTable's lookup method retrieves the Package object using the package ID.
2. Once retrieved, all required data components can be accessed directly from the Package object's attributes.

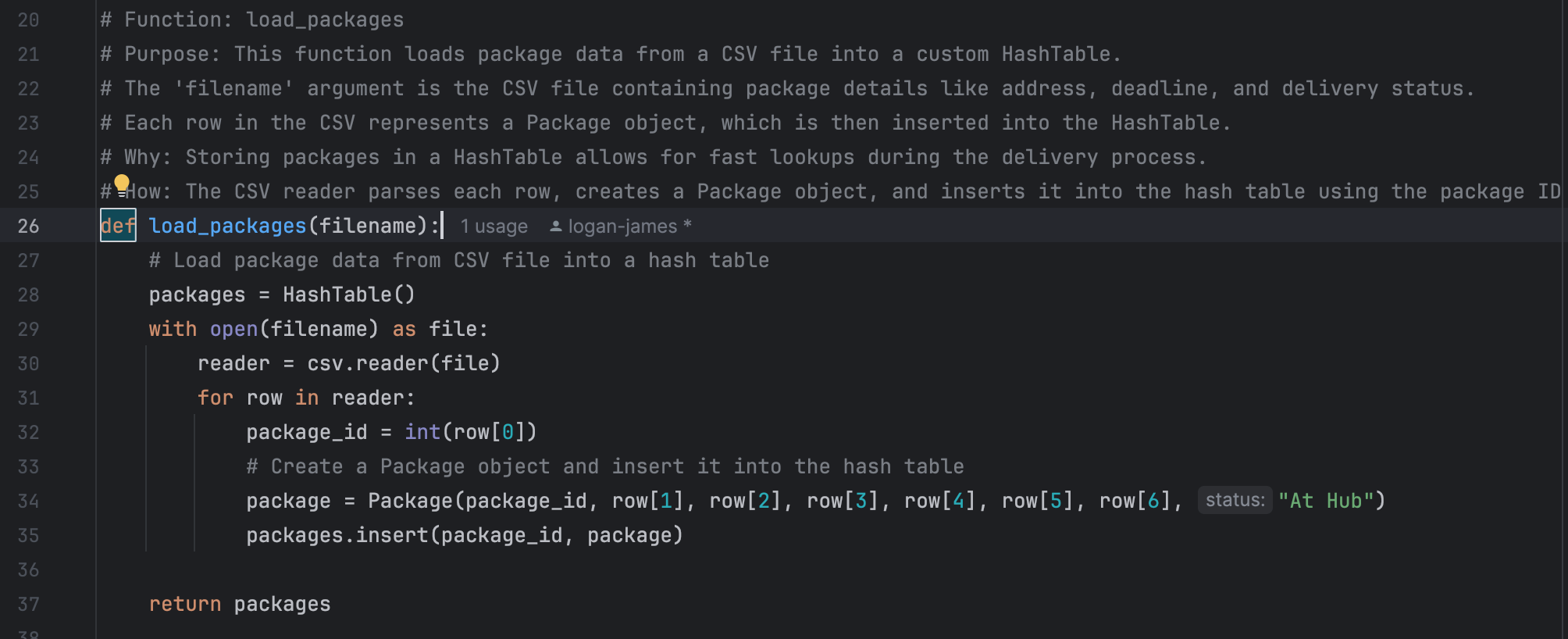
The update\_status method in the Package class ensures that the delivery status is always current based on the package's departure and delivery times.

This implementation fulfills the requirement of looking up each specific data point for a package using its ID as input, providing a comprehensive view of the package's current state and delivery information.

# C. Original Code

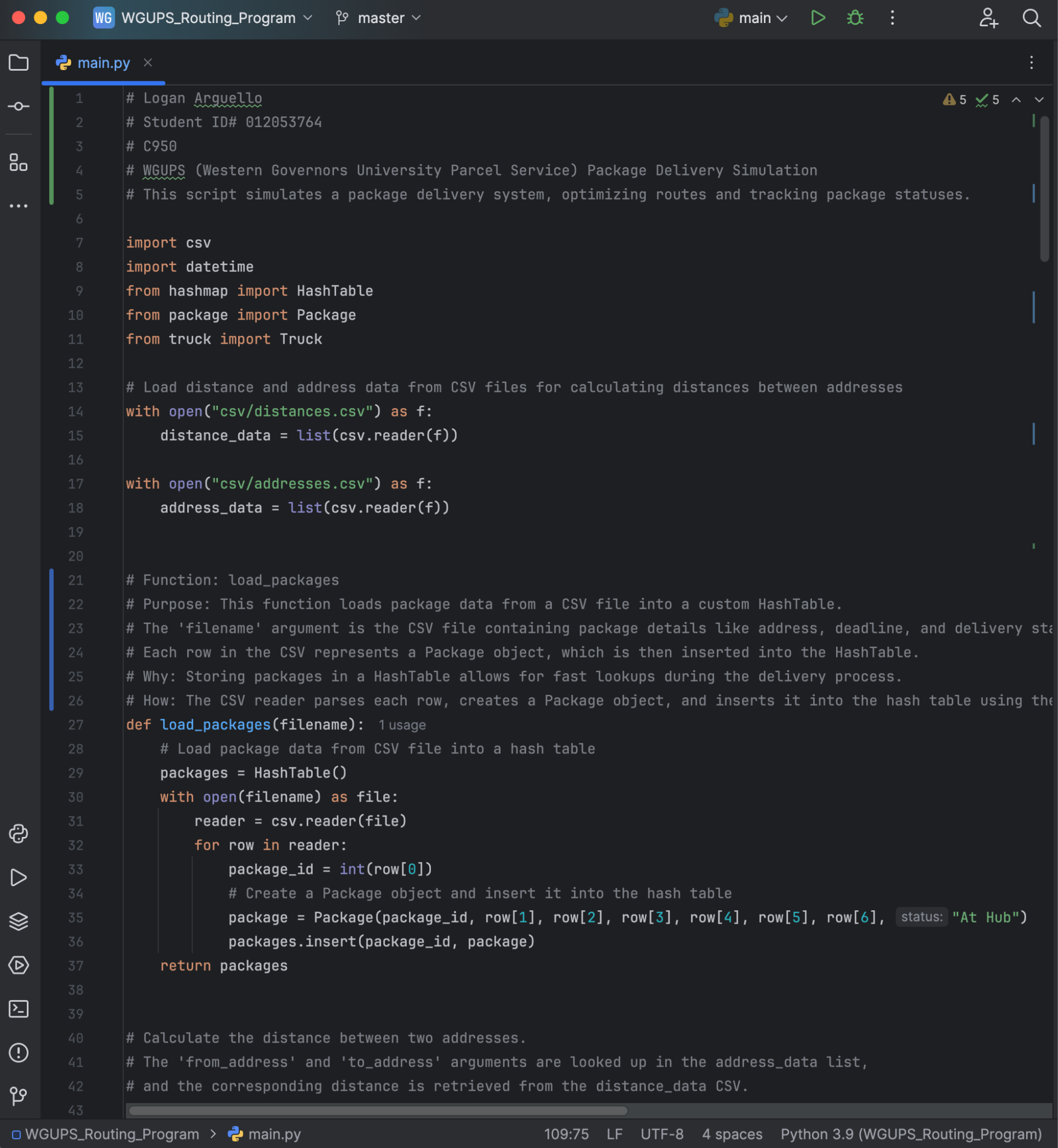


This code block implements the **Nearest Neighbor algorithm** for a package delivery system. It selects the nearest undelivered package to the truck’s current location, calculates the delivery time based on the distance and truck's speed, and updates the package's departure and delivery times. The truck’s current location is updated after each delivery, and the process repeats until all packages are delivered. This algorithm optimizes the routing by always delivering to the closest location next, helping to meet delivery efficiency and requirements based on provided data.

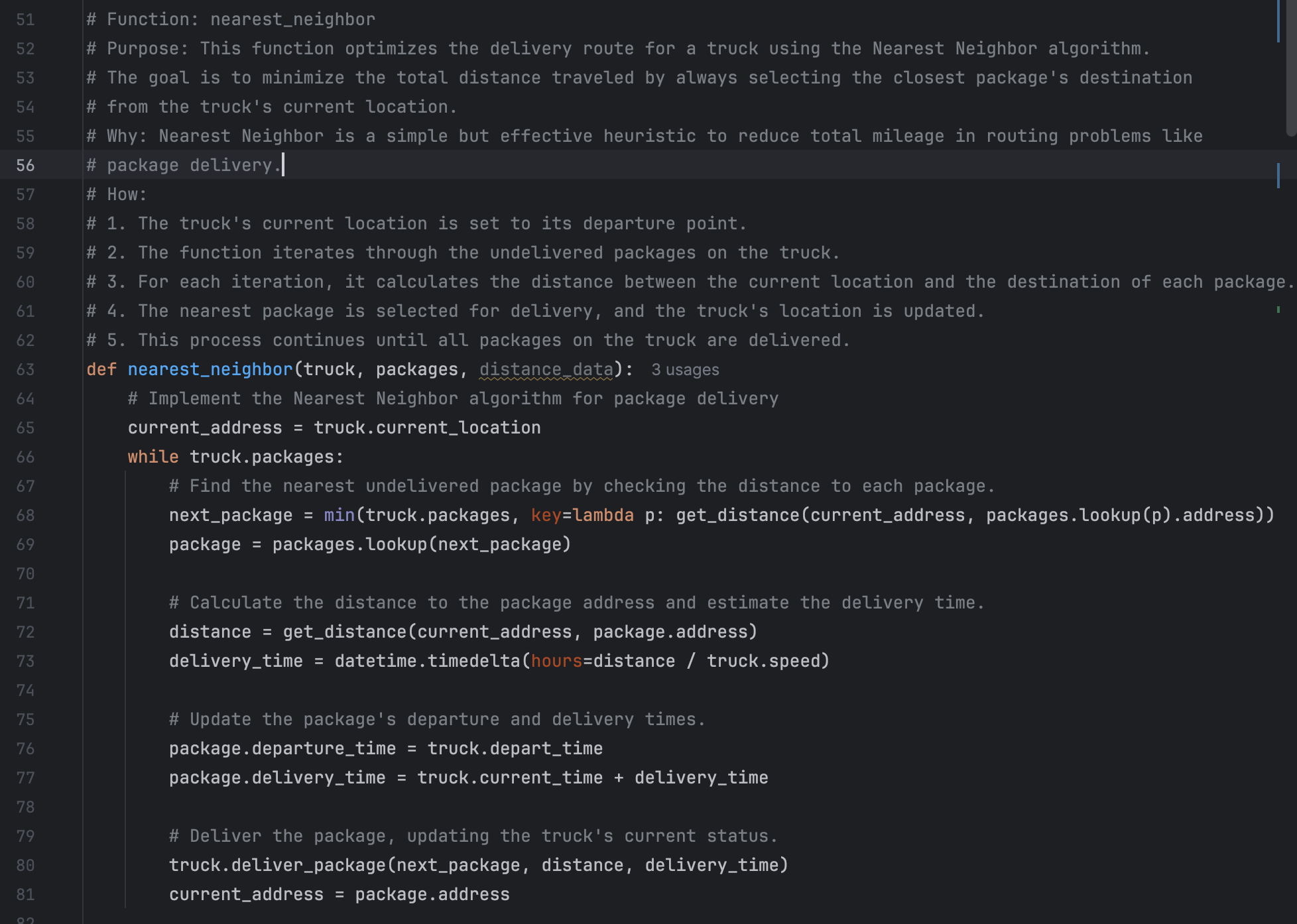


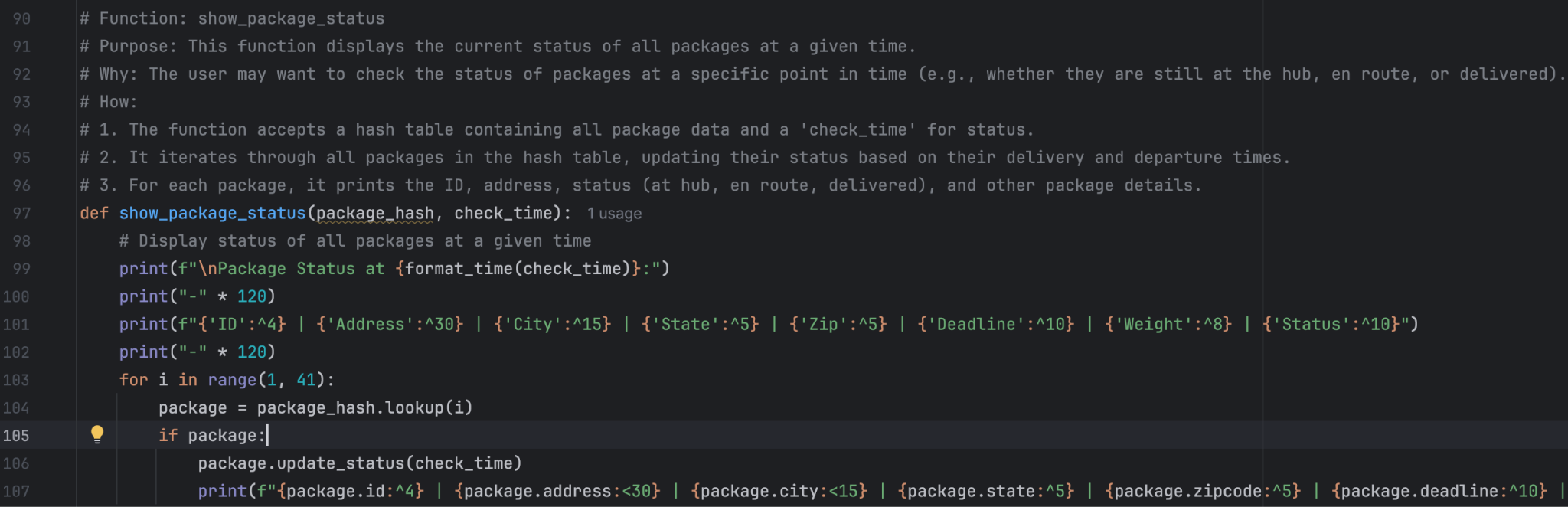
This code block defines a function called **load\_packages** that loads package data from a CSV file and stores it into a hash table. Each package is represented by a Package object and assigned a unique package\_id based on the data in the CSV. The function reads through each row in the CSV file, creates a Package object from the row data, and inserts it into the hash table using the package ID as the key. This ensures that all package information is efficiently stored and retrievable for further processing in the package delivery system.

# C1. Identification Information



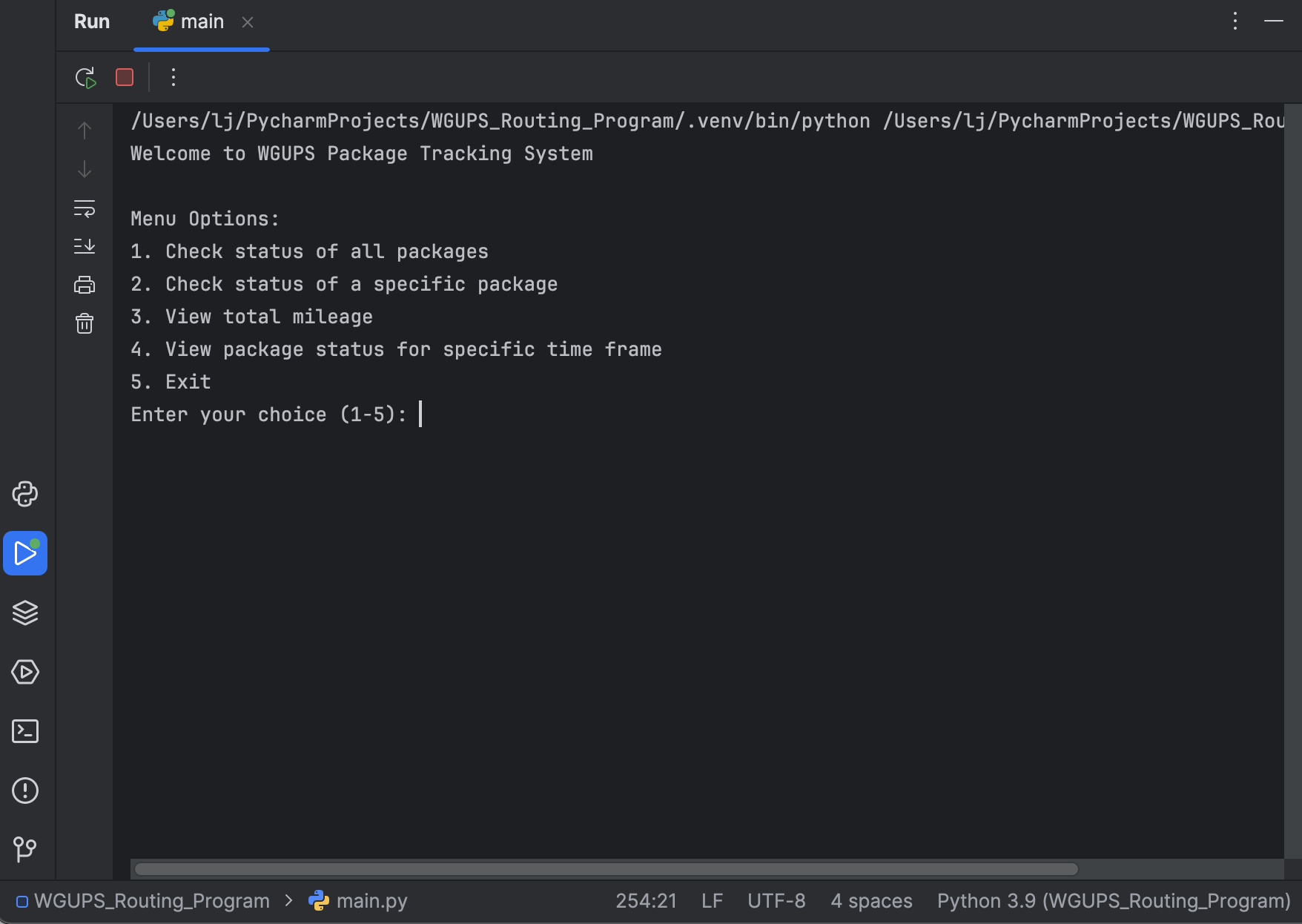
# C2. Process and Flow Comments

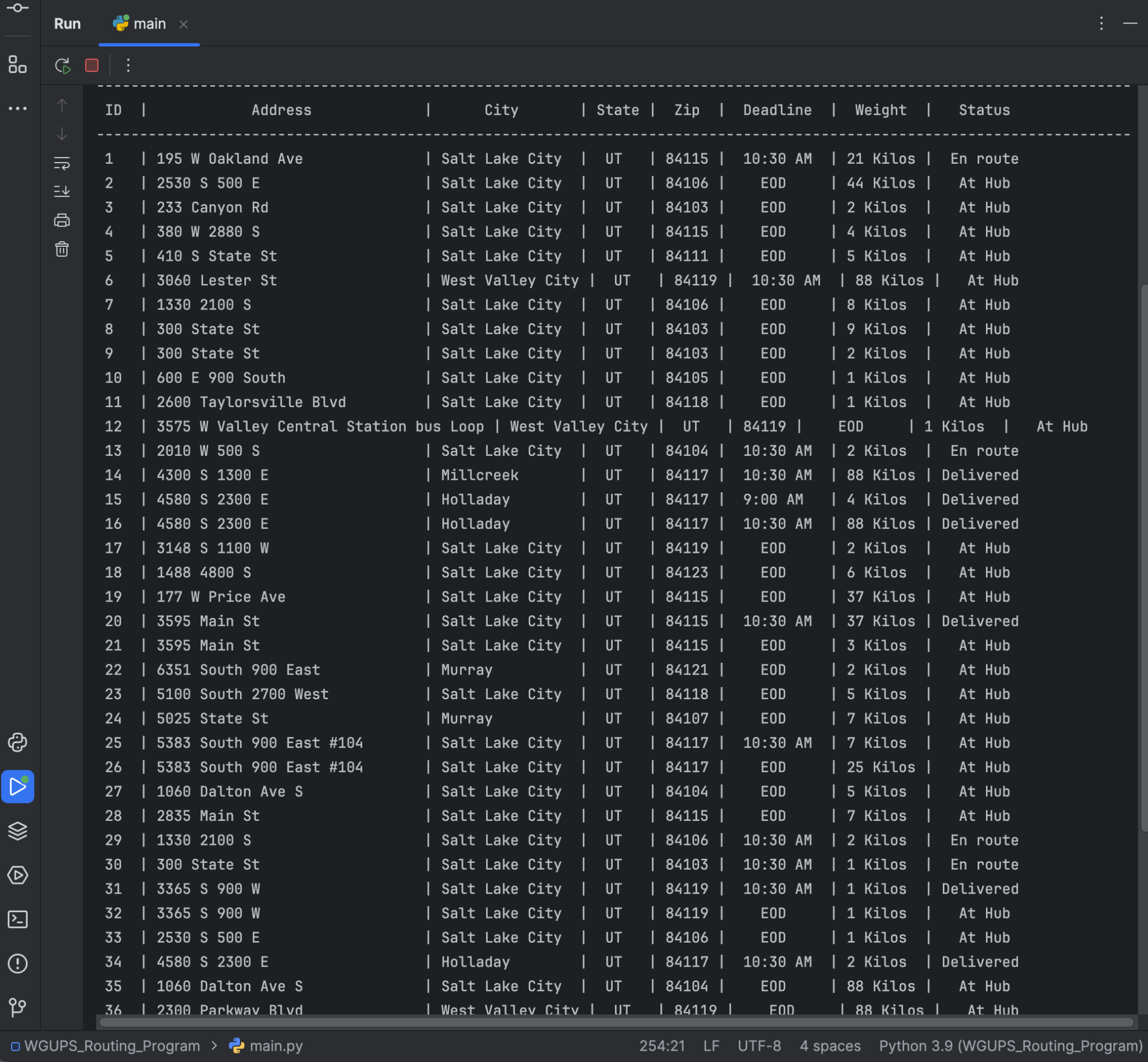




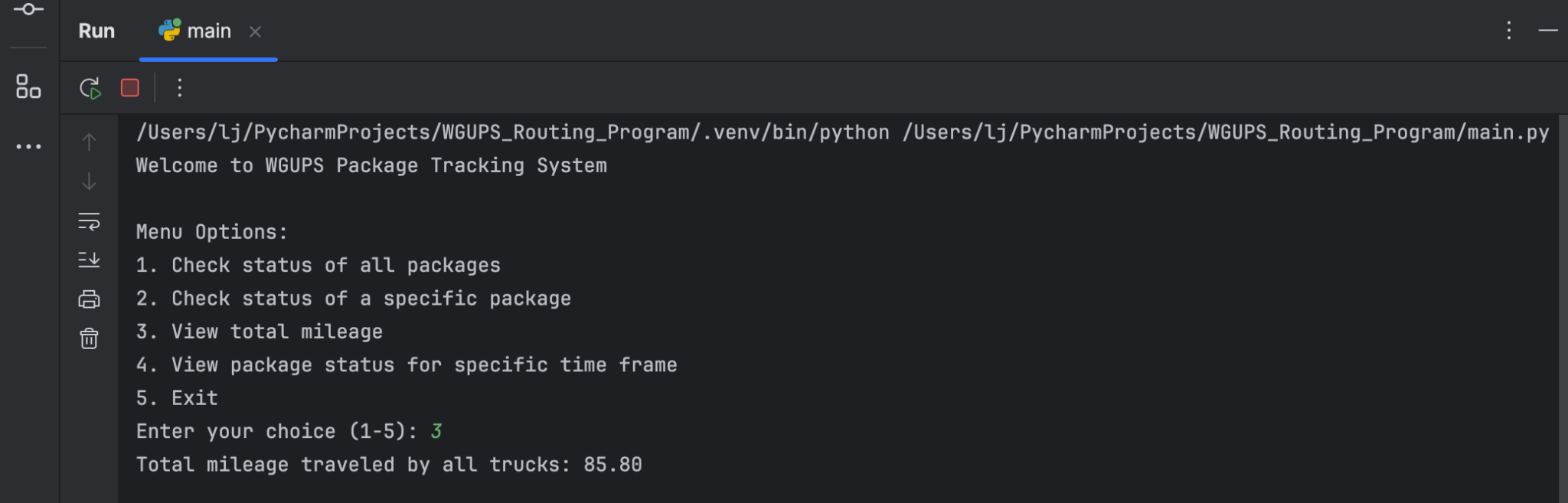
# D. Interface

Interface screenshot goes here

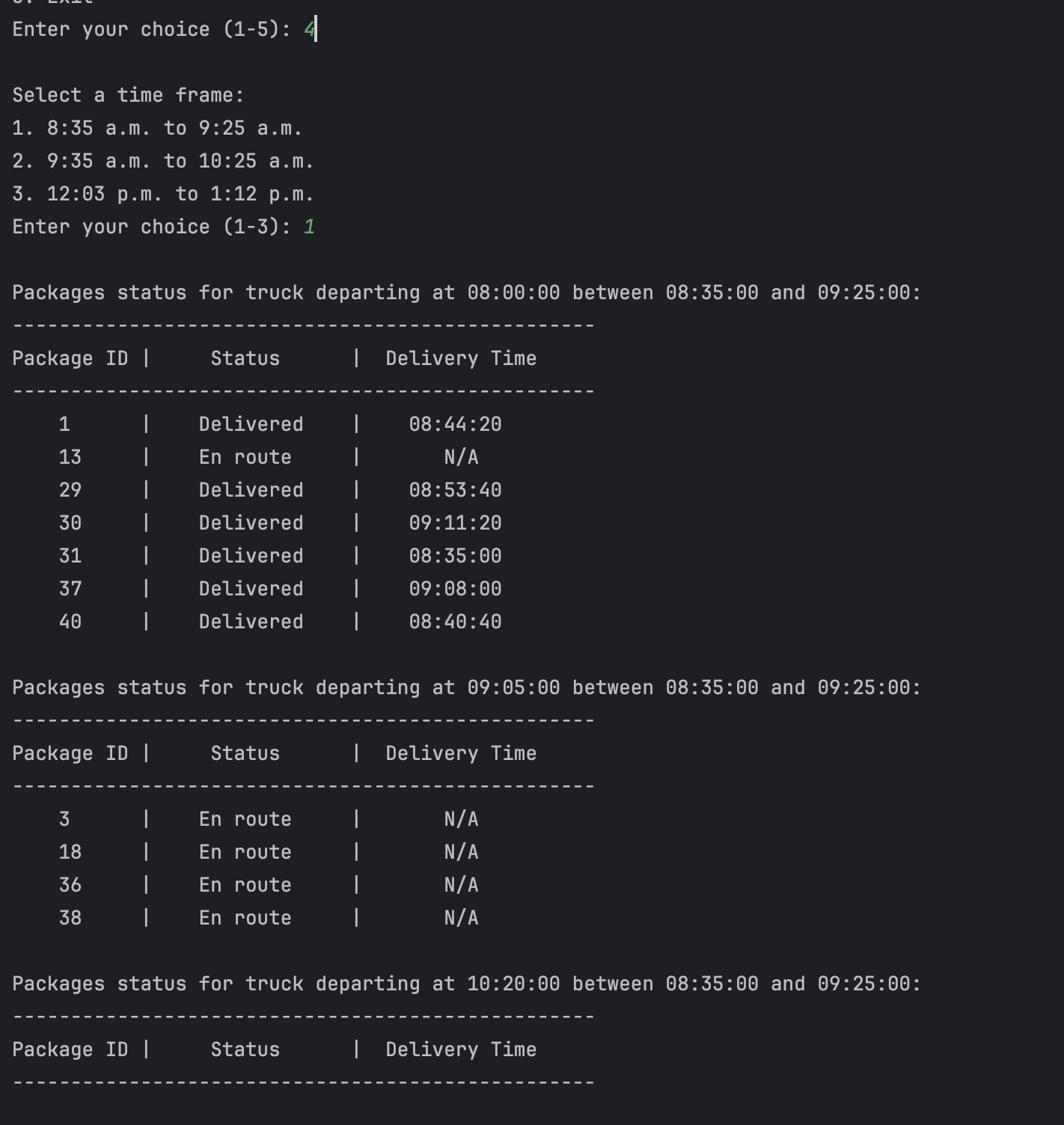




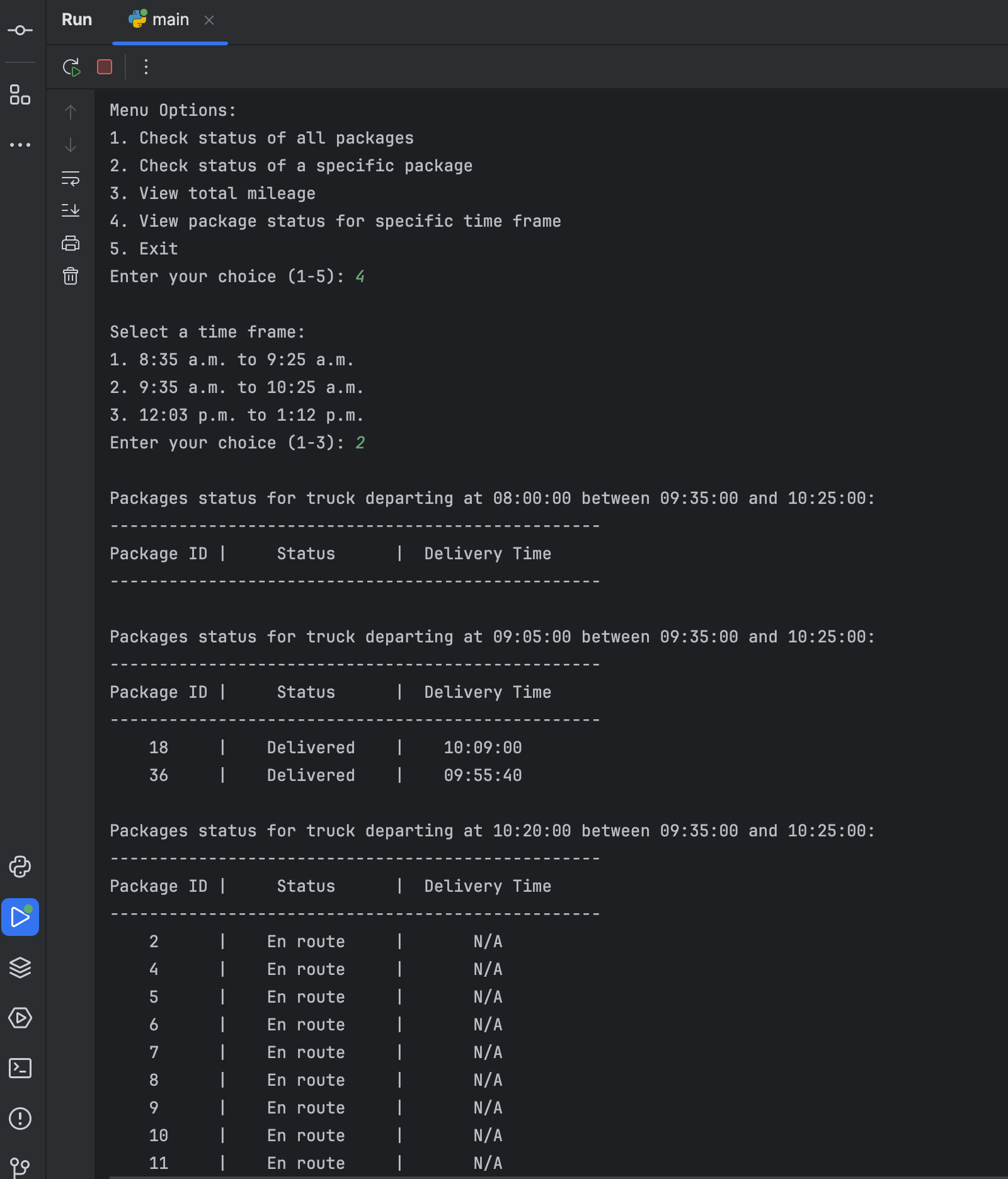
# 

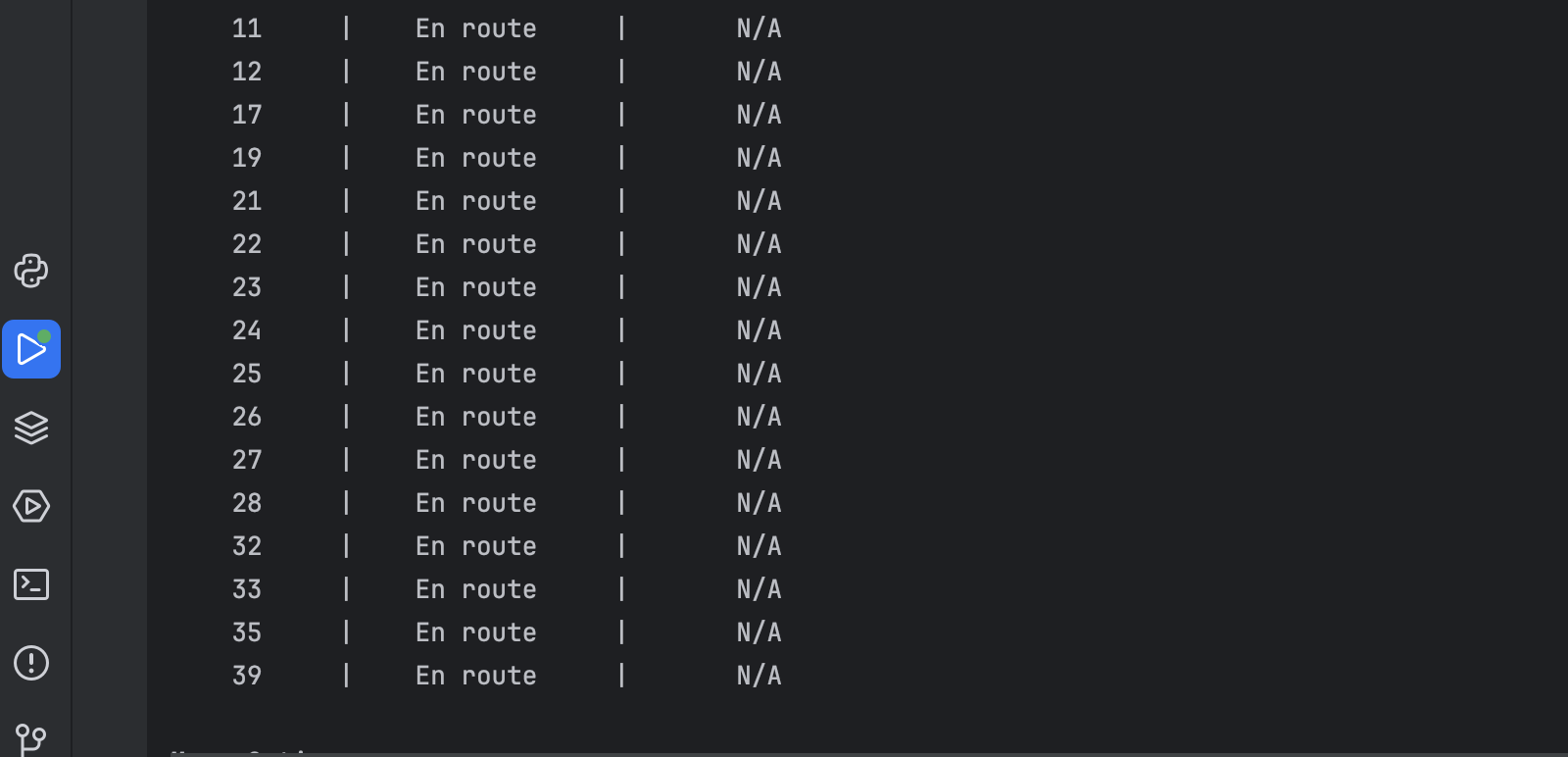


# D1. First Status Check

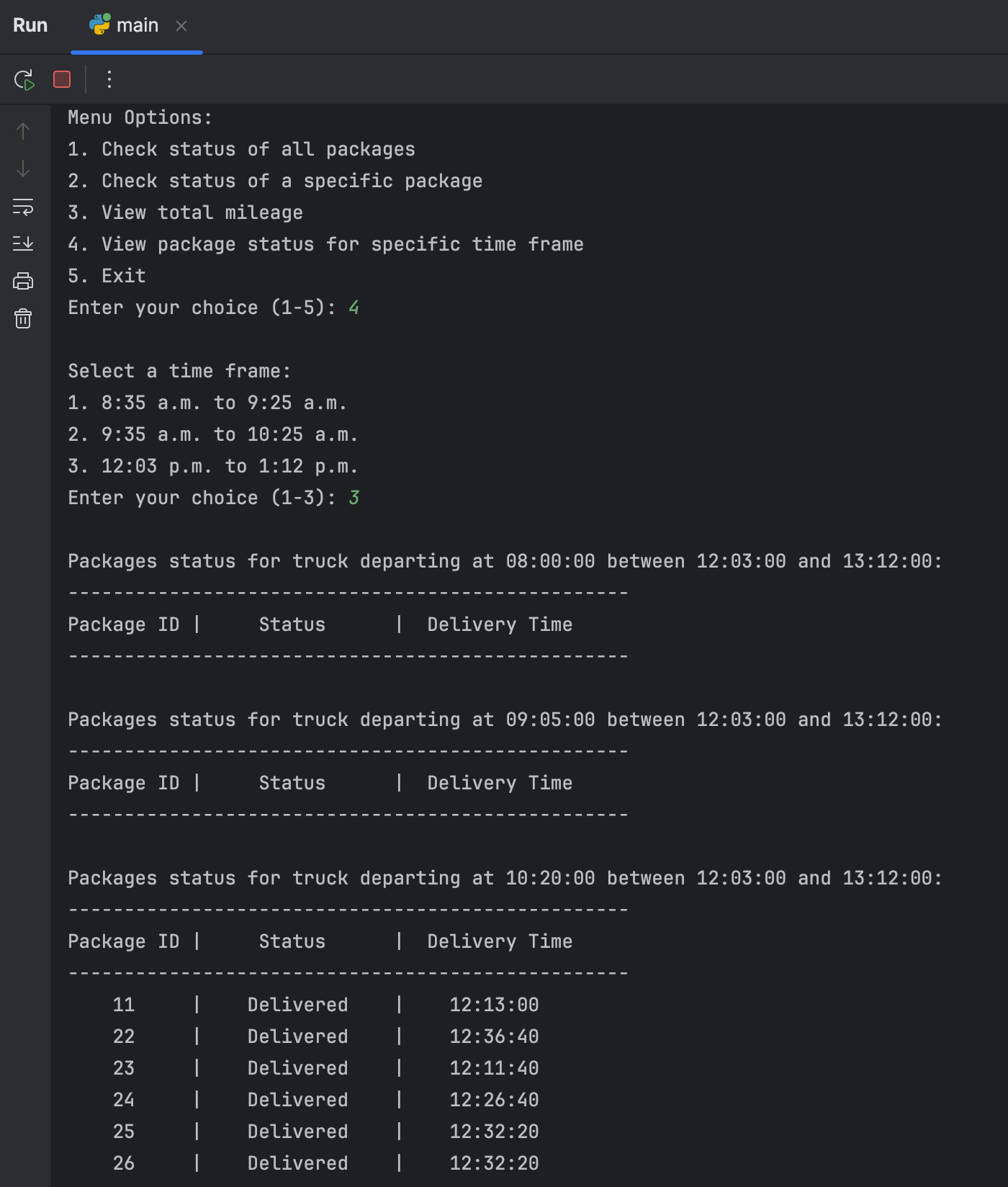


# D2. Second Status Check



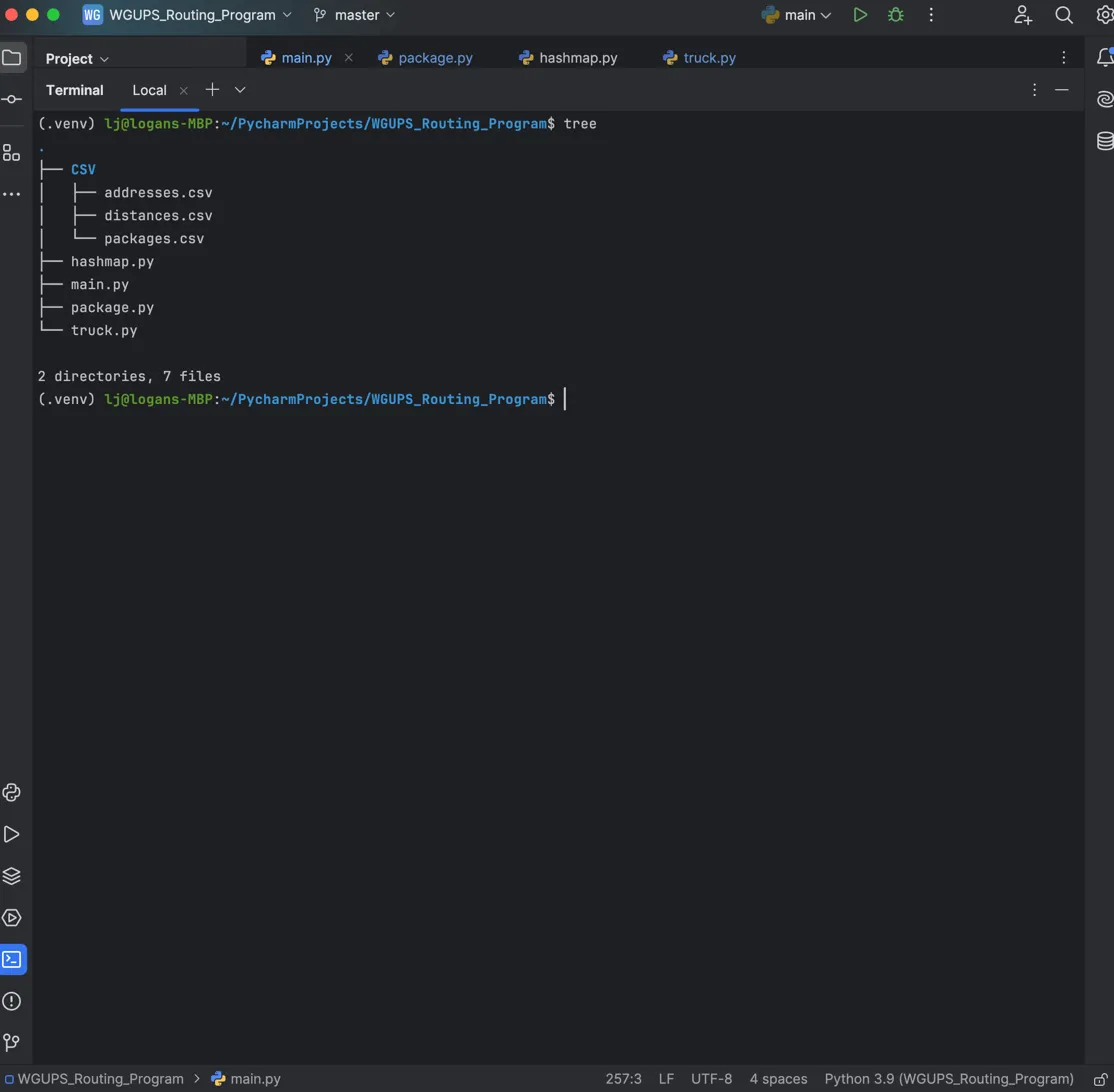


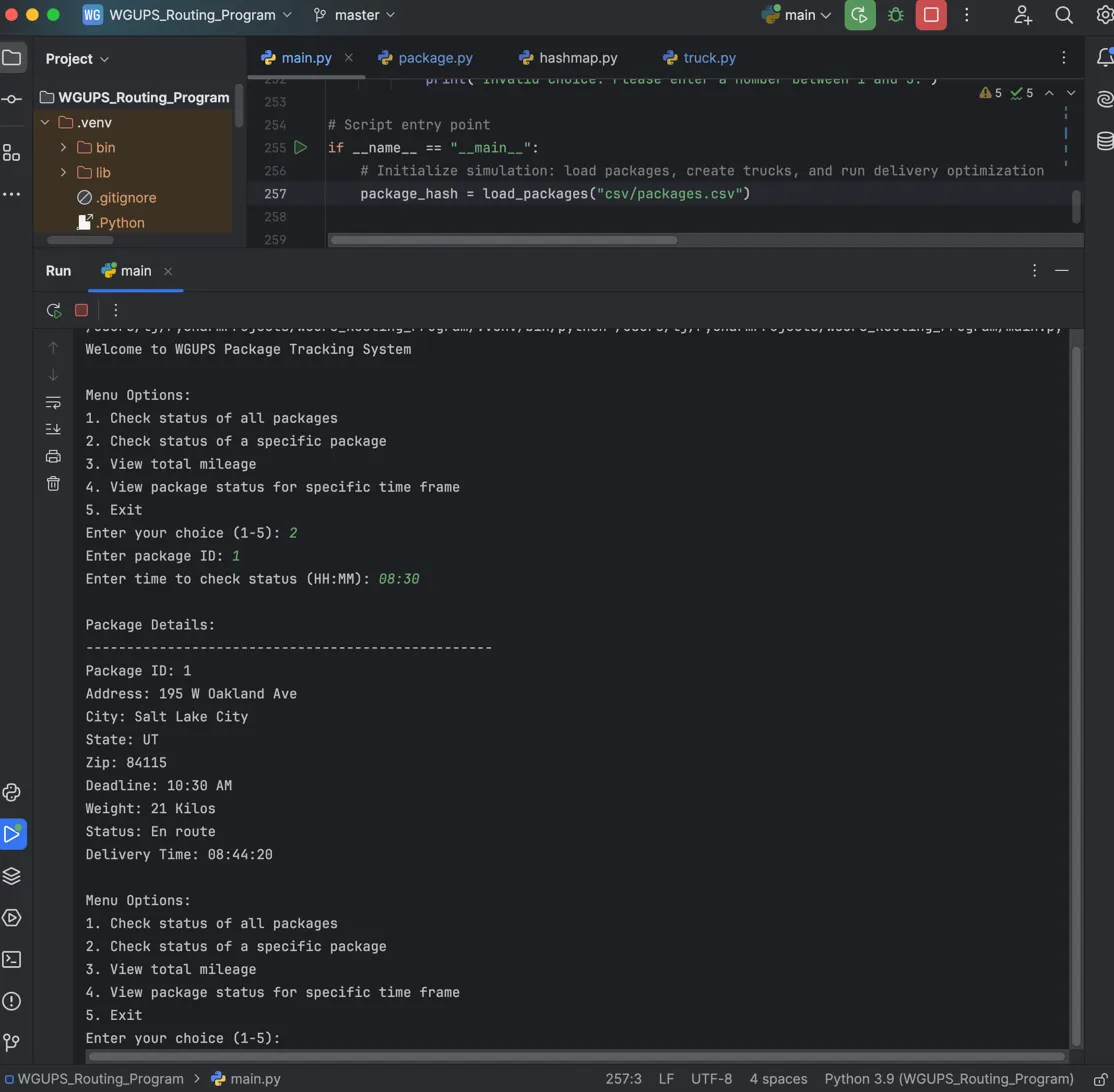
# D3. Third Status Check

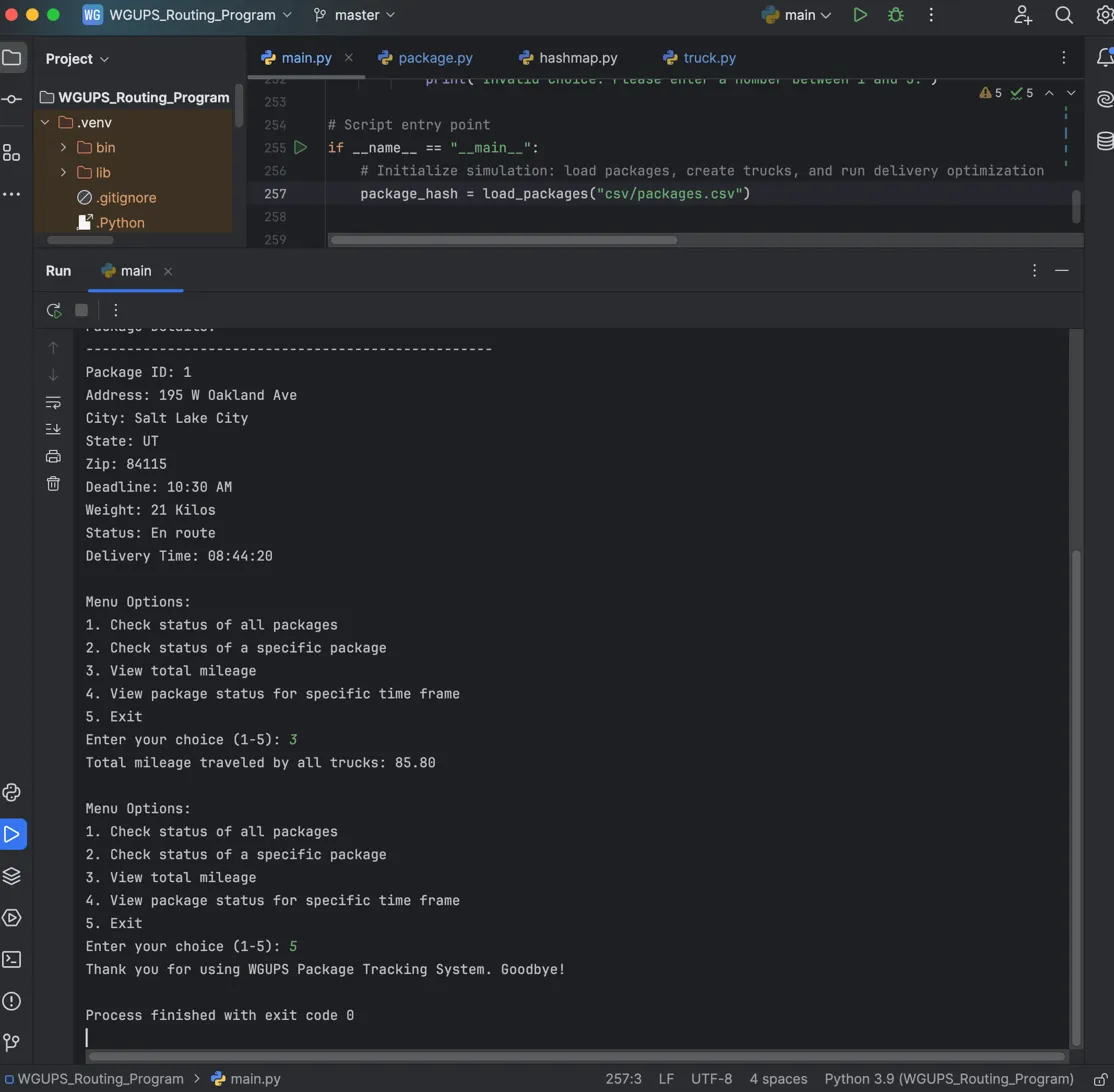


# E. Screenshot of Code Execution









# F1. Strengths of the Chosen Algorithm

The Nearest Neighbor algorithm offers two key strengths that make it particularly effective for the WGUPS scenario. First, it is **computationally simple and efficient**. The algorithm’s simplicity allows for quick development and debugging, which is advantageous when dealing with time-sensitive operations, such as package deliveries, where quick decisions need to be made. It provides a heuristic solution that, while not always optimal, is sufficiently effective for problems where time is a constraint (Cormen et al., 2009).

Second, the algorithm strikes a balance between **efficiency and practicality** in routing. Nearest Neighbor provides a near-optimal route by minimizing the distance traveled at each step, which is crucial for scenarios like package delivery where total mileage is a limiting factor (under 140 miles). Although it may not always yield the absolute best route, it delivers results that are close to optimal with significantly lower computational complexity compared to more advanced algorithms like Genetic Algorithms (Skiena, 2020). This makes it a practical choice for real-world applications where both time and resource constraints are critical.

# F2. Verification of Algorithm

The Nearest Neighbor algorithm used in the WGUPS package delivery system meets all the scenario’s requirements. It efficiently calculates delivery routes that account for each package's destination while considering both time constraints and delivery deadlines. This algorithm ensures that all packages are delivered within the 140-mile total distance constraint while providing real-time updates on package statuses (delivered, en route, or at the hub). Moreover, the Nearest Neighbor algorithm allows for real-time status checks at specific time intervals, ensuring the supervisor can monitor deliveries at any given moment, fulfilling the core needs of the WGUPS system (Sedgewick & Wayne, 2011).

# F3. Other Possible Algorithms

Two other algorithms that could also meet the requirements of the WGUPS scenario are:

1. **Genetic Algorithm**: This algorithm simulates an evolutionary process by generating a population of potential solutions (routes) and iteratively combining them to create better solutions over time. Through a combination of mutation and selection, it converges on an optimal or near-optimal solution (Skiena, 2020). While more computationally intensive than Nearest Neighbor, it has the potential to yield better routes by exploring a larger solution space. This flexibility is particularly useful in complex routing problems with multiple constraints, but the downside is its longer computation time and the need for more tuning of parameters.
2. **Simulated Annealing**: Inspired by the annealing process in metallurgy, Simulated Annealing randomly explores different routes, occasionally accepting worse solutions to avoid local optima, thus improving its chances of finding a global optimum (Cormen et al., 2009). Unlike Nearest Neighbor, which always chooses the closest location, Simulated Annealing allows for more exploration and can sometimes find better solutions in highly complex delivery scenarios. However, it requires careful parameter tuning and can be slower than Nearest Neighbor for simpler problems.

# F3a. Algorithm Differences

The **Genetic Algorithm** and **Simulated Annealing** differ from Nearest Neighbor in several ways. Nearest Neighbor selects the closest next location at each step, resulting in a myopic approach that is efficient but sometimes suboptimal. In contrast, Genetic Algorithms work by evolving multiple potential routes and improving them over generations. This allows Genetic Algorithms to explore more possible solutions but at the cost of increased computational complexity. Simulated Annealing differs by accepting worse solutions temporarily to escape local optima, providing greater flexibility in complex routing problems but also increasing the algorithm’s complexity and computational cost (Sedgewick & Wayne, 2011).

# G. Different Approach

If I were to approach this project again, I would consider using a **hybrid algorithm** that combines elements of both Nearest Neighbor and Simulated Annealing. The Nearest Neighbor could provide an initial quick solution, and then Simulated Annealing could refine this route by exploring other possibilities. Additionally, incorporating **real-time traffic data** into the algorithm would allow dynamic adjustments to routes based on real-world conditions, improving both the efficiency and accuracy of the deliveries. Another modification would involve a **package prioritization system**, which could ensure that high-priority packages are delivered first based on delivery deadlines and special instructions. These changes would enhance the algorithm's adaptability and performance in more complex real-world scenarios (Skiena, 2020).

# H. Verification of Data Structure

The HashTable used in this project meets all the requirements of the scenario. It provides fast O(1) average time complexity for lookups, insertions, and updates, which is essential for a real-time delivery system where quick access to package data is necessary. Using the package ID as the key ensures efficient and direct retrieval of package information such as address, delivery status, and deadline. This structure is particularly well-suited for the dynamic nature of the WGUPS system, where packages are constantly being updated as they move from the hub to the delivery destination (Cormen et al., 2009).

# H1. Other Data Structures

Two other data structures that could meet the same requirements for the WGUPS package delivery system are:

1. **Binary Search Tree (BST)**: A Binary Search Tree is a hierarchical data structure in which each node has a left and right child. The left child contains values smaller than the parent node, and the right child contains values larger than the parent node. The main advantage of a BST is that it allows for efficient searching, insertion, and deletion operations, typically in O(log n) time, assuming the tree is balanced. However, **unbalanced trees** can degrade performance to O(n), which makes this data structure less reliable than a hash table for real-time queries where time efficiency is critical (Cormen et al., 2009).
2. **Array-based List**: An array-based list stores elements in contiguous memory locations and provides fast random access using indices. This means that looking up an element by its index can be done in O(1) time. However, for **searching for a specific element** (like a package by its ID), an array would require O(n) time since each element must be checked sequentially. Inserting and deleting elements in an array also requires shifting elements, making these operations O(n) as well. For a package delivery system with many real-time updates and lookups, this would not be efficient compared to the hash table’s O(1) average lookup and update time (Sedgewick & Wayne, 2011).

# H1a. Data Structure Differences

The **Binary Search Tree (BST)** differs from the HashTable by providing O(log n) time complexity for balanced trees during search, insertions, and deletions. However, BSTs can become unbalanced, leading to worst-case O(n) time complexity. In contrast, the HashTable provides O(1) time complexity on average for these operations. An **Array-based List**, while simple and fast for sequential access, would require O(n) time for searching, making it less efficient for managing the large volume of package data in real-time queries. The HashTable’s ability to maintain quick access and updates makes it the better choice for this scenario (Sedgewick & Wayne, 2011).

# I. Sources

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). **Introduction to Algorithms** (3rd ed.). MIT Press.

Skiena, S. S. (2020). **The Algorithm Design Manual** (3rd ed.). Springer.

Sedgewick, R., & Wayne, K. (2011). **Algorithms** (4th ed.). Addison-Wesley Professional.

# J. Professional Communication

Thanks for reading!