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)

Trayvonious Pendleton

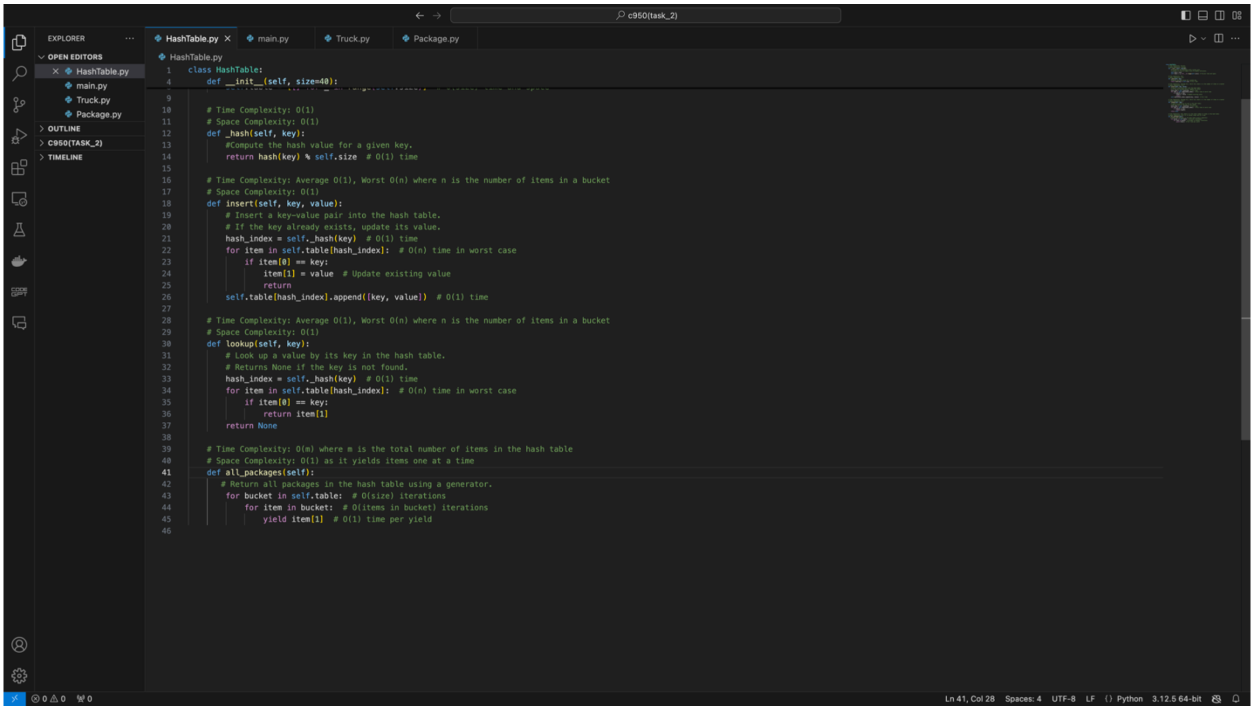
ID #011205284

WGU Email: tpend32@wgu.edu

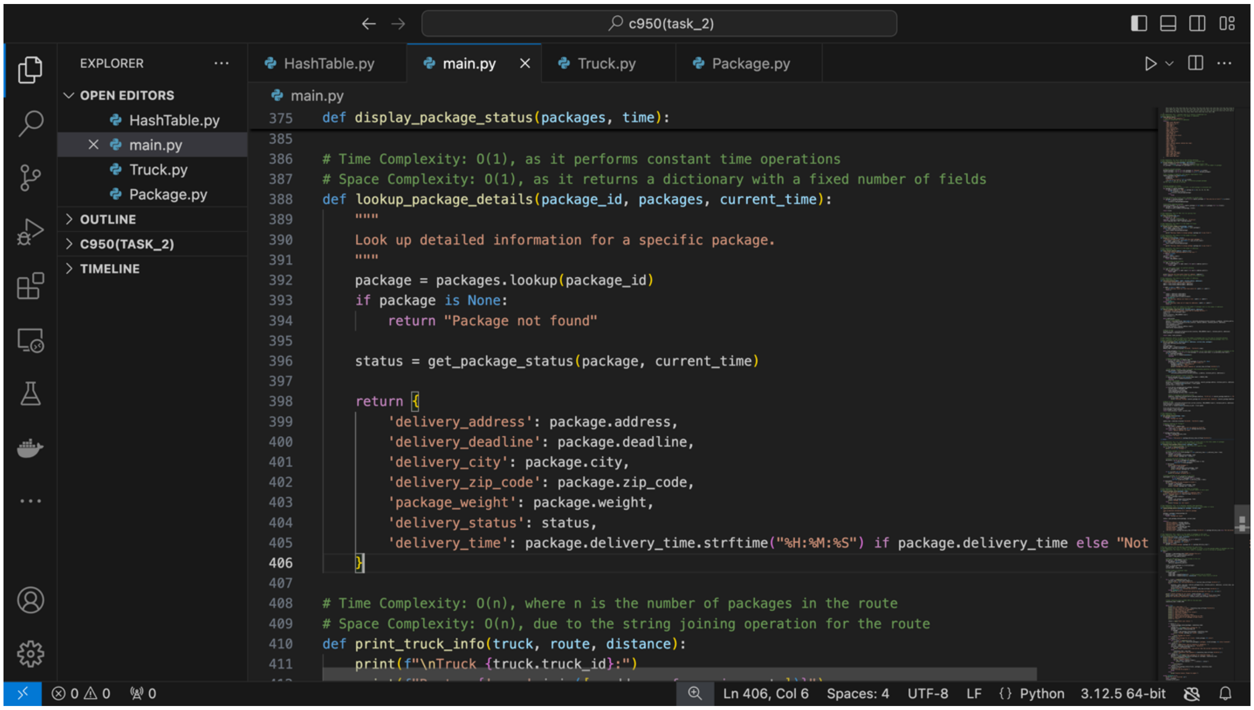
9/24/2024

C950 Data Structures and Algorithms II

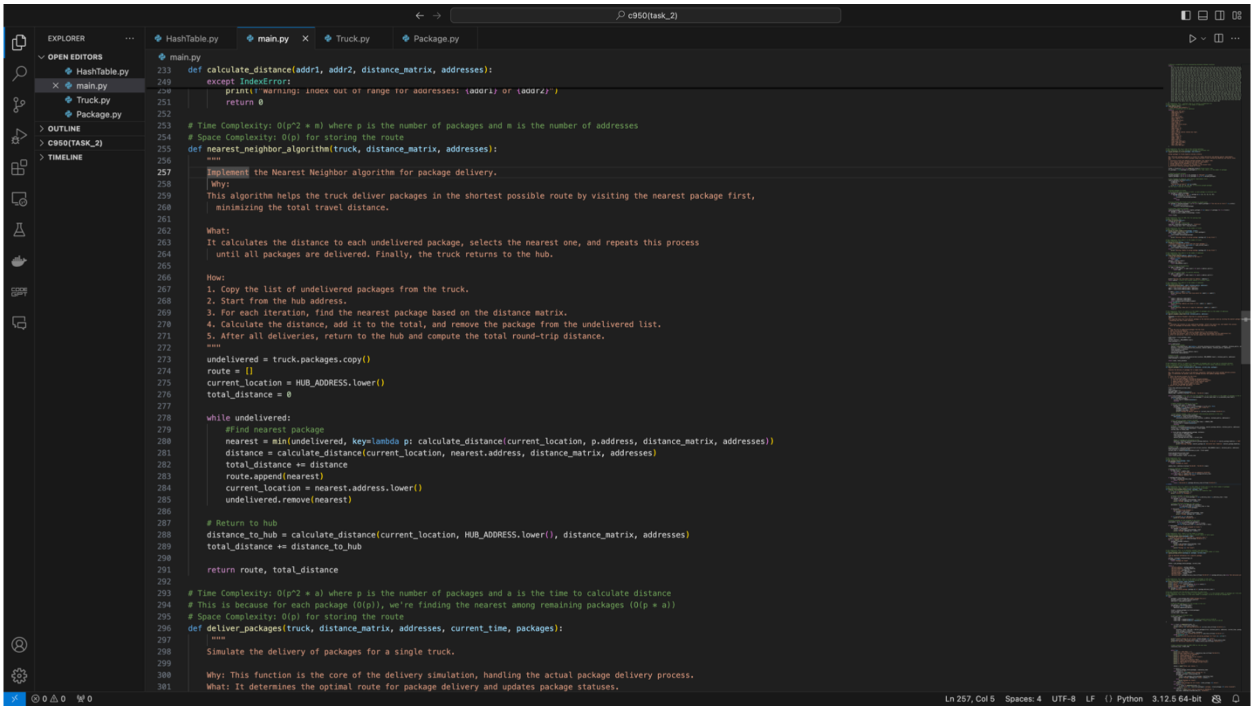
A. Hash Table


c950(task_2)
EXPLORER
OPEN EDITORS
HashTable.py
main.py
Truck.py
Package.py
OUTLINE
C950(TASK_2)
TIMELINE
HashTable.py
class HashTable:
def __init__(self, size=40):
# Time Complexity: O(1)
# Space Complexity: O(1)
def _hash(self, key):
#Compute the hash value for a given key.
return hash(key) % self.size # O(1) time
# Time Complexity: Average O(1), Worst O(n) where n is the number of items in a bucket
# Space Complexity: O(1)
def insert(self, key, value):
# Insert a key-value pair into the hash table.
# If the key already exists, update its value.
hash_index = self._hash(key) # O(1) time
for item in self.table[hash_index]: # O(n) time in worst case
if item[0] == key:
item[1] = value # Update existing value
return
self.table[hash_index].append([key, value]) # O(1) time
# Time Complexity: Average O(1), Worst O(n) where n is the number of items in a bucket
# Space Complexity: O(1)
def lookup(self, key):
# Look up a value by its key in the hash table.
# Returns None if the key is not found.
hash_index = self._hash(key) # O(1) time
for item in self.table[hash_index]: # O(n) time in worst case
if item[0] == key:
return item[1]
return None
# Time Complexity: O(m) where m is the total number of items in the hash table
# Space Complexity: O(1) as it yields items one at a time
def all_packages(self):
# Return all packages in the hash table using a generator.
for bucket in self.table: # O(size) iterations
for item in bucket: # O(items in bucket) iterations
yield item[1] # O(1) time per yield
Ln 41, Col 28 Spaces: 4 UTF-8 LF Python 3.12.5 64-bit

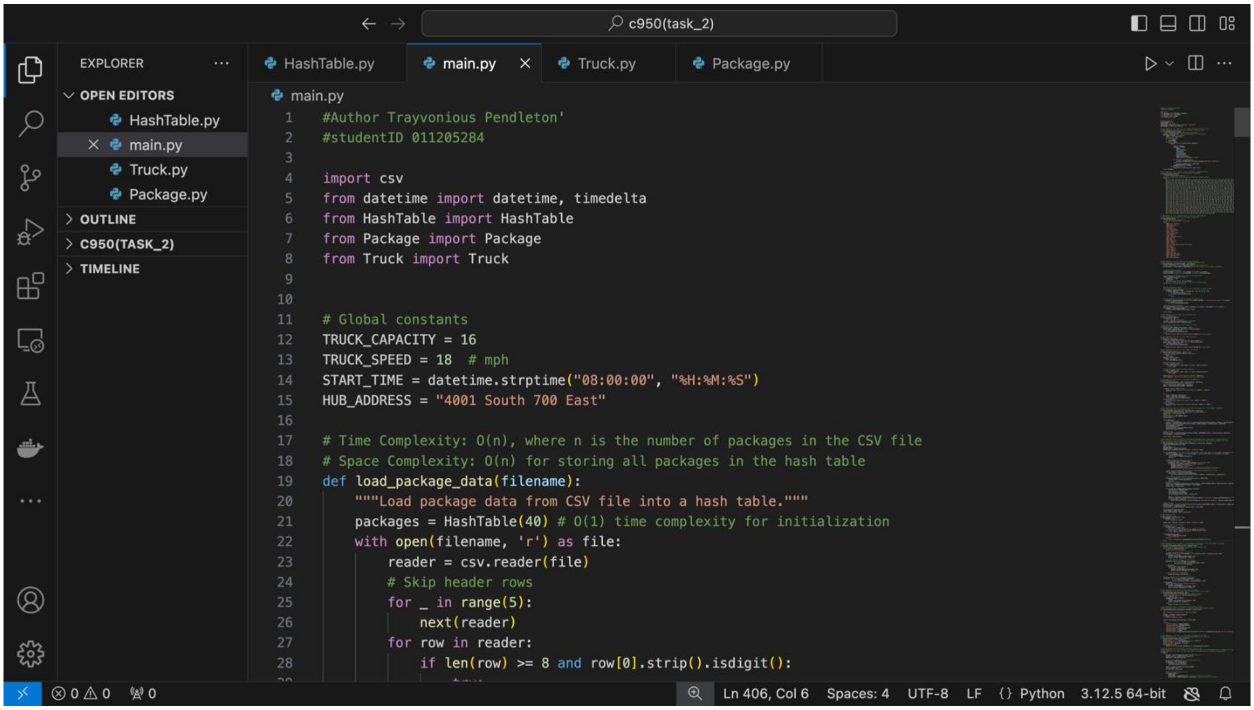
B. Look-Up Functions



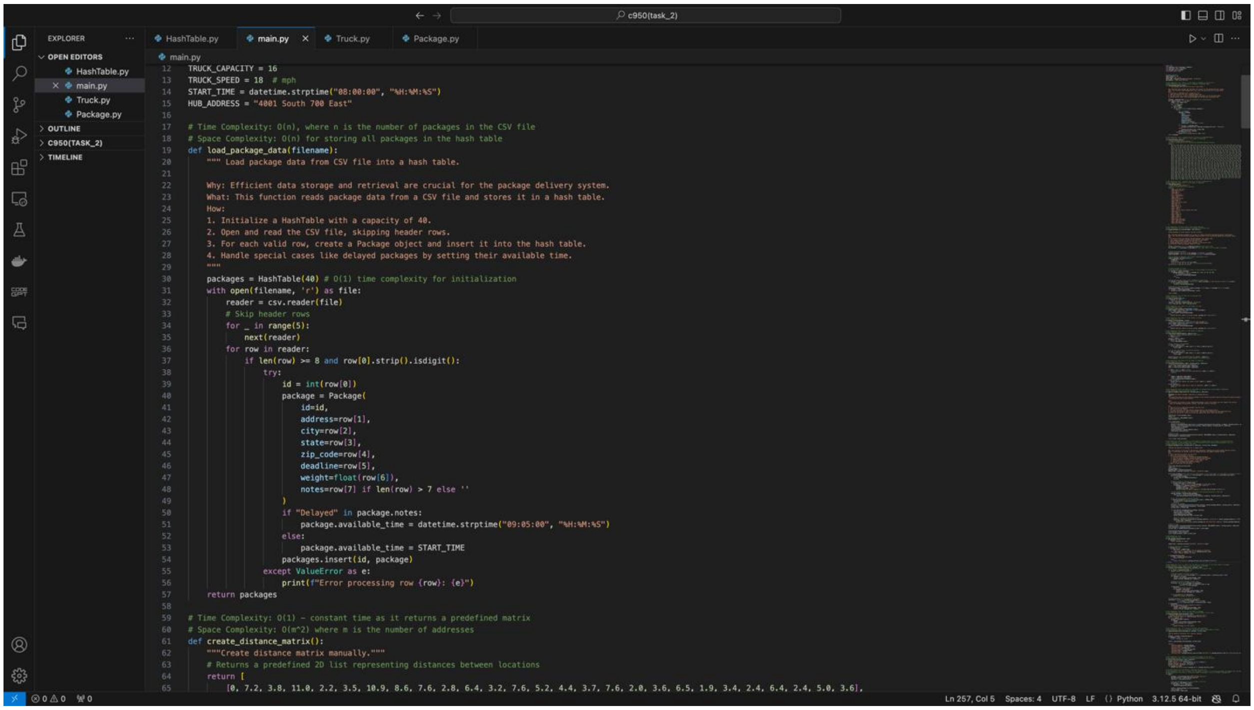
C. Original Code



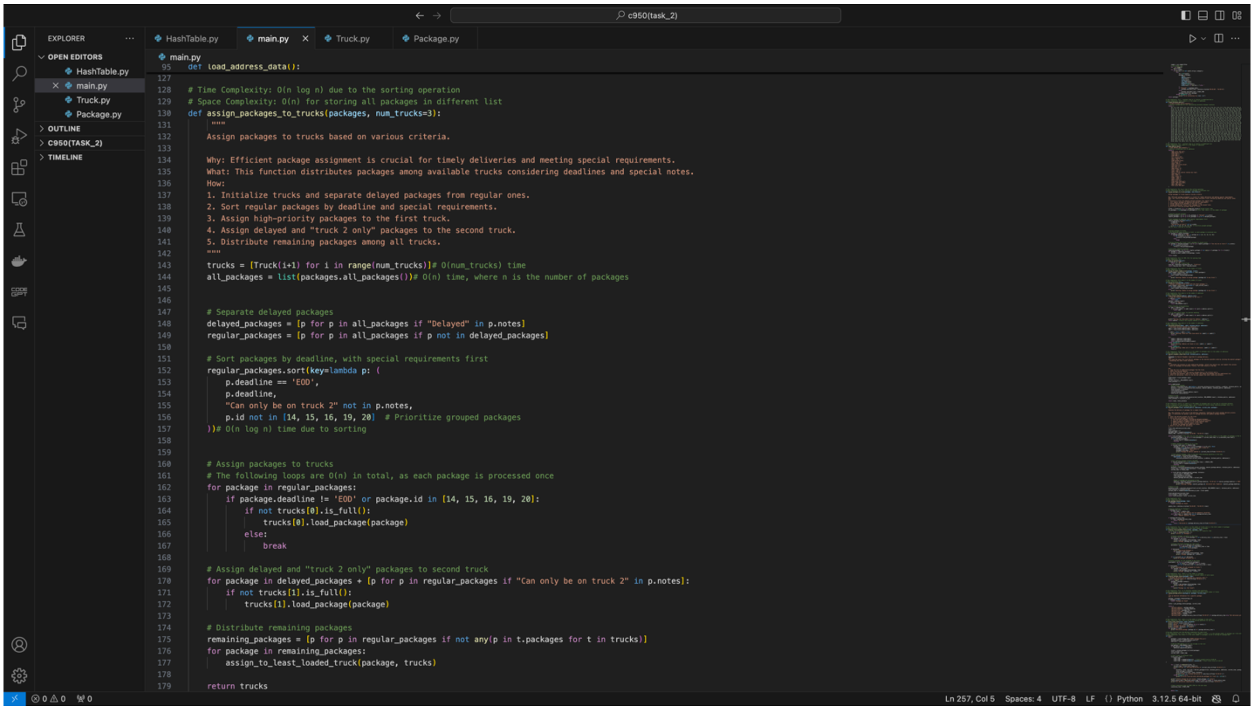
C1. Identification Information



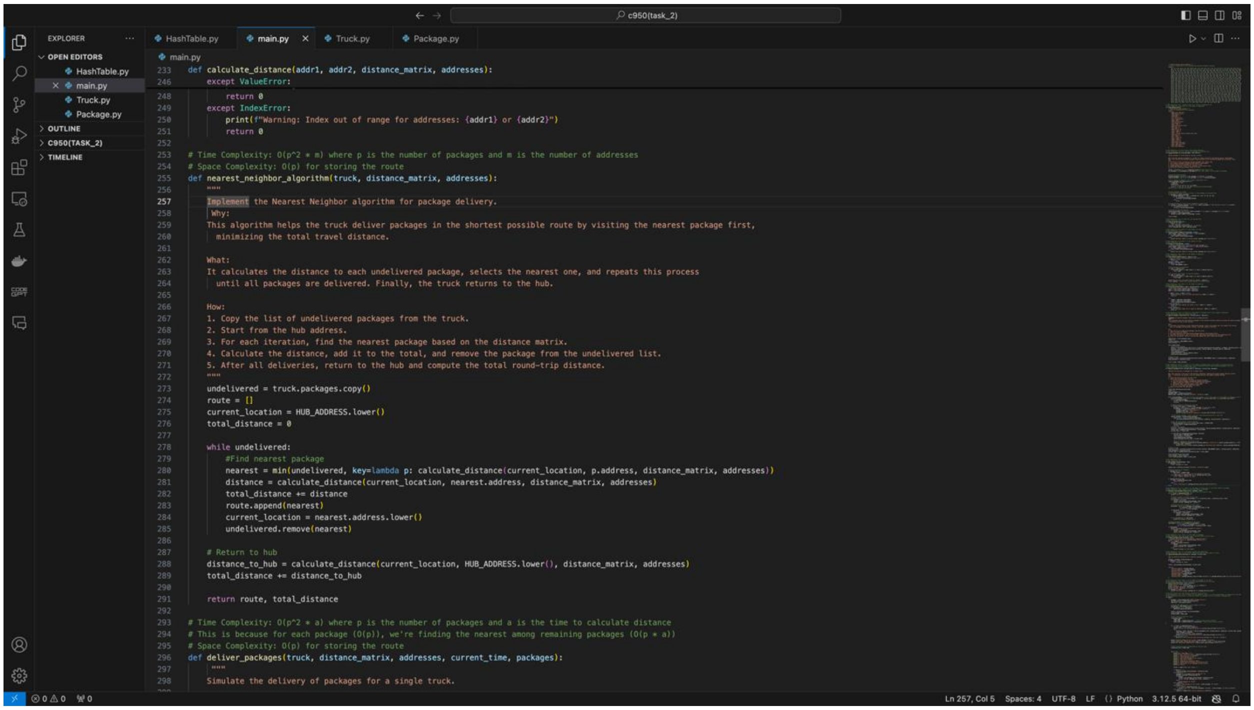
C2. Process and Flow Comments



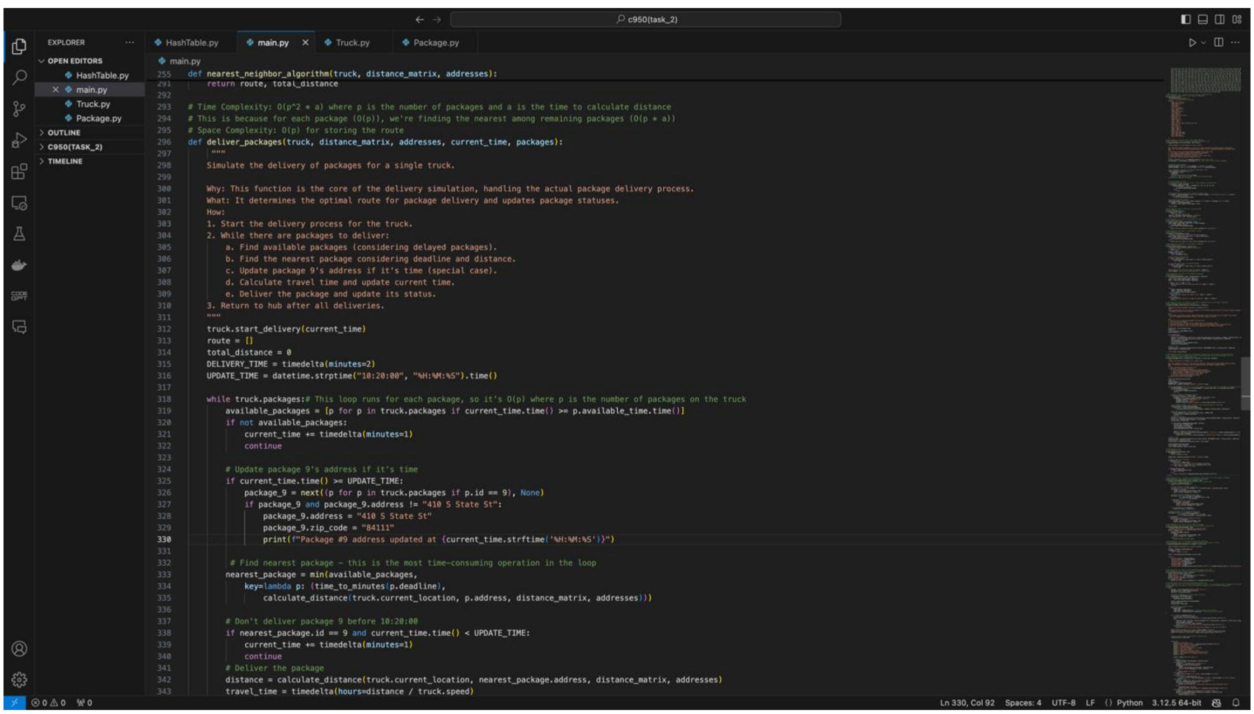
```python
TRUCK_CAPACITY = 16
TRUCK_SPEED = 18  # mph
START_TIME = datetime.strptime("08:00:00", "%H:%M:%S")
HUB_ADDRESS = "4001 South 700 East"

# Time Complexity: O(n), where n is the number of packages in the CSV file
# Space Complexity: O(n) for storing all packages in the hash table
def load_package_data(filename):
    """ Load package data from CSV file into a hash table.

    Why: Efficient data storage and retrieval are crucial for the package delivery system.
    What: This function reads package data from a CSV file and stores it in a hash table.
    How:
    1. Initialize a HashTable with a capacity of 40.
    2. Open and read the CSV file, skipping header rows.
    3. For each valid row, create a Package object and insert it into the hash table.
    4. Handle special cases like delayed packages by setting their available time.
    """
    packages = HashTable(40) # O(1) time complexity for initialization
    with open(filename, 'r') as file:
        reader = csv.reader(file)
        # Skip header rows
        for _ in range(5):
            next(reader)
        for row in reader:
            if len(row) >= 8 and row[0].strip().isdigit():
                try:
                    id = int(row[0])
                    package = Package(
                        id=id,
                        address=row[1],
                        city=row[2],
                        state=row[3],
                        zip_code=row[4],
                        deadline=row[5],
                        weight=float(row[6]),
                        notes=row[7] if len(row) > 7 else ''
                    )
                    if "Delayed" in package.notes:
                        package.available_time = datetime.strptime("09:05:00", "%H:%M:%S")
                    else:
                        package.available_time = START_TIME
                    packages.insert(id, package)
                except ValueError as e:
                    print(f"Error processing row {row}: {e}")
    return packages

# Time Complexity: O(1) - constant time as it returns a predefined matrix
# Space Complexity: O(m^2) where m is the number of addresses
def create_distance_matrix():
    """Create distance matrix manually."""
    # Returns a predefined 2D list representing distances between locations
    return [
        [0, 7.2, 3.8, 11.0, 2.2, 3.5, 10.9, 8.6, 7.6, 2.8, 6.4, 3.2, 7.6, 5.2, 4.4, 3.7, 7.6, 2.0, 3.6, 6.5, 1.9, 3.4, 2.4, 6.4, 2.4, 5.0, 3.6],
```



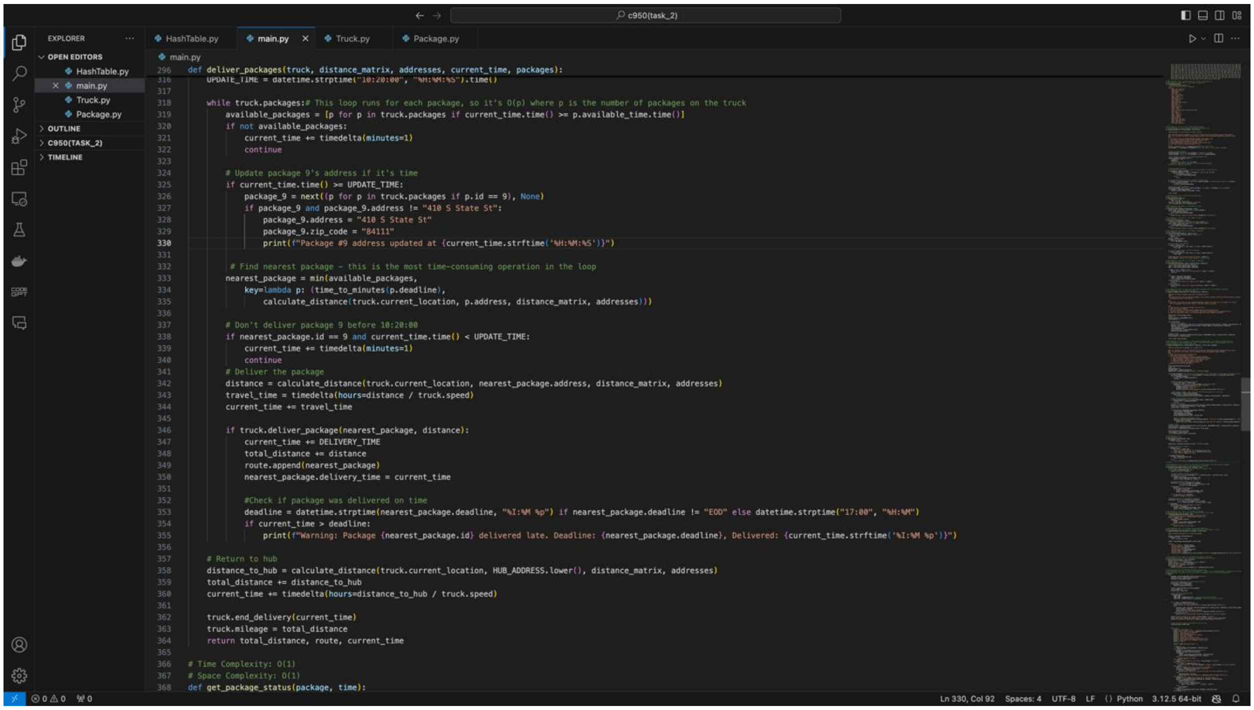
```python
def load_address_data():

# Time Complexity: O(n log n) due to the sorting operation
# Space Complexity: O(n) for storing all packages in different list
def assign_packages_to_trucks(packages, num_trucks=3):
    """
    Assign packages to trucks based on various criteria.

    Why: Efficient package assignment is crucial for timely deliveries and meeting special requirements.
    What: This function distributes packages among available trucks considering deadlines and special notes.
    How:
    1. Initialize trucks and separate delayed packages from regular ones.
    2. Sort regular packages by deadline and special requirements.
    3. Assign high-priority packages to the first truck.
    4. Assign delayed and "truck 2 only" packages to the second truck.
    5. Distribute remaining packages among all trucks.
    """
    trucks = [Truck(i+1) for i in range(num_trucks)]# O(num_trucks) time
    all_packages = list(packages.all_packages())# O(n) time, where n is the number of packages


    # Separate delayed packages
    delayed_packages = [p for p in all_packages if "Delayed" in p.notes]
    regular_packages = [p for p in all_packages if p not in delayed_packages]

    # Sort packages by deadline, with special requirements first
    regular_packages.sort(key=lambda p: (
        p.deadline == 'EOD',
        p.deadline,
        "Can only be on truck 2" not in p.notes,
        p.id not in [14, 15, 16, 19, 20]  # Prioritize grouped packages
    ))# O(n log n) time due to sorting


    # Assign packages to trucks
    # The following loops are O(n) in total, as each package is processed once
    for package in regular_packages:
        if package.deadline != 'EOD' or package.id in [14, 15, 16, 19, 20]:
            if not trucks[0].is_full():
                trucks[0].load_package(package)
            else:
                break

    # Assign delayed and "truck 2 only" packages to second truck
    for package in delayed_packages + [p for p in regular_packages if "Can only be on truck 2" in p.notes]:
        if not trucks[1].is_full():
            trucks[1].load_package(package)

    # Distribute remaining packages
    remaining_packages = [p for p in regular_packages if not any(p in t.packages for t in trucks)]
    for package in remaining_packages:
        assign_to_least_loaded_truck(package, trucks)

    return trucks
```



```python
def calculate_distance(addr1, addr2, distance_matrix, addresses):
    except ValueError:
        return 0
    except IndexError:
        print(f"Warning: Index out of range for addresses: {addr1} or {addr2}")
        return 0

# Time Complexity: O(p^2 * m) where p is the number of packages and m is the number of addresses
# Space Complexity: O(p) for storing the route
def nearest_neighbor_algorithm(truck, distance_matrix, addresses):
    """
    Implement the Nearest Neighbor algorithm for package delivery.
     Why:
    This algorithm helps the truck deliver packages in the shortest possible route by visiting the nearest package first,
      minimizing the total travel distance.

    What:
    It calculates the distance to each undelivered package, selects the nearest one, and repeats this process
      until all packages are delivered. Finally, the truck returns to the hub.

    How:
    1. Copy the list of undelivered packages from the truck.
    2. Start from the hub address.
    3. For each iteration, find the nearest package based on the distance matrix.
    4. Calculate the distance, add it to the total, and remove the package from the undelivered list.
    5. After all deliveries, return to the hub and compute the total round-trip distance.
    """
    undelivered = truck.packages.copy()
    route = []
    current_location = HUB_ADDRESS.lower()
    total_distance = 0

    while undelivered:
        #Find nearest package
        nearest = min(undelivered, key=lambda p: calculate_distance(current_location, p.address, distance_matrix, addresses))
        distance = calculate_distance(current_location, nearest.address, distance_matrix, addresses)
        total_distance += distance
        route.append(nearest)
        current_location = nearest.address.lower()
        undelivered.remove(nearest)

    # Return to hub
    distance_to_hub = calculate_distance(current_location, HUB_ADDRESS.lower(), distance_matrix, addresses)
    total_distance += distance_to_hub

    return route, total_distance

# Time Complexity: O(p^2 * a) where p is the number of packages and a is the time to calculate distance
# This is because for each package (O(p)), we're finding the nearest among remaining packages (O(p * a))
# Space Complexity: O(p) for storing the route
def deliver_packages(truck, distance_matrix, addresses, current_time, packages):
     """
    Simulate the delivery of packages for a single truck.
```

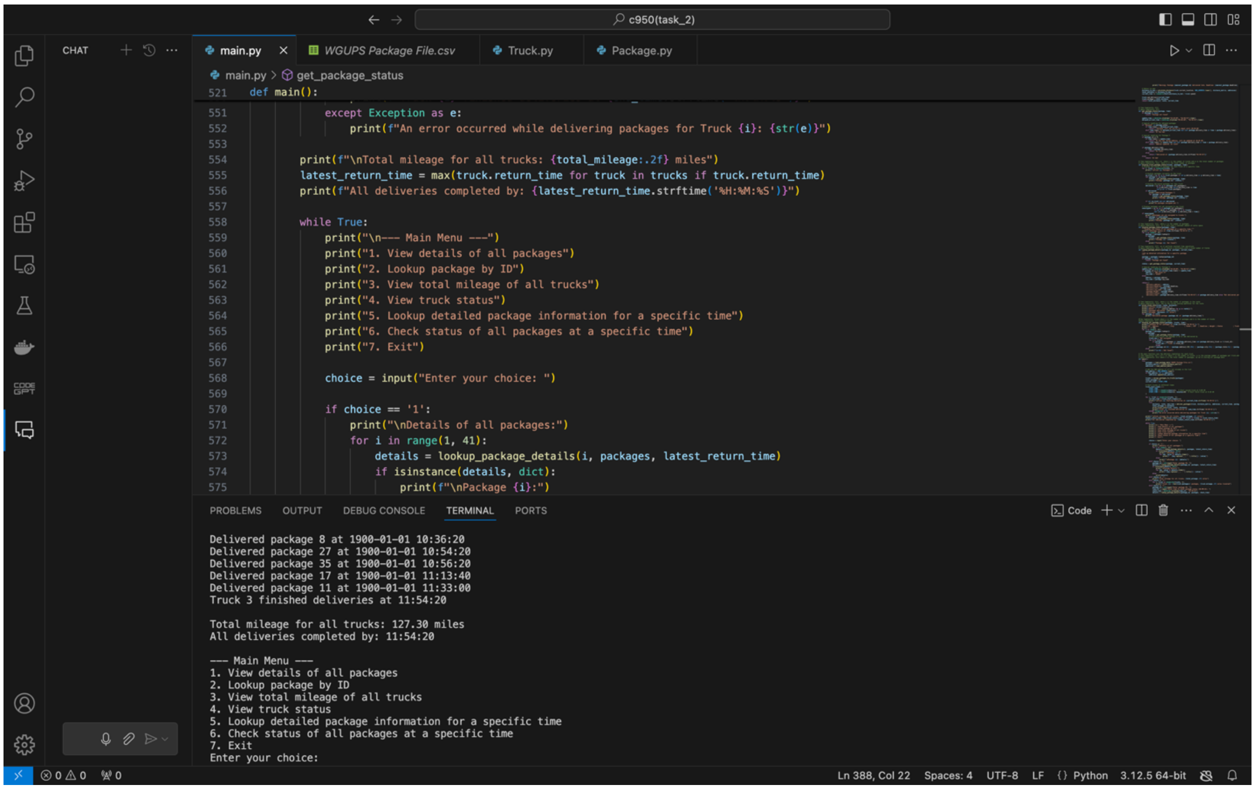


```python
def nearest_neighbor_algorithm(truck, distance_matrix, addresses):
    return route, total_distance

# Time Complexity: O(p^2 * a) where p is the number of packages and a is the time to calculate distance
# This is because for each package (O(p)), we're finding the nearest among remaining packages (O(p * a))
# Space Complexity: O(p) for storing the route
def deliver_packages(truck, distance_matrix, addresses, current_time, packages):
     """
    Simulate the delivery of packages for a single truck.

    Why: This function is the core of the delivery simulation, handling the actual package delivery process.
    What: It determines the optimal route for package delivery and updates package statuses.
    How:
    1. Start the delivery process for the truck.
    2. While there are packages to deliver:
        a. Find available packages (considering delayed packages).
        b. Find the nearest package considering deadline and distance.
        c. Update package 9's address if it's time (special case).
        d. Calculate travel time and update current time.
        e. Deliver the package and update its status.
    3. Return to hub after all deliveries.
    """
    truck.start_delivery(current_time)
    route = []
    total_distance = 0
    DELIVERY_TIME = timedelta(minutes=2)
    UPDATE_TIME = datetime.strptime("10:20:00", "%H:%M:%S").time()

    while truck.packages:# This loop runs for each package, so it's O(p) where p is the number of packages on the truck
        available_packages = [p for p in truck.packages if current_time.time() >= p.available_time.time()]
        if not available_packages:
            current_time += timedelta(minutes=1)
            continue

        # Update package 9's address if it's time
        if current_time.time() >= UPDATE_TIME:
            package_9 = next((p for p in truck.packages if p.id == 9), None)
            if package_9 and package_9.address != "410 S State St":
                package_9.address = "410 S State St"
                package_9.zip_code = "84111"
                print(f"Package #9 address updated at {current_time.strftime('%H:%M:%S')}")

         # Find nearest package - this is the most time-consuming operation in the loop
        nearest_package = min(available_packages,
            key=lambda p: (time_to_minutes(p.deadline),
                calculate_distance(truck.current_location, p.address, distance_matrix, addresses)))

        # Don't deliver package 9 before 10:20:00
        if nearest_package.id == 9 and current_time.time() < UPDATE_TIME:
            current_time += timedelta(minutes=1)
            continue
        # Deliver the package
        distance = calculate_distance(truck.current_location, nearest_package.address, distance_matrix, addresses)
        travel_time = timedelta(hours=distance / truck.speed)
```

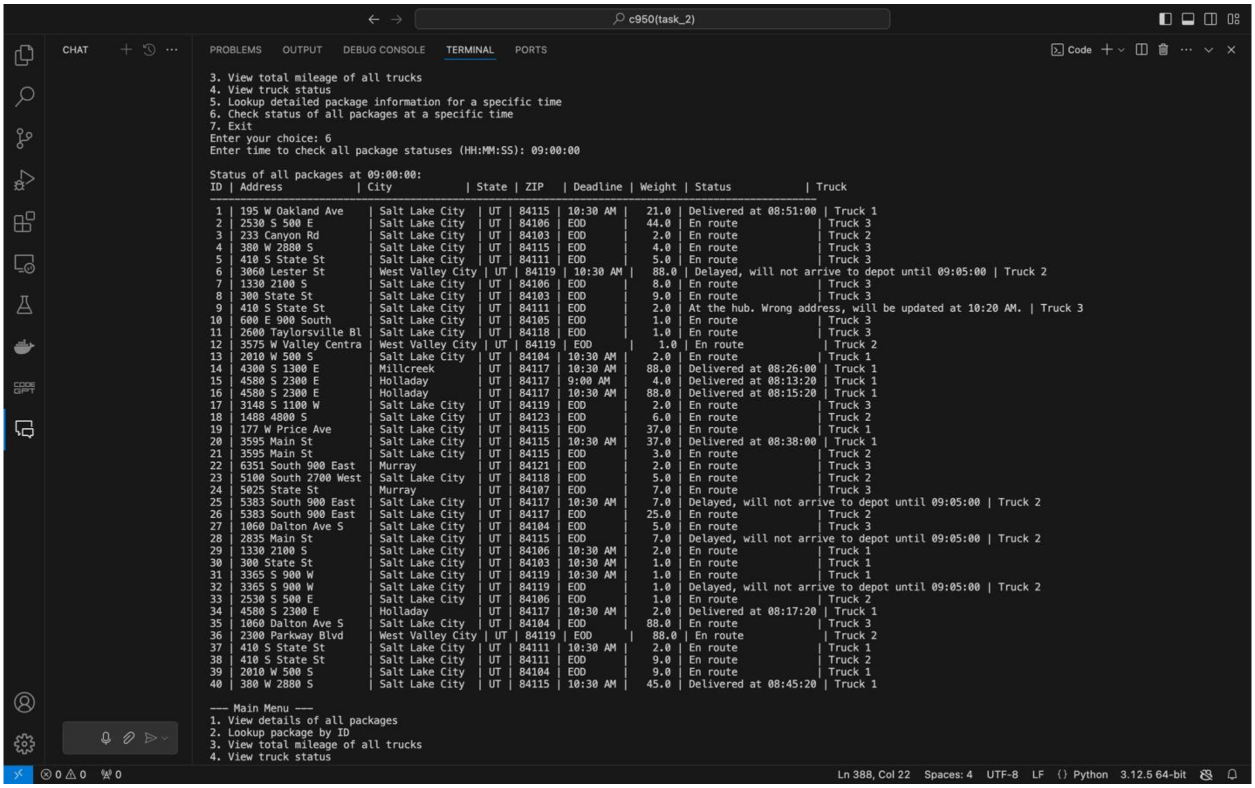


c950(task_2)
HashTable.py
main.py
Truck.py
Package.py
def deliver_packages(truck, distance_matrix, addresses, current_time, packages):
    UPDATE_TIME = datetime.strptime("10:20:00", "%H:%M:%S").time()
    while truck.packages:# This loop runs for each package, so it's O(p) where p is the number of packages on the truck
        available_packages = [p for p in truck.packages if current_time.time() >= p.available_time.time()]
        if not available_packages:
            current_time += timedelta(minutes=1)
            continue
        # Update package 9's address if it's time
        if current_time.time() >= UPDATE_TIME:
            package_9 = next((p for p in truck.packages if p.id == 9), None)
            if package_9 and package_9.address != "410 S State St":
                package_9.address = "410 S State St"
                package_9.zip_code = "84111"
                print(f"Package #9 address updated at {current_time.strftime('%H:%M:%S')}")
        # Find nearest package - this is the most time-consuming operation in the loop
        nearest_package = min(available_packages,
            key=lambda p: (time_to_minutes(p.deadline),
                calculate_distance(truck.current_location, p.address, distance_matrix, addresses)))
        # Don't deliver package 9 before 10:20:00
        if nearest_package.id == 9 and current_time.time() < UPDATE_TIME:
            current_time += timedelta(minutes=1)
            continue
        # Deliver the package
        distance = calculate_distance(truck.current_location, nearest_package.address, distance_matrix, addresses)
        travel_time = timedelta(hours=distance / truck.speed)
        current_time += travel_time
        if truck.deliver_package(nearest_package, distance):
            current_time += DELIVERY_TIME
            total_distance += distance
            route.append(nearest_package)
            nearest_package.delivery_time = current_time
            #Check if package was delivered on time
            deadline = datetime.strptime(nearest_package.deadline, "%I:%M %p") if nearest_package.deadline != "EOD" else datetime.strptime("17:00", "%H:%M")
            if current_time > deadline:
                print(f"Warning: Package {nearest_package.id} delivered late. Deadline: {nearest_package.deadline}, Delivered: {current_time.strftime('%I:%M %p')}")
    # Return to hub
    distance_to_hub = calculate_distance(truck.current_location, HUB_ADDRESS.lower(), distance_matrix, addresses)
    total_distance += distance_to_hub
    current_time += timedelta(hours=distance_to_hub / truck.speed)
    truck.end_delivery(current_time)
    truck.mileage = total_distance
    return total_distance, route, current_time
# Time Complexity: O(1)
# Space Complexity: O(1)
def get_package_status(package, time):

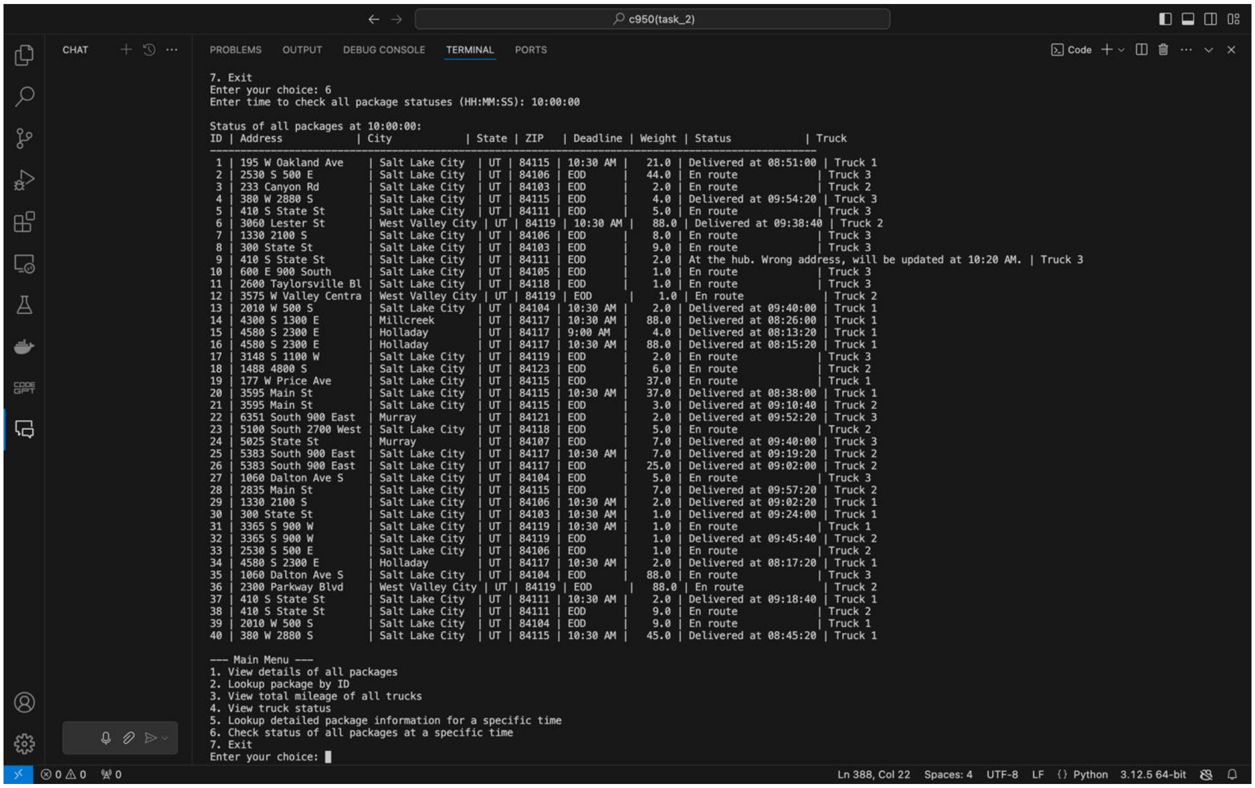
D. Interface


c950(task_2)
CHAT
main.py
WGUPS Package File.csv
Truck.py
Package.py
main.py > get_package_status
def main():
except Exception as e:
print(f"An error occurred while delivering packages for Truck {i}: {str(e)}")
print(f"\nTotal mileage for all trucks: {total_mileage:.2f} miles")
latest_return_time = max(truck.return_time for truck in trucks if truck.return_time)
print(f"All deliveries completed by: {latest_return_time.strftime('%H:%M:%S')}")
while True:
print("\n--- Main Menu ---")
print("1. View details of all packages")
print("2. Lookup package by ID")
print("3. View total mileage of all trucks")
print("4. View truck status")
print("5. Lookup detailed package information for a specific time")
print("6. Check status of all packages at a specific time")
print("7. Exit")
choice = input("Enter your choice: ")
if choice == '1':
print("\nDetails of all packages:")
for i in range(1, 41):
details = lookup_package_details(i, packages, latest_return_time)
if isinstance(details, dict):
print(f"\nPackage {i}:")
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
Code
Delivered package 8 at 1900-01-01 10:36:20
Delivered package 27 at 1900-01-01 10:54:20
Delivered package 35 at 1900-01-01 10:56:20
Delivered package 17 at 1900-01-01 11:13:40
Delivered package 11 at 1900-01-01 11:33:00
Truck 3 finished deliveries at 11:54:20
Total mileage for all trucks: 127.30 miles
All deliveries completed by: 11:54:20
--- Main Menu ---
1. View details of all packages
2. Lookup package by ID
3. View total mileage of all trucks
4. View truck status
5. Lookup detailed package information for a specific time
6. Check status of all packages at a specific time
7. Exit
Enter your choice:
Ln 388, Col 22
Spaces: 4
UTF-8
LF
Python
3.12.5 64-bit

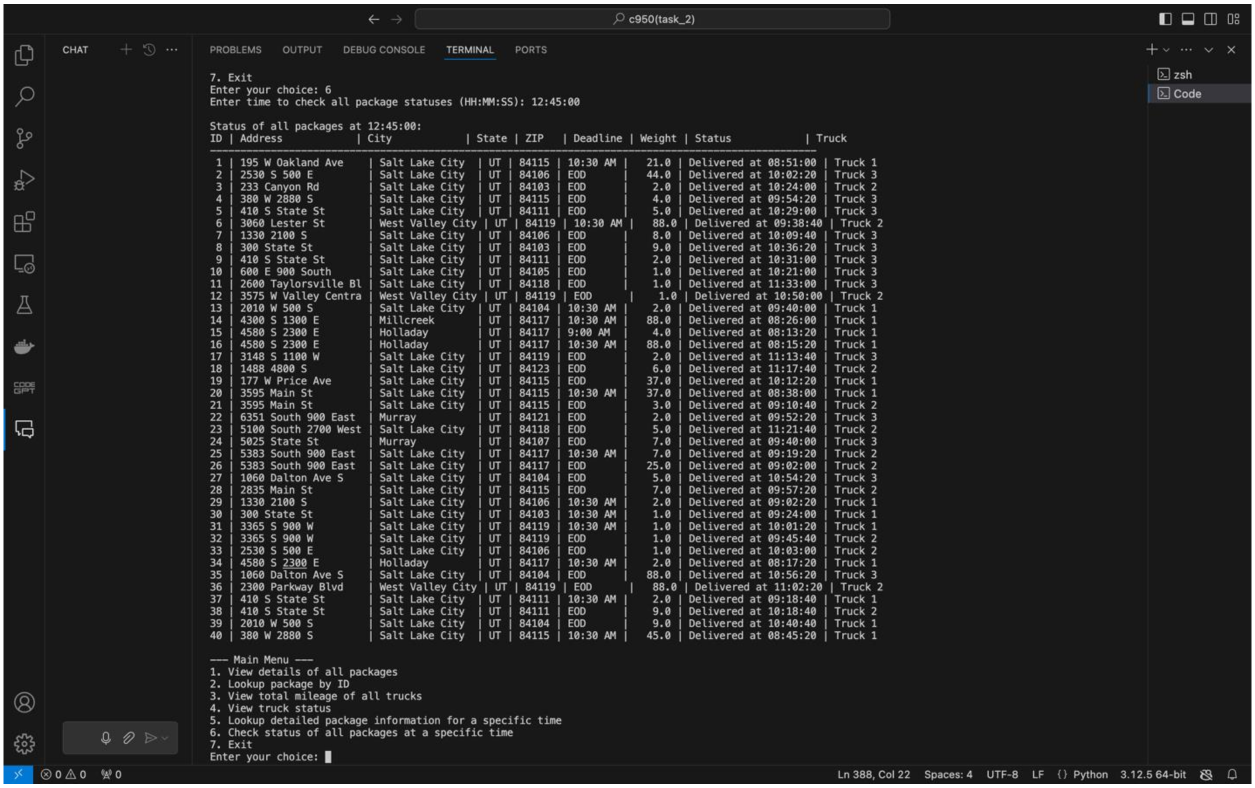
D1. First Status Check



D2. Second Status Check

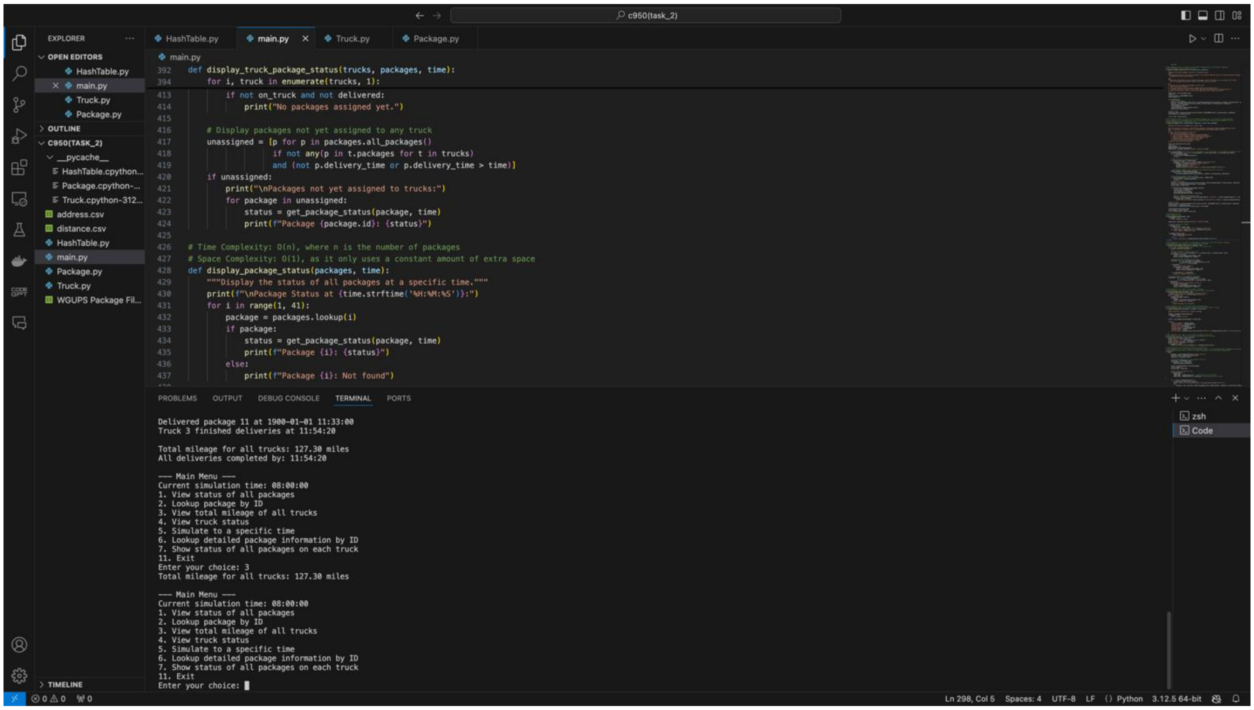


D3. Third Status Check



```
7. Exit
Enter your choice: 6
Enter time to check all package statuses (HH:MM:SS): 12:45:00

Status of all packages at 12:45:00:
ID | Address              | City             | State | ZIP   | Deadline | Weight | Status                | Truck
------------------------------------------------------------------------------------------------------------------
 1 | 195 W Oakland Ave     | Salt Lake City  | UT | 84115 | 10:30 AM |   21.0 | Delivered at 08:51:00 | Truck 1
 2 | 2530 S 500 E          | Salt Lake City  | UT | 84106 | EOD      |   44.0 | Delivered at 10:02:20 | Truck 3
 3 | 233 Canyon Rd         | Salt Lake City  | UT | 84103 | EOD      |    2.0 | Delivered at 10:24:00 | Truck 2
 4 | 380 W 2880 S          | Salt Lake City  | UT | 84115 | EOD      |    4.0 | Delivered at 09:54:20 | Truck 3
 5 | 410 S State St        | Salt Lake City  | UT | 84111 | EOD      |    5.0 | Delivered at 10:29:00 | Truck 3
 6 | 3060 Lester St        | West Valley City | UT | 84119 | 10:30 AM |   88.0 | Delivered at 09:38:40 | Truck 2
 7 | 1330 2100 S           | Salt Lake City  | UT | 84106 | EOD      |    8.0 | Delivered at 10:09:40 | Truck 3
 8 | 300 State St          | Salt Lake City  | UT | 84103 | EOD      |    9.0 | Delivered at 10:36:20 | Truck 3
 9 | 410 S State St        | Salt Lake City  | UT | 84111 | EOD      |    2.0 | Delivered at 10:31:00 | Truck 3
10 | 600 E 900 South       | Salt Lake City  | UT | 84105 | EOD      |    1.0 | Delivered at 10:21:00 | Truck 3
11 | 2600 Taylorsville Bl  | Salt Lake City  | UT | 84118 | EOD      |    1.0 | Delivered at 11:33:00 | Truck 3
12 | 3575 W Valley Centra  | West Valley City | UT | 84119 | EOD      |    1.0 | Delivered at 10:50:00 | Truck 2
13 | 2010 W 500 S          | Salt Lake City  | UT | 84104 | 10:30 AM |    2.0 | Delivered at 09:40:00 | Truck 1
14 | 4300 S 1300 E         | Millcreek       | UT | 84117 | 10:30 AM |   88.0 | Delivered at 08:26:00 | Truck 1
15 | 4580 S 2300 E         | Holladay        | UT | 84117 | 9:00 AM  |    4.0 | Delivered at 08:13:20 | Truck 1
16 | 4580 S 2300 E         | Holladay        | UT | 84117 | 10:30 AM |   88.0 | Delivered at 08:15:20 | Truck 1
17 | 3148 S 1100 W         | Salt Lake City  | UT | 84119 | EOD      |    2.0 | Delivered at 11:13:40 | Truck 3
18 | 1488 4800 S           | Salt Lake City  | UT | 84123 | EOD      |    6.0 | Delivered at 11:17:40 | Truck 2
19 | 177 W Price Ave       | Salt Lake City  | UT | 84115 | EOD      |   37.0 | Delivered at 10:12:20 | Truck 1
20 | 3595 Main St          | Salt Lake City  | UT | 84115 | 10:30 AM |   37.0 | Delivered at 08:38:00 | Truck 1
21 | 3595 Main St          | Salt Lake City  | UT | 84115 | EOD      |    3.0 | Delivered at 09:10:40 | Truck 2
22 | 6351 South 900 East   | Murray          | UT | 84121 | EOD      |    2.0 | Delivered at 09:52:20 | Truck 3
23 | 5100 South 2700 West  | Salt Lake City  | UT | 84118 | EOD      |    5.0 | Delivered at 11:21:40 | Truck 2
24 | 5025 State St         | Murray          | UT | 84107 | EOD      |    7.0 | Delivered at 09:40:00 | Truck 3
25 | 5383 South 900 East   | Salt Lake City  | UT | 84117 | 10:30 AM |    7.0 | Delivered at 09:19:20 | Truck 2
26 | 5383 South 900 East   | Salt Lake City  | UT | 84117 | EOD      |   25.0 | Delivered at 09:02:00 | Truck 2
27 | 1060 Dalton Ave S     | Salt Lake City  | UT | 84104 | EOD      |    5.0 | Delivered at 10:54:20 | Truck 3
28 | 2835 Main St          | Salt Lake City  | UT | 84115 | EOD      |    7.0 | Delivered at 09:57:20 | Truck 2
29 | 1330 2100 S           | Salt Lake City  | UT | 84106 | 10:30 AM |    2.0 | Delivered at 09:02:20 | Truck 1
30 | 300 State St          | Salt Lake City  | UT | 84103 | 10:30 AM |    1.0 | Delivered at 09:24:00 | Truck 1
31 | 3365 S 900 W          | Salt Lake City  | UT | 84119 | 10:30 AM |    1.0 | Delivered at 10:01:20 | Truck 1
32 | 3365 S 900 W          | Salt Lake City  | UT | 84119 | EOD      |    1.0 | Delivered at 09:45:40 | Truck 2
33 | 2530 S 500 E          | Salt Lake City  | UT | 84106 | EOD      |    1.0 | Delivered at 10:03:00 | Truck 2
34 | 4580 S 2300 E         | Holladay        | UT | 84117 | 10:30 AM |    2.0 | Delivered at 08:17:20 | Truck 1
35 | 1060 Dalton Ave S     | Salt Lake City  | UT | 84104 | EOD      |   88.0 | Delivered at 10:56:20 | Truck 3
36 | 2300 Parkway Blvd     | West Valley City | UT | 84119 | EOD      |   88.0 | Delivered at 11:02:20 | Truck 2
37 | 410 S State St        | Salt Lake City  | UT | 84111 | 10:30 AM |    2.0 | Delivered at 09:18:40 | Truck 1
38 | 410 S State St        | Salt Lake City  | UT | 84111 | EOD      |    9.0 | Delivered at 10:18:40 | Truck 2
39 | 2010 W 500 S          | Salt Lake City  | UT | 84104 | EOD      |    9.0 | Delivered at 10:40:40 | Truck 1
40 | 380 W 2880 S          | Salt Lake City  | UT | 84115 | 10:30 AM |   45.0 | Delivered at 08:45:20 | Truck 1

--- Main Menu ---
1. View details of all packages
2. Lookup package by ID
3. View total mileage of all trucks
4. View truck status
5. Lookup detailed package information for a specific time
6. Check status of all packages at a specific time
7. Exit
Enter your choice:
```

E. Screenshot of Code Execution



F1. Strengths of the Chosen Algorithm

1. **Efficiency and Simplicity**: The nearest neighbor algorithm used in the solution is computationally efficient and easy to implement. It makes locally optimal choices at each step by selecting the nearest undelivered package, which generally leads to a good (though not always optimal) overall route. This approach allows the algorithm to handle many packages and addresses without becoming computationally intractable, making it suitable for real-time route planning in delivery scenarios.

2. **Adaptability to Real-World Constraints**: The algorithm demonstrates strong adaptability to real-world delivery constraints. It handles time-sensitive deliveries by considering package deadlines

alongside distances. Additionally, it manages exceptional cases such as delayed packages and address updates (e.g., Package 9) seamlessly within the delivery process. This flexibility allows the algorithm to produce realistic and practical delivery routes that accommodate package delivery operations' dynamic nature.

These strengths make the algorithm well-suited for the given package delivery problem, balancing computational efficiency with the ability to handle complex, real-world delivery scenarios.

F2. Verification of Algorithm

Verification of Algorithm Requirements:

Package Delivery Deadline Compliance: The algorithm considers package deadlines when determining the delivery order. In the deliver_packages function, packages are prioritized based on their deadlines:

Special Notes Handling: The algorithm handles special notes for packages. For example:

- Delayed packages are handled in assign_packages_to_trucks.
- Packages that can only be on truck two are specifically assigned.
- The algorithm also handles packages that must be shipped in the same truck.
- The algorithm uses the truck speed limit of 18 mph when calculating travel times.

- The delivery simulation starts at 8:00 AM, as specified.
- After delivering all packages, each truck returns to the hub.
- The algorithm calculates and reports the total mileage for all trucks.
- The algorithm provides functionality to check the status of any package at any time.

The algorithm implemented in the solution meets all the specified requirements of the scenario, handling package deadlines, special notes, address updates, and truck constraints and providing the necessary reporting and status check functionalities.

F3. Other Possible Algorithms

The two alternative algorithms I would have chosen are the Two-Phase Hybrid and Geographical Clustering.

F3a. Algorithm Differences

The Two-Phase Hybrid Algorithm and the Geographical Clustering Algorithm provide more advanced approaches to solving routing problems than the Nearest Neighbor algorithm. While Nearest Neighbor focuses on local optimization by selecting the closest unvisited location at each step, the Two-Phase Hybrid Algorithm adopts a global perspective. In the construction phase, it builds complete routes, and in the improvement phase, it refines the overall solution, leading to better solution quality. Nearest Neighbor, conversely, can quickly get stuck in suboptimal routes due to its greedy nature. The Two-Phase Hybrid also has greater flexibility in incorporating complex constraints and objectives, though it can be more computationally intensive, particularly during the refinement phase.

The Geographical Clustering Algorithm differs from Nearest Neighbor by decomposing the routing problem into smaller clusters based on geographical proximity, making it more scalable for more extensive delivery networks. Nearest Neighbor handles all points as a single problem, often leading to inefficient routes with crossovers, whereas clustering allows for more naturally structured routes. Additionally, Nearest Neighbor is primarily focused on minimizing the distance between consecutive points, while Geographical Clustering can incorporate diverse objectives, such as balancing workloads across delivery vehicles. Both alternative algorithms are better suited to handling more extensive and complex scenarios, producing higher-quality solutions through a more holistic consideration of the problem structure.

G. Different Approach

If I revisited this package delivery project, I would implement several vital modifications to enhance the system’s efficiency, flexibility, and ability to adapt to real-world conditions. A significant improvement would be the introduction of dynamic route planning, enabling the system to adjust routes in real time based on new information, such as traffic conditions, urgent deliveries, or vehicle breakdowns. Additionally, I would implement a more sophisticated time window optimization system, allowing for better scheduling precision, such as handling updates like the address change for

Package 9. Expanding the system to consider multi-objective optimization would improve results by balancing multiple factors, including minimizing travel time, maximizing on-time deliveries, and reducing fuel consumption, all while balancing truck workloads. Machine learning integration could also play a vital role by predicting delivery times based on historical data, accounting for variables like traffic, weather, and time of day to create more accurate route plans.

I would adopt more advanced data structures like R-trees or quad-trees to efficiently handle large datasets and spatial data to support these changes. Redesigning the system with a modular architecture would make updating or swapping components easier, facilitating testing and system improvements. Developing a simulation and testing framework would help fine-tune algorithms and better prepare the system for real-world variability. A graphical user interface (GUI) with real-time maps and a dashboard to monitor key performance indicators would further enhance usability and system monitoring. Scalability would be addressed through parallel processing techniques and distributed computing, enabling the system to handle large-scale delivery operations. Integration with external systems, such as traffic monitoring and weather services, would improve real-time adaptability, while customer notification systems would enhance service quality. These changes would result in a more robust, adaptable, and scalable solution for package delivery.

H. Verification of Data Structure

The primary data structure utilized in the solution is a hash table (HashTable class), which effectively satisfies the scenario's requirements by offering O(1) average-case time complexity for package lookup operations. This efficiency is crucial for quickly retrieving and updating package information during delivery. The hash table stores all necessary package details, including special notes and deadlines, making it well-suited for managing constraints such as delivery deadlines and address updates, like the change for Package 9 at 10:20 AM. With an initial capacity of 40 (HashTable(40)), it efficiently accommodates all 40 packages, ensuring that they can be stored, updated, and accessed in a timely manner, meeting the problem's scalability and performance needs.

H1. Other Data Structures

Alternative data structures, such as a Binary Search Tree (BST) or an array-based list with binary search, could also be effective for the package delivery system. A self-balancing BST, such as an AVL or Red-Black tree, offers O(log n) time complexity for search, insert, and delete operations, making it efficient for the scale of 40 packages. It supports dynamic updates, which is crucial for handling real-time changes like the address update for Package 9. Naturally, it keeps packages sorted by ID, which can be advantageous for reporting or processing. Each node in the tree can store necessary package details, such as addresses and deadlines, providing flexibility in managing package information. The BST implementation could be structured for efficient insertion, searching, and updating of package data as the delivery system processes various constraints.

An array-based list combined with binary search is another alternative, providing simplicity and efficiency for a fixed number of packages. Although insert and delete operations are O(n), binary search ensures O(log n) search times, making it suitable for 40 packages. The straightforward nature of Python's built-in list functionality makes this approach easy to implement and iterate while being memory efficient compared to a hash table. The array-based list simplifies tasks that require sorting or processing packages in order, and dynamic updates can be achieved by searching for a package via binary search and updating the necessary details. The BST and array-based list approaches offer efficient lookup and update capabilities, and the choice between them would depend on specific implementation needs and preferences for the system.

H1a. Data Structure Differences

The Binary Search Tree (BST) and Hash Table offer distinct trade-offs when compared. A Hash Table provides O(1) average time complexity for insert, delete, and search operations, making it highly efficient for quick lookups. However, it lacks inherent ordering and requires collision resolution strategies, which adds complexity. In contrast, a BST ensures O(log n) time complexity for operations while maintaining elements in sorted order. This suits it better for scenarios where ordered data or range queries are essential. Though a BST may require more complex balancing mechanisms in the case of self-balancing trees, it is generally more space-efficient, especially when dealing with smaller datasets, as it does not have the unused buckets that can occur in Hash Tables.

Similarly, an array-based list with binary search offers different advantages over a Hash Table. While a Hash Table is faster for most operations with O(1) average time complexity, an array-based list allows for ordered storage and more straightforward memory usage since it only holds the actual elements. Although it has O(log n) search time with binary search, its insert and delete operations are slower at O(n). Additionally, the array-based list is more straightforward to implement, leveraging built-in list functionalities in many programming languages, and supports easy, ordered iteration through elements. In contrast, the Hash Table requires more complex implementation considerations, such as hash functions and dynamic resizing when the load factor increases. Ultimately, the choice between these data structures depends on specific project needs, such as whether fast lookup or ordered data is prioritized and the simplicity or complexity of implementation.

J. Professional Communication