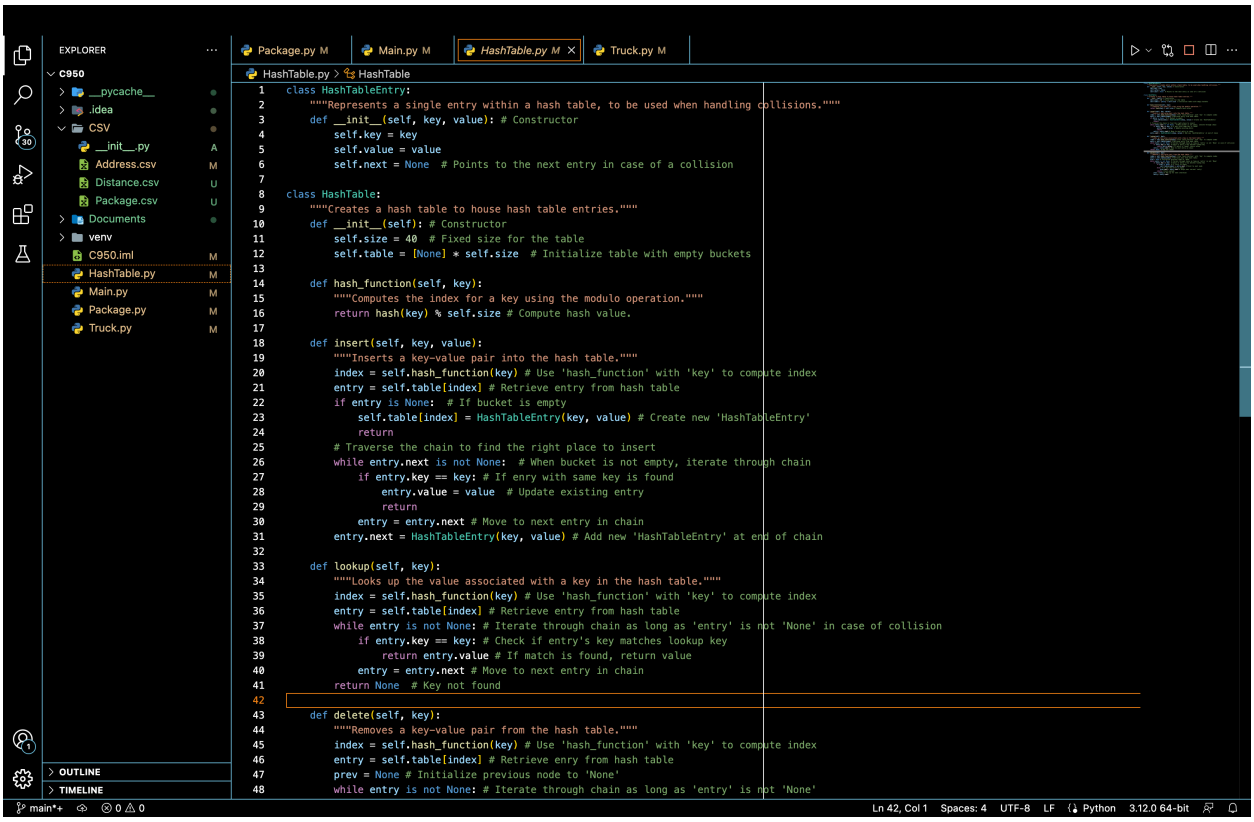
C950 Task-2 WGUPS Write-Up

Eric Jacobs
Student ID: 010580832
WGU Email: ejaco70@wgu.edu
11/17/2023

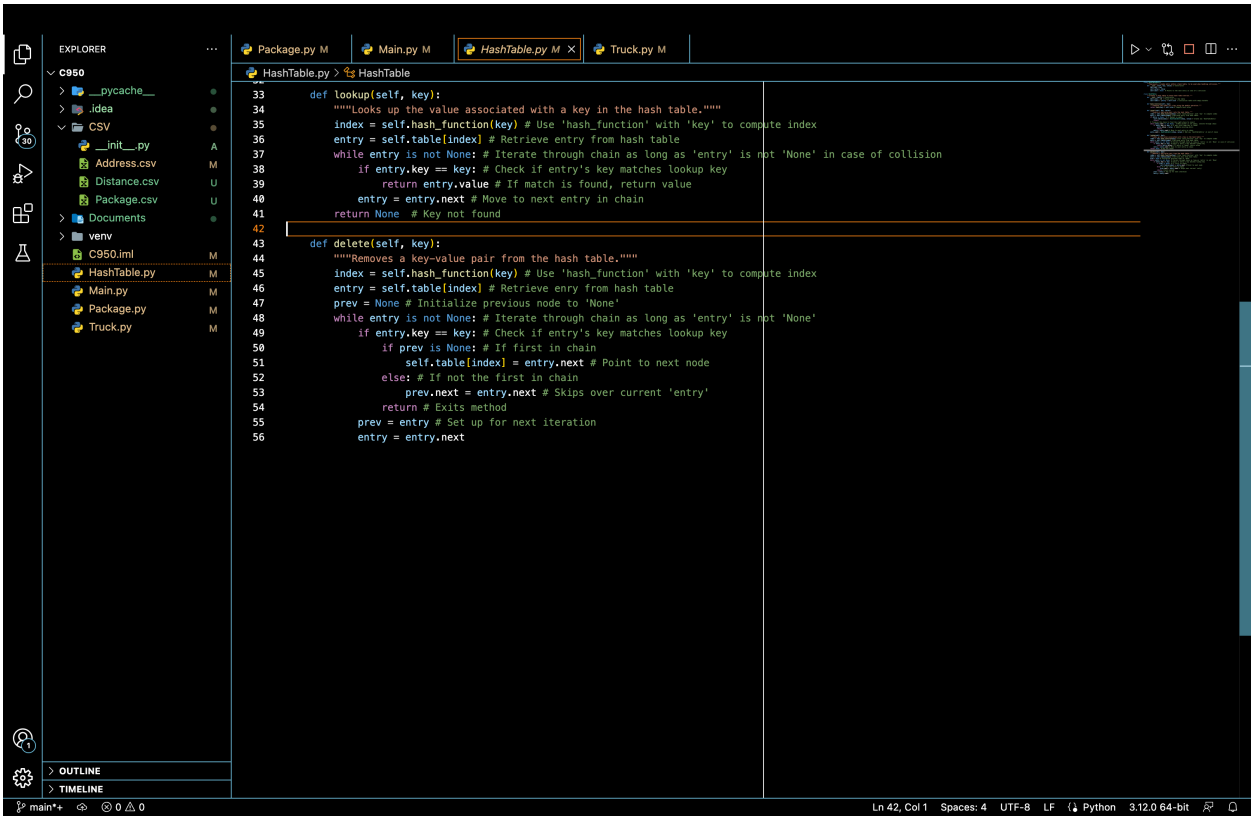
C950 Data Structures and Algorithms II

A. Hash Table Screenshot



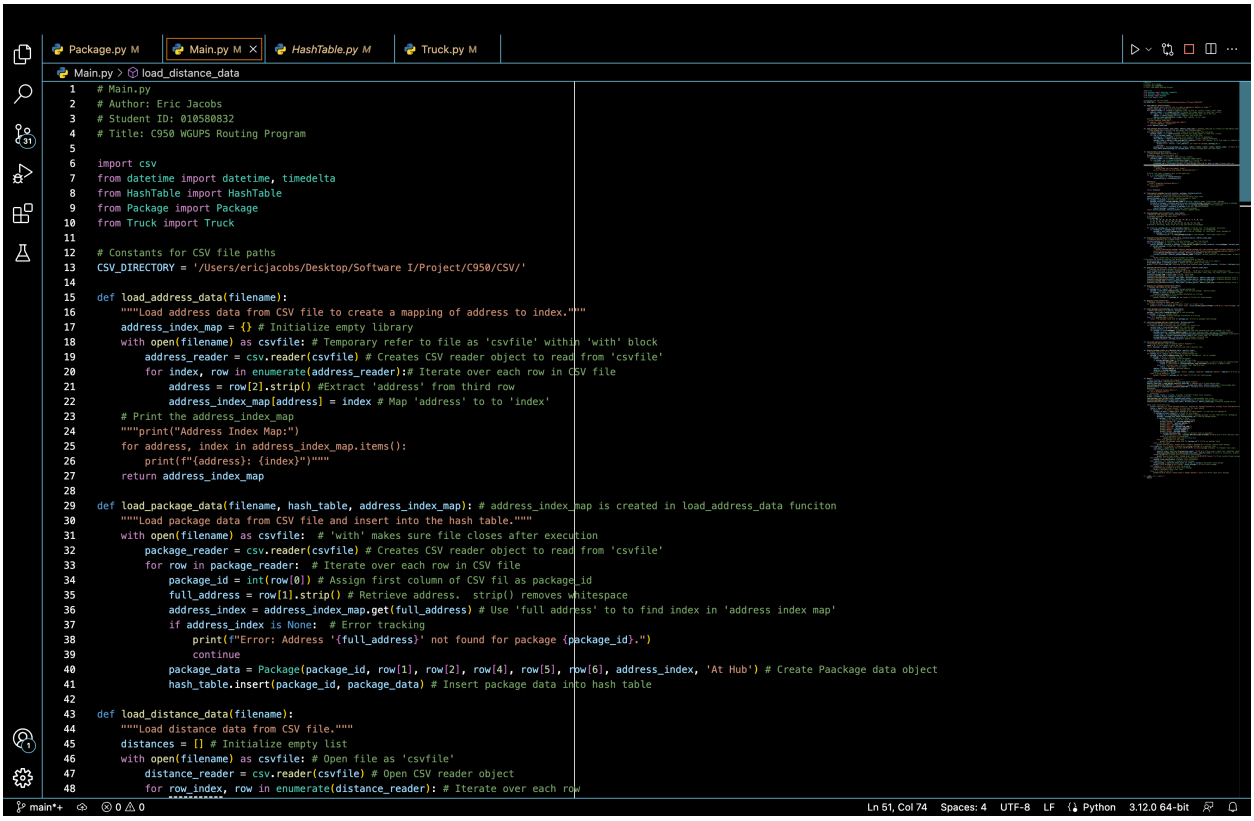
B. Lookup Function Screenshot

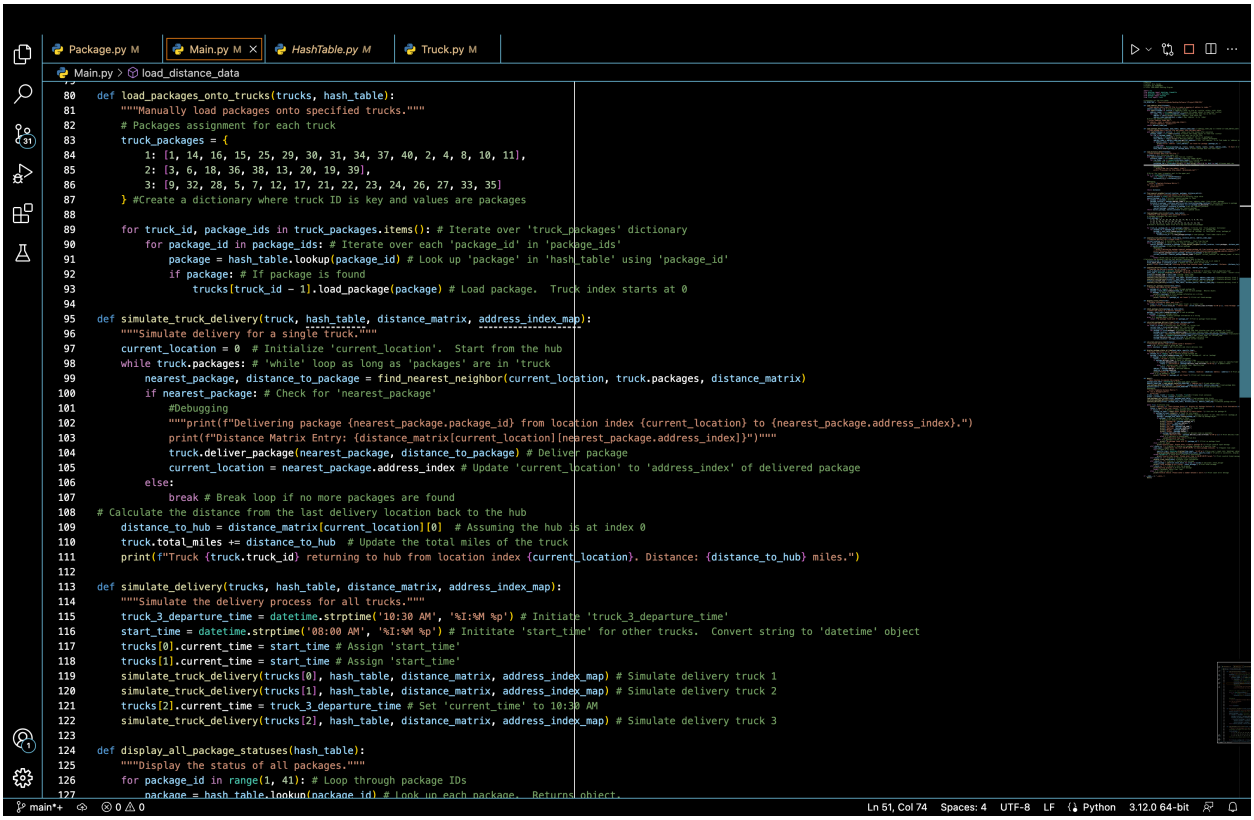
-The ‘lookup’ function in the hash table retrieves the ‘Package’ object associated with a given package ID. The ‘Package’ object itself contains all the detailed information required, making the retrieval of specific package aligned with the stated requirements as demonstrated in Option 1 “Check Package Status” in user interface (bottom screenshot)



C. Original Code

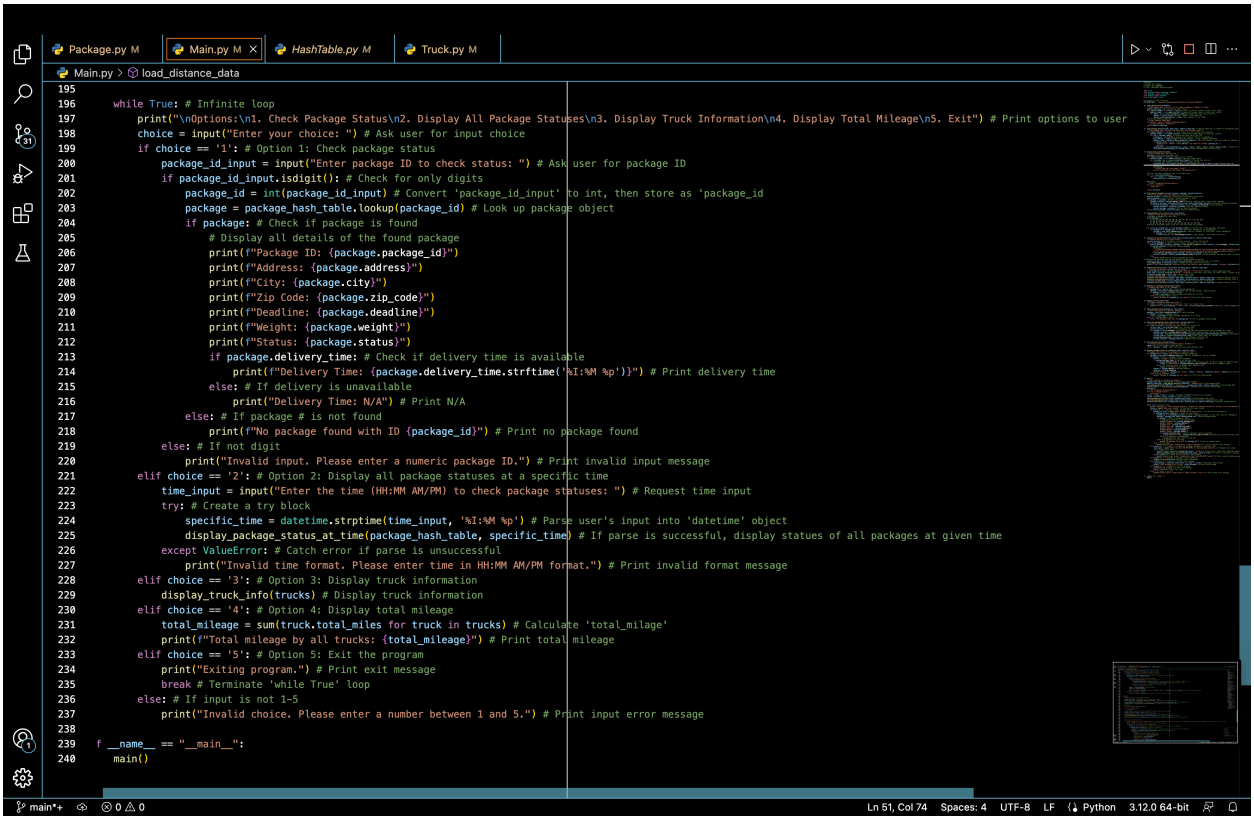
Major Code Blocks




Package.py M
Main.py M
HashTable.py M
Truck.py M
Main.py > load_distance_data
def load_packages_onto_trucks(trucks, hash_table):
"""Manually load packages onto specified trucks."""
# Packages assignment for each truck
truck_packages = {
1: [1, 14, 16, 15, 25, 29, 30, 31, 34, 37, 40, 2, 4, 8, 10, 11],
2: [3, 6, 18, 36, 38, 13, 20, 19, 39],
3: [9, 32, 28, 5, 7, 12, 17, 21, 22, 23, 24, 26, 27, 33, 35]
} #Create a dictionary where truck ID is key and values are packages
for truck_id, package_ids in truck_packages.items(): # Iterate over 'truck_packages' dictionary
for package_id in package_ids: # Iterate over each 'package_id' in 'package_ids'
package = hash_table.lookup(package_id) # Look up 'package' in 'hash_table' using 'package_id'
if package: # If package is found
trucks[truck_id - 1].load_package(package) # Load package. Truck index starts at 0
def simulate_truck_delivery(truck, hash_table, distance_matrix, address_index_map):
"""Simulate delivery for a single truck."""
current_location = 0 # Initialize 'current_location'. Start from the hub
while truck.packages: # 'while' loop as long as 'packages' are in 'truck
nearest_package, distance_to_package = find_nearest_neighbor(current_location, truck.packages, distance_matrix)
if nearest_package: # Check for 'nearest_package'
#Debugging
"""print(f"Delivering package {nearest_package.package_id} from location index {current_location} to {nearest_package.address_index}.")
print(f"Distance Matrix Entry: {distance_matrix[current_location][nearest_package.address_index]}")"""
truck.deliver_package(nearest_package, distance_to_package) # Deliver package
current_location = nearest_package.address_index # Update 'current_location' to 'address_index' of delivered package
else:
break # Break loop if no more packages are found
# Calculate the distance from the last delivery location back to the hub
distance_to_hub = distance_matrix[current_location][0] # Assuming the hub is at index 0
truck.total_miles += distance_to_hub # Update the total miles of the truck
print(f"Truck {truck.truck_id} returning to hub from location index {current_location}. Distance: {distance_to_hub} miles.")
def simulate_delivery(trucks, hash_table, distance_matrix, address_index_map):
"""Simulate the delivery process for all trucks."""
truck_3_departure_time = datetime.strptime('10:30 AM', '%I:%M %p') # Initiate 'truck_3_departure_time'
start_time = datetime.strptime('08:00 AM', '%I:%M %p') # Inititate 'start_time' for other trucks. Convert string to 'datetime' object
trucks[0].current_time = start_time # Assign 'start_time'
trucks[1].current_time = start_time # Assign 'start_time'
simulate_truck_delivery(trucks[0], hash_table, distance_matrix, address_index_map) # Simulate delivery truck 1
simulate_truck_delivery(trucks[1], hash_table, distance_matrix, address_index_map) # Simulate delivery truck 2
trucks[2].current_time = truck_3_departure_time # Set 'current_time' to 10:30 AM
simulate_truck_delivery(trucks[2], hash_table, distance_matrix, address_index_map) # Simulate delivery truck 3
def display_all_package_statuses(hash_table):
"""Display the status of all packages."""
for package_id in range(1, 41): # Loop through package IDs
package = hash_table.lookup(package_id) # Look up each package. Returns object.
main*+
0 0
Ln 51, Col 74
Spaces: 4
UTF-8
LF
Python
3.12.0 64-bit


Package.py M
Main.py M
HashTable.py M
Truck.py M
Main.py > load_distance_data
def display_all_package_statuses(hash_table):
"""Display the status of all packages."""
for package_id in range(1, 41): # Loop through package IDs
package = hash_table.lookup(package_id) # Look up each package. Returns object.
if package: # Check if package is found
print(str(package)) # Print package information as a string
else: # If package doesn't exist
print(f"Package ID {package_id} not found.") # Print not found message
def display_truck_info(trucks):
"""Display information about each truck."""
for truck in trucks: # Iterate through each 'truck' in 'trucks' list
print(f"Truck {truck.truck_id} -> Return Time: {truck.current_time.strftime('%I:%M %p')}, Total Mileage: {truck.total_miles}, Packages: {[p.package_id for p in truck.pa
def check_package_status(package_id, hash_table):
"""Check the status of a specific package."""
package = hash_table.lookup(package_id) # Look up package.
if package: # Check if package exists
return str(package) # Return package information as a string
else: # If package doesn't exist
return f"No package found with ID {package_id}" # Print no package found message
def calculate_package_delivery_times(trucks, distance_matrix):
"""Calculate the delivery times for all packages."""
for truck in trucks: # Iterate over each 'truck' in 'trucks list
current_time = truck.current_time # Set 'current_time'
current_location = 0 # Set 'current_location' to hub
for package in truck.packages: # Initiate nested loop that iterates over each 'package' in 'truck'
package_location = package.address_index # Retrieve 'address_index' and set as 'package_location'
travel_time = calculate_delivery_time(distance_matrix[current_location][package_location]) # Calculate travel time to package
current_time += timedelta(minutes=travel_time) # Add 'travel_time' to 'current_time'
package.delivery_time = current_time # Set package's delivery time
current_location = package_location # Update current location
def calculate_delivery_time(distance):
"""Calculate delivery time in minutes given a distance."""
speed = 18 # Truck speed in miles per hour
return (distance / speed) * 60 # Calculate and return delivery time
def display_package_status_at_time(hash_table, specific_time):
"""Display the status of all packages at a specific time."""
for package_id in range(1, 41): # Iterate through package IDs
package = hash_table.lookup(package_id) # Look up 'package-id', set as 'package'
if package: # Check if package is found
status = "At Hub" # Initial status assignment
if package.delivery_time: # Check for delivery time
if package.delivery_time <= specific_time: # If 'delivery_time' is less or equal to 'specific-time'
status = f"Delivered at {package.delivery_time.strftime('%I:%M %p')}" # Update status
main*+
0 0
Ln 51, Col 74
Spaces: 4
UTF-8
LF
Python
3.12.0 64-bit

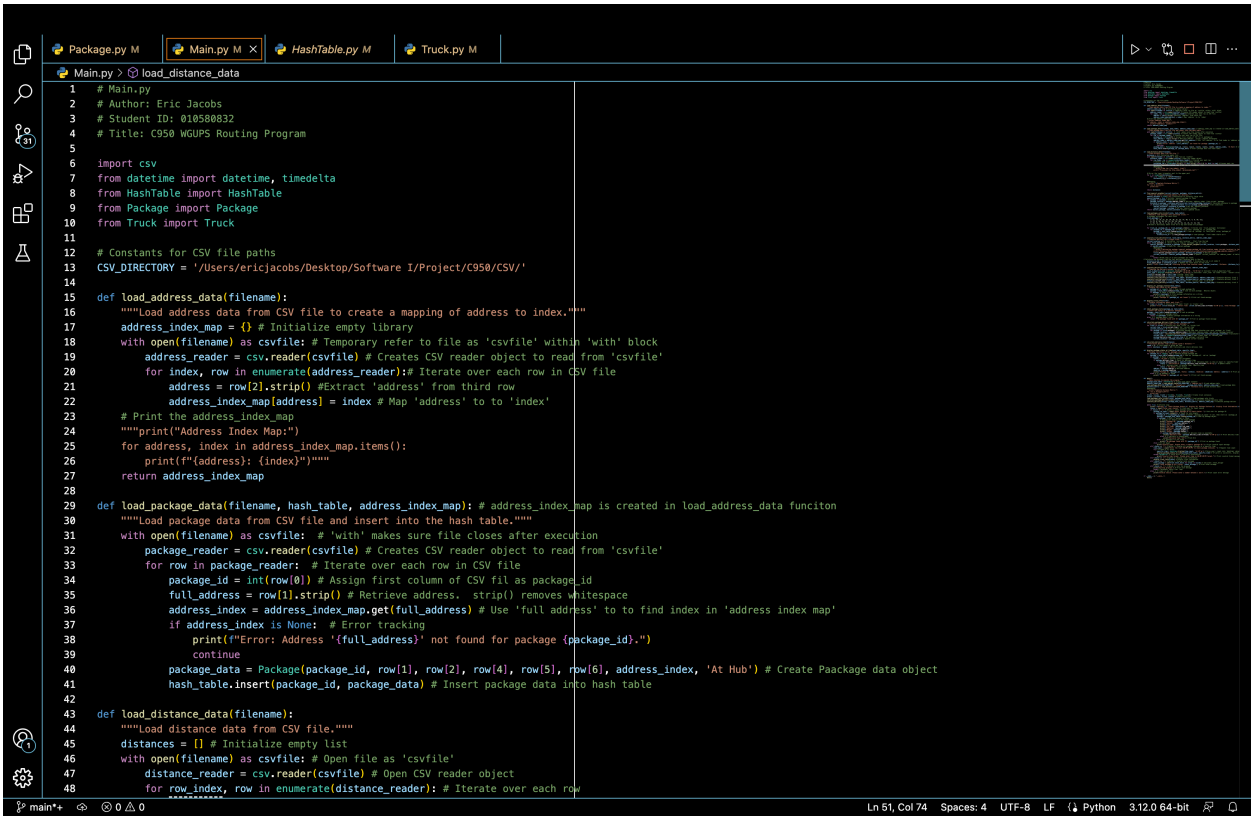

Package.py M
Main.py M
HashTable.py M
Truck.py M
Main.py > load_distance_data
def display_package_status_at_time(hash_table, specific_time):
"""Display the status of all packages at a specific time."""
for package_id in range(1, 41): # Iterate through package IDs
package = hash_table.lookup(package_id) # Look up 'package-id', set as 'package'
if package: # Check if package is found
status = "At Hub" # Initial status assignment
if package.delivery_time: # Check for delivery time
if package.delivery_time <= specific_time: # If 'delivery_time' is less or equal to 'specific-time'
status = f"Delivered at {package.delivery_time.strftime('%I:%M %p')}" # Update status
else: # If 'delivery_time' is greater than 'specific_time'
status = "En Route" # Set status
address = package.address # Retrieve address
deadline = package.deadline
print(f"Package ID: {package_id}, Status: {status}, Deadline: {deadline} Address: {address}") # Print package information
else: # If no package is found
print(f"Package ID {package_id} not found.") # Print not found message
def main():
"""Main function to execute the program."""
package_hash_table = HashTable() # Create hash table instance
address_index_map = load_address_data(CSV_DIRECTORY + 'Address.csv') # Load address data
load_package_data(CSV_DIRECTORY + 'Package.csv', package_hash_table, address_index_map) # Load package data
distance_matrix = load_distance_data(CSV_DIRECTORY + 'Distance.csv') # Load distance data
#Debugging
"""print("Complete Distance Matrix:")
for row in distance_matrix:
print(row)"""
truck1, truck2, truck3 = Truck(1), Truck(2), Truck(3) # Create truck instances
trucks = [truck1, truck2, truck3] # Create truck list
load_packages_onto_trucks(trucks, package_hash_table) # Load packages onto trucks
calculate_package_delivery_times(trucks, distance_matrix) # Calculate package delivery times
simulate_delivery(trucks, package_hash_table, distance_matrix, address_index_map) # Simulate package deliver
while True: # Infinite loop
print("\nOptions:\n1. Check Package Status\n2. Display All Package Statuses\n3. Display Truck Information\n4. Display Total Mileage\n5. Exit") # Print options to user
choice = input("Enter your choice: ") # Ask user for input choice
if choice == '1': # Option 1: Check package status
package_id_input = input("Enter package ID to check status: ") # Ask user for package ID
if package_id_input.isdigit(): # Check for only digits
package_id = int(package_id_input) # Convert 'package_id_input' to int, then store as 'package_id
package = package_hash_table.lookup(package_id) # Look up package object
if package: # Check if package is found
# Display all details of the found package
print(f"Package ID: {package.package_id}")
print(f"Address: {package.address}")
print(f"City: {package.city}")
print(f"Zip Code: {package.zip_code}")
print(f"Deadline: {package.deadline}")
main*+
0 0
Ln 51, Col 74
Spaces: 4
UTF-8
LF
Python
3.12.0 64-bit


Package.py M
Main.py M
HashTable.py M
Truck.py M
Main.py > load_distance_data
while True: # Infinite loop
print("\nOptions:\n1. Check Package Status\n2. Display All Package Statuses\n3. Display Truck Information\n4. Display Total Mileage\n5. Exit") # Print options to user
choice = input("Enter your choice: ") # Ask user for input choice
if choice == '1': # Option 1: Check package status
package_id_input = input("Enter package ID to check status: ") # Ask user for package ID
if package_id_input.isdigit(): # Check for only digits
package_id = int(package_id_input) # Convert 'package_id_input' to int, then store as 'package_id
package = package_hash_table.lookup(package_id) # Look up package object
if package: # Check if package is found
# Display all details of the found package
print(f"Package ID: {package.package_id}")
print(f"Address: {package.address}")
print(f"City: {package.city}")
print(f"Zip Code: {package.zip_code}")
print(f"Deadline: {package.deadline}")
print(f"Weight: {package.weight}")
print(f"Status: {package.status}")
if package.delivery_time: # Check if delivery time is available
print(f"Delivery Time: {package.delivery_time.strftime('%I:%M %p')}") # Print delivery time
else: # If delivery is unavailable
print("Delivery Time: N/A") # Print N/A
else: # If package # is not found
print(f"No package found with ID {package_id}") # Print no package found
else: # If not digit
print("Invalid input. Please enter a numeric package ID.") # Print invalid input message
elif choice == '2': # Option 2: Display all package statuses at a specific time
time_input = input("Enter the time (HH:MM AM/PM) to check package statuses: ") # Request time input
try: # Create a try block
specific_time = datetime.strptime(time_input, '%I:%M %p') # Parse user's input into 'datetime' object
display_package_status_at_time(package_hash_table, specific_time) # If parse is successful, display statues of all packages at given time
except ValueError: # Catch error if parse is unsuccessful
print("Invalid time format. Please enter time in HH:MM AM/PM format.") # Print invalid format message
elif choice == '3': # Option 3: Display truck information
display_truck_info(trucks) # Display truck information
elif choice == '4': # Option 4: Display total mileage
total_mileage = sum(truck.total_miles for truck in trucks) # Calculate 'total_milage'
print(f"Total mileage by all trucks: {total_mileage}") # Print total mileage
elif choice == '5': # Option 5: Exit the program
print("Exiting program.") # Print exit message
break # Terminate 'while True' loop
else: # If input is not 1-5
print("Invalid choice. Please enter a number between 1 and 5.") # Print input error message
f __name__ == "__main__":
main()
main*+
0 0
Ln 51, Col 74
Spaces: 4
UTF-8
LF
Python
3.12.0 64-bit


Package.py M
Main.py M
HashTable.py M
Truck.py M
Package.py > ...
class Package:
def __init__(self, package_id, address, city, zip_code, deadline, weight, address_index, status='At the hub'): # Constructor
"""
Initialize a new package with given details.
:param package_id: Unique identifier for the package.
:param address: Delivery address of the package.
:param city: City for the delivery address.
:param zip_code: Zip code for the delivery address.
:param deadline: Delivery deadline for the package.
:param weight: Weight of the package.
:param address_index: Index of the package's address in the distance matrix.
:param status: Current status of the package (default is 'At the hub').
"""
self.package_id = package_id
self.address = address
self.city = city
self.zip_code = zip_code
self.deadline = deadline
self.weight = weight
self.address_index = address_index
self.status = status # Current status of package
self.delivery_time = None # Time when the package is delivered, None if not yet delivered.
self.is_delivered = False # Boolean to track whether the package has been delivered.
def update_status(self, status, delivery_time=None):
"""
Update the status and delivery time of the package.
:param status: New status to be assigned to the package.
:param delivery_time: The time when the package was delivered.
"""
self.status = status # Update status
self.is_delivered = (status == 'Delivered') # Set 'is_delivered_ to 'True' if status is 'Delievered
if delivery_time: # If 'delivery_time is provided
self.delivery_time = delivery_time # Update 'delivery_time'
self.delivery_time_formatted = delivery_time.strftime('%I:%M %p') # Formatted delivery time.
def __str__(self):
"""
String representation of the package's details including delivery time.
"""
delivery_time_formatted = self.delivery_time.strftime('%I:%M %p') if self.delivery_time else 'N/A' # Convert 'delivery_time' to string, otherwise N/A
return (f"Package ID: {self.package_id}, Address: {self.address}, City: {self.city}, "
f"Zip: {self.zip_code}, Deadline: {self.deadline}, Weight: {self.weight}, "
f"Status: {self.status}, Delivery Time: {delivery_time_formatted}") # Return formatted string
main*+
0 0
Ln 46, Col 110
Spaces: 4
UTF-8
LF
Python
3.12.0 64-bit


Package.py M
Main.py M
HashTable.py M
Truck.py M
Truck.py > Truck > return_to_hub
class Truck:
def __init__(self, truck_id, delivery_speed=18, capacity=16): # Constructor
self.truck_id = truck_id # Initialize truck ID
self.delivery_speed = delivery_speed # Speed of the truck in miles per hour
self.capacity = capacity # Maximum number of packages the truck can carry
self.packages = [] # List to store packages loaded onto the truck
self.total_miles = 0 # Total miles traveled by the truck
self.current_location_index = 0 # Assuming index 0 is the hub
self.current_time = datetime.strptime('08:00 AM', '%I:%M %p') # Default starting time
def load_package(self, package):
"""Load a package onto the truck if there's available capacity."""
if len(self.packages) < self.capacity: # Check if number of 'packages' is less than trucks 'capacity'
self.packages.append(package) # Add 'package' to 'packages'
package.update_status('En Route') # Update package status to 'En Route'
else: # If truck is full
print(f"Truck {self.truck_id} is full and cannot load any more packages.") # Print full statement
def deliver_package(self, package, distance):
"""Deliver a package and update truck's total mileage and time."""
# Ensure package #9 is not delivered before 10:20 AM
update_time = datetime.strptime('10:20 AM', '%I:%M %p') # Initialize 'update_time'
if package.package_id == 9 and self.current_time < update_time: # Check if package #9 and before 10:20 AM
print("Package #9 cannot be delivered before 10:20 AM. It will be reattempted later.") # Print unsuccessful delivery message
return False # Delivery attempt was unsuccessful
# Print the distance being used for delivery
print(f"Delivering package {package.package_id} from location index {self.current_location_index} to {package.address_index}. Distance: {distance} miles.")
# Calculate delivery time based on distance
delivery_time = self.current_time + timedelta(hours=distance / self.delivery_speed)
package.update_status('Delivered', delivery_time) # Update package status to 'Delivered'
self.total_miles += distance # Update total miles
self.current_location_index = package.address_index # Update current location
self.current_time = delivery_time # Update current time
return True # Delivery was successful
def return_to_hub(self, distance_matrix):
"""Simulate the truck's return to the hub and update its mileage."""
return_distance = distance_matrix[self.current_location_index][0] # Calculate 'return_distance'
self.total_miles += return_distance # Update total miles for the return trip
self.current_location_index = 0 # Reset location to the hub
self.current_time += timedelta(hours=return_distance / self.delivery_speed) # Calculate and update 'current_time'
print(f"Truck {self.truck_id} returning to hub, total miles: {self.total_miles}") # Print return message
def __str__(self):
"""String representation of the truck's current package load."""
return f"Truck {self.truck_id} -> Packages: {[p.package_id for p in self.packages]}" # Create and return formatted string
main*+
0 0
Ln 44, Col 4
Spaces: 4
UTF-8
LF
Python
3.12.0 64-bit

C1: Identification Information



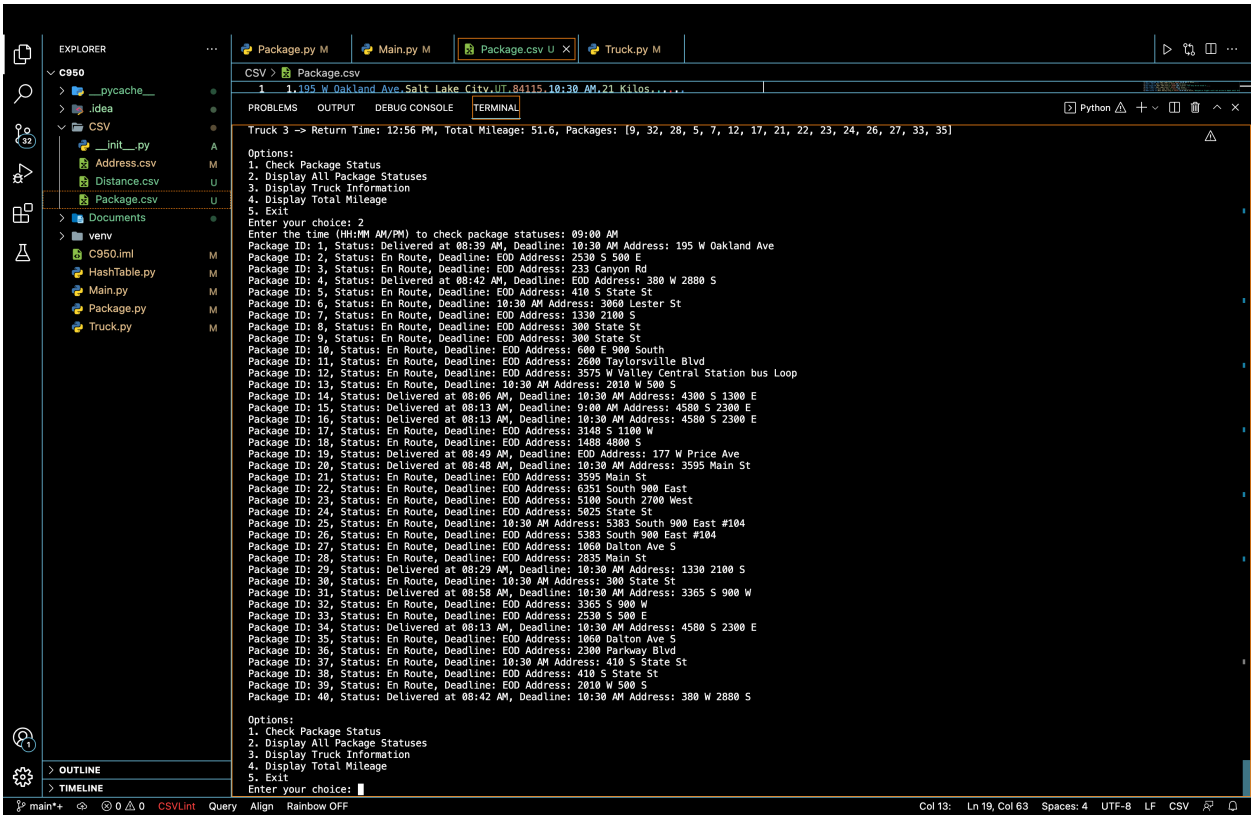
C2: Process and Flow Comments

```python
def simulate_truck_delivery(truck, hash_table, distance_matrix, address_index_map):
    """Simulate delivery for a single truck."""
    current_location = 0  # Initialize 'current_location'.  Start from the hub
    while truck.packages: # 'while' loop as long as 'packages' are in 'truck
        nearest_package, distance_to_package = find_nearest_neighbor(current_location, truck.packages, distance_matrix)
        if nearest_package: # Check for 'nearest_package'
            """print(f"Delivering package {nearest_package.package_id} from location index {current_location} to {nearest_package.address_index}.")
            print(f"Distance Matrix Entry: {distance_matrix[current_location][nearest_package.address_index]}")"""
            truck.deliver_package(nearest_package, distance_to_package) # Deliver package
            current_location = nearest_package.address_index # Update 'current_location' to 'address_index' of delivered package
        else:
            break # Break loop if no more packages are found
# Calculate the distance from the last delivery location back to the hub
    distance_to_hub = distance_matrix[current_location][0]  # Assuming the hub is at index 0
    truck.total_miles += distance_to_hub  # Update the total miles of the truck
    print(f"Truck {truck.truck_id} returning to hub from location index {current_location}. Distance: {distance_to_hub} miles.")

def simulate_delivery(trucks, hash_table, distance_matrix, address_index_map):
    """Simulate the delivery process for all trucks."""
    truck_3_departure_time = datetime.strptime('10:30 AM', '%I:%M %p') # Initiate 'truck_3_departure_time'
    start_time = datetime.strptime('08:00 AM', '%I:%M %p') # Inititate 'start_time' for other trucks.  Convert string to 'datetime' object
    trucks[0].current_time = start_time # Assign 'start_time'
    trucks[1].current_time = start_time # Assign 'start_time'
    simulate_truck_delivery(trucks[0], hash_table, distance_matrix, address_index_map) # Simulate delivery truck 1
    simulate_truck_delivery(trucks[1], hash_table, distance_matrix, address_index_map) # Simulate delivery truck 2
    trucks[2].current_time = truck_3_departure_time # Set 'current_time' to 10:30 AM
    simulate_truck_delivery(trucks[2], hash_table, distance_matrix, address_index_map) # Simulate delivery truck 3

def display_all_package_statuses(hash_table):
    """Display the status of all packages."""
    for package_id in range(1, 41): # Loop through package IDs
        package = hash_table.lookup(package_id) # Look up each package.  Returns object.
        if package: # Check if package is found
            print(str(package)) # Print package information as a string
        else: # If package doesn't exist
            print(f"Package ID {package_id} not found.") # Print not found message

def display_truck_info(trucks):
    """Display information about each truck."""
    for truck in trucks: # Iterate through each 'truck' in 'trucks' list
        print(f"Truck {truck.truck_id} -> Return Time: {truck.current_time.strftime('%I:%M %p')}, Total Mileage: {truck.total_miles}, Packages: {[p.package_id for p in truck.p

def check_package_status(package_id, hash_table):
    """Check the status of a specific package."""
    package = hash_table.lookup(package_id) # Look up package.
    if package: # Check if package exists
        return str(package) # Return package information as a string
    else: # If package doesn't exist
```

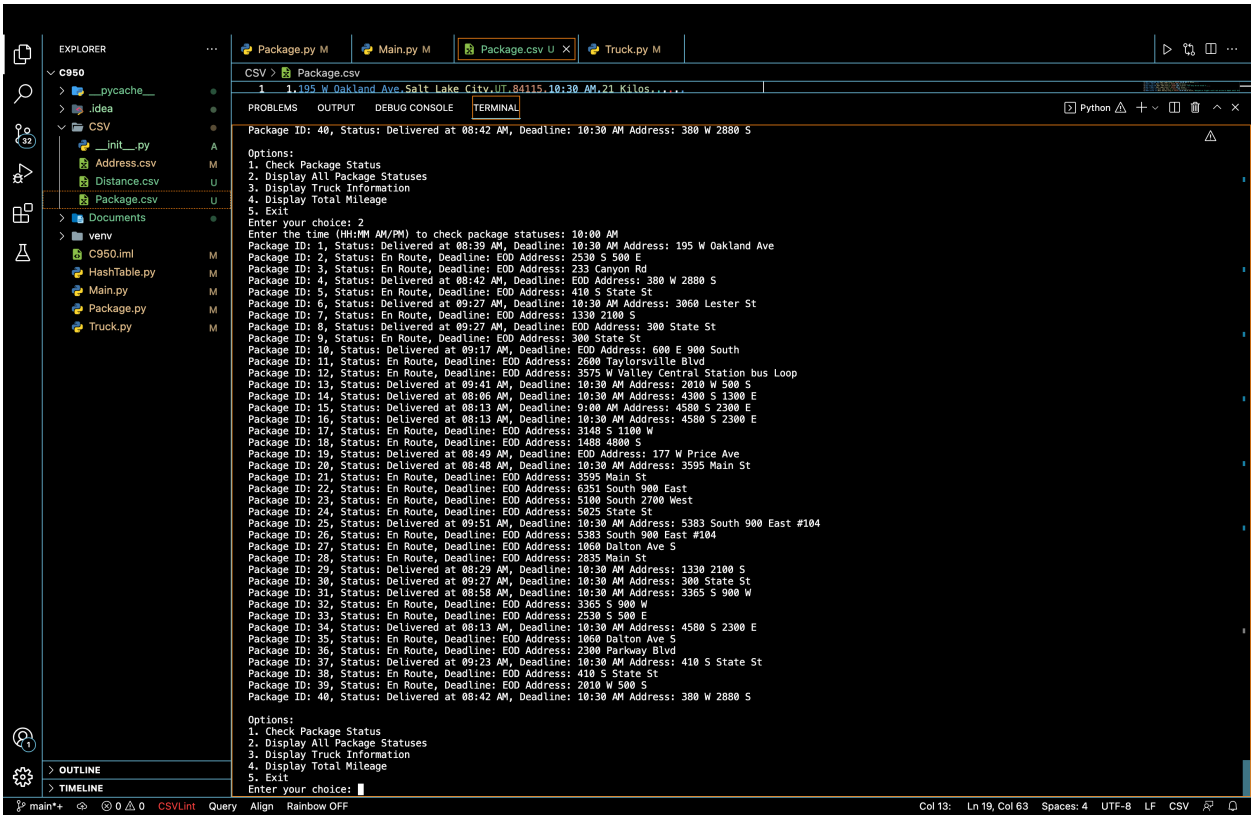
```python
def calculate_package_delivery_times(trucks, distance_matrix):
    """Calculate the delivery times for all packages."""
    for truck in trucks: # Iterate over each 'truck' in 'trucks list
        current_time = truck.current_time # Set 'current_time'
        current_location = 0 # Set 'current_location' to hub
        for package in truck.packages: # Initiate nested loop that iterates over each 'package' in 'truck'
            package_location = package.address_index # Retrieve 'address_index' and set as 'package_location'
            travel_time = calculate_delivery_time(distance_matrix[current_location][package_location]) # Calculate travel time to package
            current_time += timedelta(minutes=travel_time) # Add 'travel_time' to 'current_time'
            package.delivery_time = current_time # Set package's delivery time
            current_location = package_location # Update current location

def calculate_delivery_time(distance):
    """Calculate delivery time in minutes given a distance."""
    speed = 18  # Truck speed in miles per hour
    return (distance / speed) * 60 # Calculate and return delivery time

def display_package_status_at_time(hash_table, specific_time):
    """Display the status of all packages at a specific time."""
    for package_id in range(1, 41): # Iterate through package IDs
        package = hash_table.lookup(package_id) # Look up 'package-id', set as 'package'
        if package: # Check if package is found
            status = "At Hub" # Initial status assignment
            if package.delivery_time: # Check for delivery time
                if package.delivery_time <= specific_time: # If 'delivery_time' is less or equal to 'specific-time'
                    status = f"Delivered at {package.delivery_time.strftime('%I:%M %p')}" # Update status
                else: # If 'delivery_time' is greater than 'specific_time'
                    status = "En Route" # Set status
            address = package.address # Retrieve address
            deadline = package.deadline
            print(f"Package ID: {package_id}, Status: {status}, Deadline: {deadline} Address: {address}") # Print package information
        else: # If no package is found
            print(f"Package ID {package_id} not found.") # Print not found message

def main():
    """Main function to execute the program."""
    package_hash_table = HashTable() # Create hash table instance
    address_index_map = load_address_data(CSV_DIRECTORY + 'Address.csv') # Load address data
    load_package_data(CSV_DIRECTORY + 'Package.csv', package_hash_table, address_index_map) # Load package data
    distance_matrix = load_distance_data(CSV_DIRECTORY + 'Distance.csv') # Load distance data
    """print("Complete Distance Matrix:")
    for row in distance_matrix:
        print(row)"""
    truck1, truck2, truck3 = Truck(1), Truck(2), Truck(3) # Create truck instances
    trucks = [truck1, truck2, truck3] # Create truck list
    load_packages_onto_trucks(trucks, package_hash_table) # Load packages onto trucks
    calculate_package_delivery_times(trucks, distance_matrix) # Calculate package delivery times
```

D: Interface

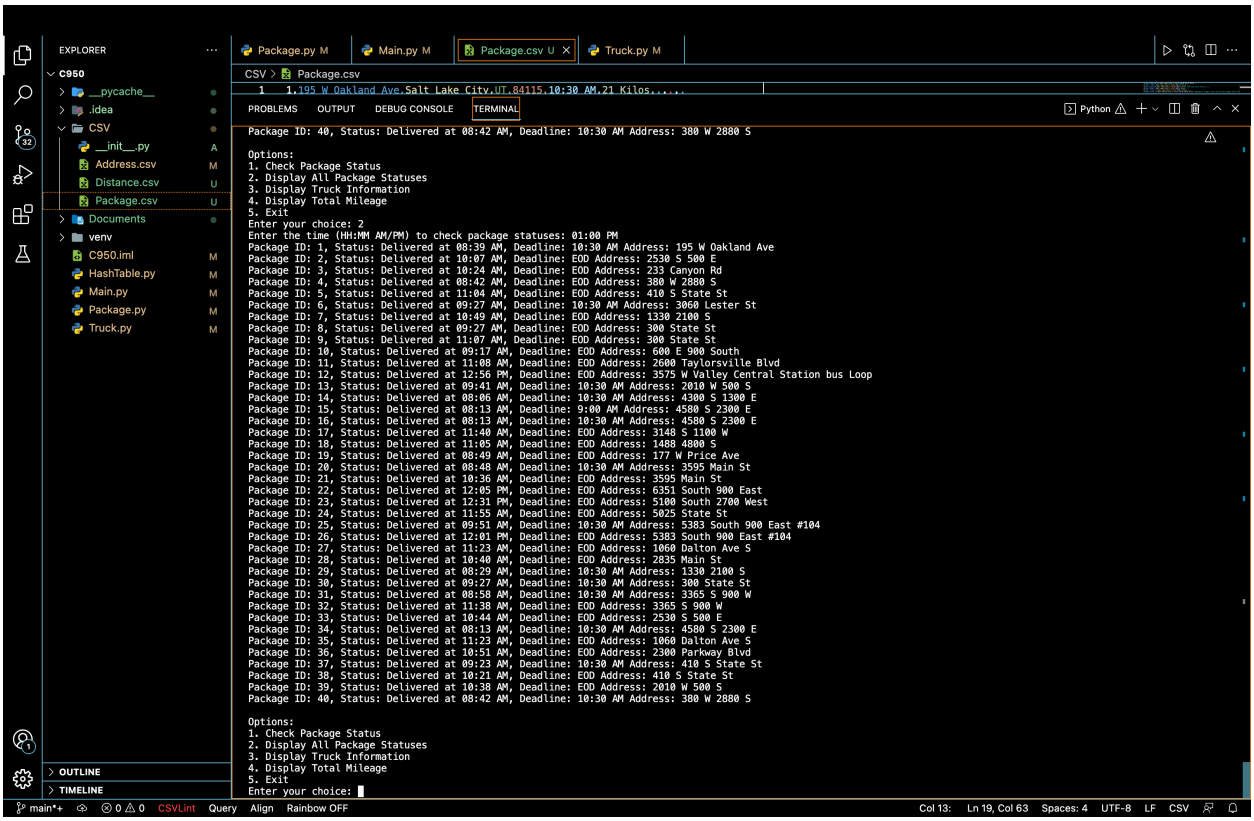
D1: First Status Check



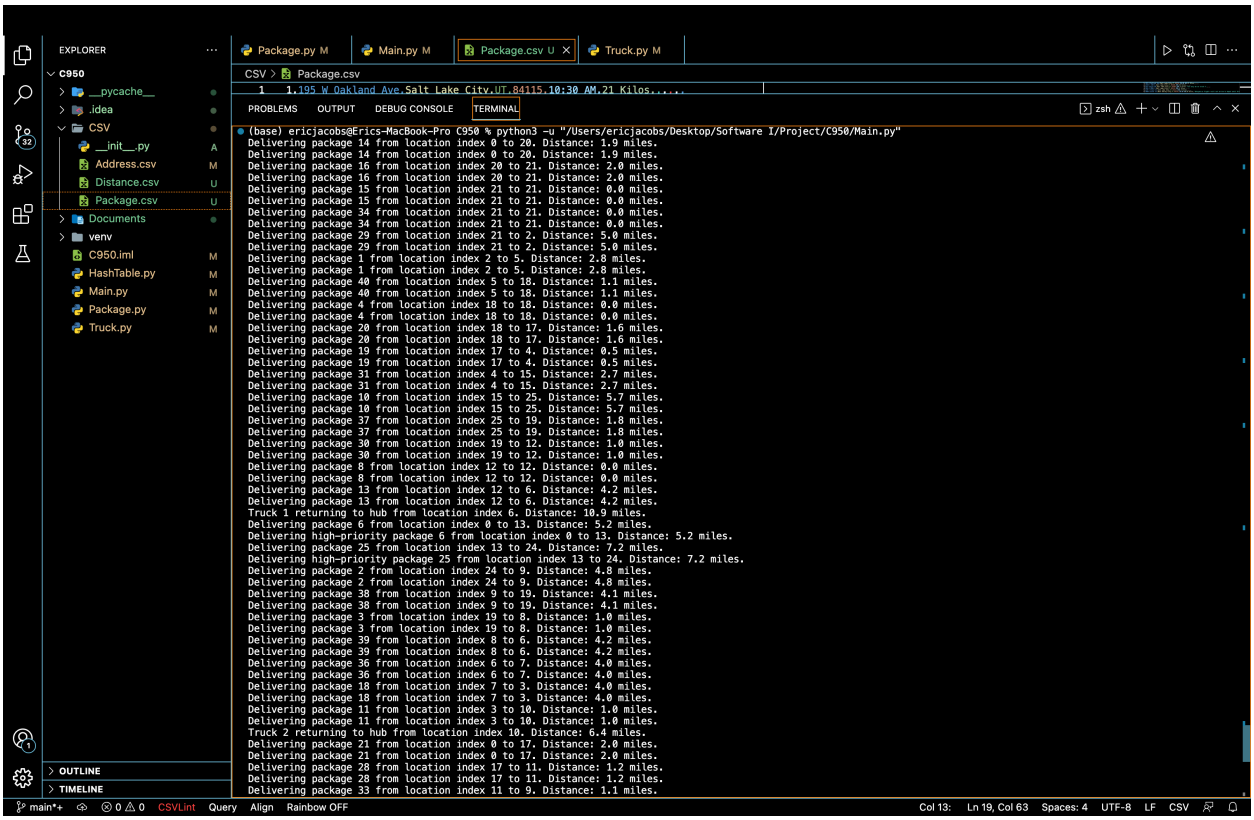
D2: Second Status Check

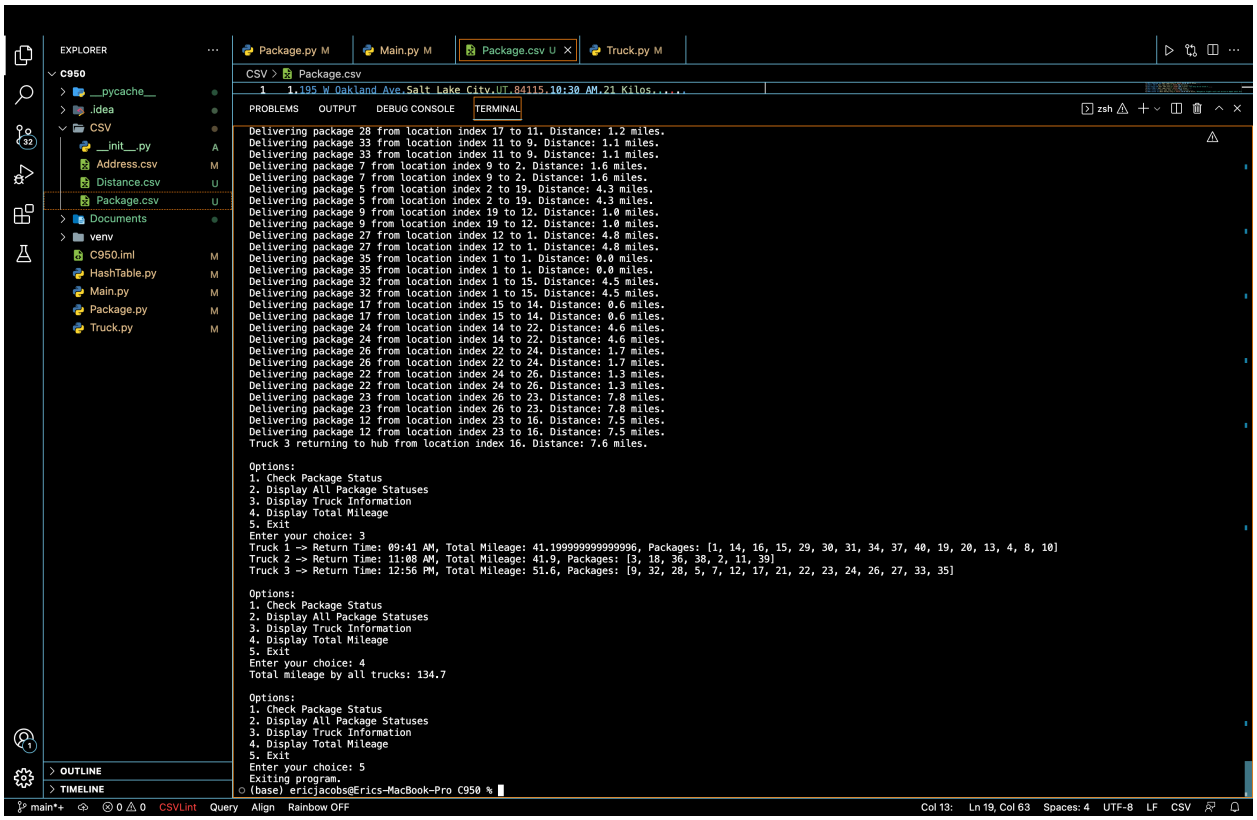


D3: Third Status Check



E. Screenshot of Code Execution





F1: Strengths of Chosen Algorithm

Simplicity and Speed:
The Nearest Neighbor algorithm's primary strength lies in its simplicity, making it easy to implement and understand. It requires no complex data structures or intricate heuristics, making the development and debugging processes more straightforward. This simplicity also translates to speed, as the algorithm can quickly make delivery decisions, allowing for rapid progression through the delivery route without the need for extensive pre-computation or planning.

Local Optimization and Real-Time Decision Making:
Another significant advantage of the Nearest Neighbor algorithm is its ability to make real-time, locally optimized decisions. By always selecting the closest next destination, it minimizes the travel time between stops, potentially reducing the overall delivery time for packages. This local optimization is particularly effective when deliveries are clustered in certain areas, as it allows the algorithm to minimize travel distance incrementally.

Low Overhead:
The Nearest Neighbor algorithm operates with low overhead, as it does not require storing or managing large sets of possible routes. Instead, it dynamically selects the next step based on current conditions. This is advantageous for systems with limited computational resources or when the delivery parameters can change rapidly, such as in real-time traffic conditions or when dealing with last-minute delivery schedule changes.

In the context of the WGUPS Routing Program, these strengths align with the operational requirements of quick dispatch and the ability to adapt to changes throughout the delivery day. The decision to use the Nearest Neighbor algorithm is also justified by the scale of the operation, where the relatively small number of destinations does not necessitate the overhead of more complex algorithms that may offer global optimization but at the cost of increased complexity and computation time.

F2: Verification of Algorithm

Effective Route Optimization:
The Nearest Neighbor algorithm adeptly aligns with the WGUPS routing program's need for efficient route planning. By prioritizing the closest unvisited locations for each delivery, this algorithm streamlines the delivery process, potentially reducing the overall time taken between consecutive deliveries.

Compatibility with Non-Linear Data Structures:
This algorithm seamlessly integrates with non-linear data structures like hash tables used in the program. The hash table effectively manages package data, including tracking and updating delivery statuses. The Nearest Neighbor algorithm leverages this data to make informed decisions on routing, thereby enhancing its efficiency.

Adherence to Operational Constraints: The Nearest Neighbor algorithm operates within the key constraints of the delivery system. It respects the truck capacity limits, ensuring that each truck's package load remains within the permissible 16-package maximum. Moreover, it aids in adhering to delivery deadlines by selecting the most immediately accessible destinations, thus minimizing the risk of delays.

Simple and Efficient Implementation: The algorithm’s simplicity makes it a practical choice for the WGUPS routing program. Its straightforward implementation ensures that the system remains robust and less prone to errors, which is critical in a real-world logistics operation.

Scalability and Flexibility: Despite its simplicity, the Nearest Neighbor algorithm offers a degree of scalability and flexibility. It can adapt to varying numbers of packages and different route configurations, making it a versatile tool for daily delivery operations.

Local Optimization for Quick Decisions: The algorithm's approach to local optimization, focusing on the immediate best choice without overcomplicating the overall route, enables quick and efficient decision-making. This aspect is particularly beneficial in dynamic delivery environments where time efficiency is paramount.

Facilitates Real-Time Adjustments: Given the real-time nature of package delivery, the Nearest Neighbor algorithm is apt as it allows for on-the-go route adjustments. This adaptability is crucial for handling unforeseen changes in delivery schedules or routes.

In summary, the Nearest Neighbor algorithm's use in the WGUPS routing program satisfies the scenario's requirements. Its ability to efficiently plan routes, work in tandem with complex data structures, and operate within the given logistical constraints, all while maintaining simplicity and adaptability, makes it a well-suited choice for the task at hand.

F3: Other Possible Algorithms

Dijkstra's Algorithm:
Dijkstra's algorithm excels in mapping out the shortest path from one origin to various destinations, which is beneficial for optimizing the route for a single truck from the hub to multiple delivery points. Unlike the Nearest Neighbor algorithm, which selects the next closest destination without considering future paths, Dijkstra's algorithm takes a broader view of the network, potentially finding a shorter overall path. This could result in substantial mileage savings, especially in a complex urban delivery network. For the given program, I prioritize simplicity and real-time decision-making, which Nearest Neighbor provides, over the complete route optimization that Dijkstra's algorithm offers.

Dynamic Programming:
Dynamic programming approach would analyze all possible delivery sequences and choose the one with the optimal outcome, considering every package's constraints and delivery windows. This method is substantially different from the Nearest Neighbor algorithm, which does not consider future deliveries when choosing the next destination. Dynamic programming could potentially optimize the entire delivery day, but at the cost of increased initial computation time. Given that the package delivery system requires quick adjustments and real-time tracking, the Nearest Neighbor algorithm's simplicity and speed are more aligned with operational needs than the comprehensive but slower dynamic programming approach.

F3a: Algorithm Differences

Dijkstra's Algorithm:
Dijkstra's Algorithm differs from the Nearest Neighbor approach primarily in its route optimization and computational complexity. Dijkstra's focuses on calculating the shortest possible route from a single point to all others in the network, considering the overall path. This comprehensive view potentially yields a more efficient total route, but at the cost of higher computational demands. Unlike Nearest Neighbor, which selects the next closest point without considering the total journey, Dijkstra's Algorithm ensures a globally optimized path. However, its complexity makes it less adaptable to real-time changes, as any alterations in the delivery plan might necessitate recalculating the entire route.

Dynamic Programming:
Dynamic Programming stands apart from the Nearest Neighbor method in its optimization scope and approach to computational complexity. It breaks down the delivery routing problem into smaller sub-problems, optimizing the entire route by considering all constraints and future decisions. This method potentially offers a more efficient solution for the entire delivery schedule but requires significant computational resources upfront. Unlike the Nearest Neighbor approach, which makes decisions based solely on immediate proximity, Dynamic Programming's holistic approach is less adaptable to last-minute changes, as it involves pre-computing the entire delivery plan and struggles to incorporate real-time adjustments.

G: Different Approach

Enhanced User Interface for Monitoring and Updates:
A more sophisticated user interface would significantly improve the user experience. Implementing a dashboard with real-time tracking, alerts for delivery updates, and interactive maps would allow users and supervisors to monitor progress effectively and make informed decisions.

Integration of Machine Learning for Predictive Analytics:
Machine learning models could be developed to predict traffic patterns, estimate delivery times more accurately, and suggest route alterations in response to unforeseen circumstances, such as road closures or delivery delays.

Use of API Integration for Real-Time Data: Integrating APIs from traffic and weather services could provide real-time data that can dynamically adjust routes to avoid delays, thereby improving punctuality and customer satisfaction.

H: Verification of Data Structure

Efficient Access to Package Data:
The hash table structure in the WGUPS Routing Program excels in providing fast access to package information. Each package is uniquely identified by its ID, serving as the key in the hash table. This structure enables constant-time complexity (O(1)) for critical operations like inserting, retrieving, and updating package data, which is pivotal in a dynamic delivery environment.

Flexibility in Data Management: The hash table's design allows for flexibility in handling package data. For instance, when the address for package #9 needed updating at 10:20 AM, the hash table efficiently facilitated this change. This adaptability is essential for real-world scenarios where package information can change due to various factors.

Handling Multiple Package Attributes: The hash table effectively stores and manages various attributes for each package, such as delivery address, deadline, and status. This comprehensive data handling ensures that all necessary details are readily available for each package, enabling informed decision-making during the delivery process.

Compatibility with Routing Algorithm: The integration of the hash table with the Nearest Neighbor algorithm demonstrates a harmonious relationship between data storage and routing logic. The hash table provides the algorithm with quick access to package locations and statuses, aiding in efficient route calculation and package prioritization.

Scalability and Maintenance: The hash table's structure supports scalability, allowing the system to handle a growing number of packages without significant performance degradation. Moreover, the simplicity and clarity of the hash table's implementation contribute to easier maintenance and potential future enhancements of the program.

Resilience to Dynamic Changes: The hash table's ability to quickly adapt to changes, such as updating delivery statuses from 'At the hub' to 'En route' or 'Delivered', showcases its suitability for dynamic operational environments like package delivery, where statuses are frequently updated throughout the day.

In conclusion, the hash table used in the WGUPS Routing Program fulfills all the requirements set out in the scenario. Its efficiency in data retrieval and modification, combined with the ability to handle complex data structures and integrate seamlessly with the routing algorithm, makes it an ideal choice for this application.

H1: Other Data Structures

Graphs:
A graph data structure consists of nodes (vertices) and edges, which naturally represent networks such as city maps and delivery routes, making it an ideal choice for routing problems. In a graph, each delivery location can be a node, with edges representing the direct paths between them, along with the distances as weights. This allows for the use of graph-specific algorithms like Dijkstra's or A* for finding the shortest path, which could potentially be more efficient than the nearest neighbor approach when dealing with complex routing scenarios. Graphs also facilitate the representation of various constraints and conditions of the delivery scenario, such as time windows, package dependencies, and truck capacities. Unlike hash tables, which are excellent for key-value pair storage and retrieval, graphs excel in modeling relational data and spatial structures, providing a more holistic view of the relationships between different delivery points.

Binary Search Trees (BSTs):
A binary search tree is a hierarchical data structure where each node has at most two children, referred to as the left child and the right child. BSTs are sorted in a way that allows for efficient searching, insertion, and deletion operations that, on average, have a time complexity of O(log n). However, in the worst case, such as when the tree becomes unbalanced, these operations can degrade to O(n). In the context of package delivery, a BST could be used to organize packages by delivery deadlines or other sortable criteria, facilitating an ordered traversal that could assist in prioritizing deliveries. However, unlike hash tables which provide rapid access to package data via direct hashing, BSTs would require traversal from the root to the appropriate node, which can be less efficient for lookups. Balancing a BST to ensure its efficiency, such as in AVL trees or Red-Black trees, adds additional complexity to the implementation, which is not required for hash tables.

H1a: Data Structure Differences

Graphs:
While graphs offer a visual and relational representation of the delivery network, the complexity of graph algorithms and the overhead of maintaining such structures may not be necessary for smaller scale routing problems. In the program, I opted for hash tables over graphs because they provide faster direct access for package data retrieval, which is crucial for real-time updates on package status, and are simpler to implement for the specific use case that does not require visual mapping of routes.

Binary Search Trees (BSTs):
Although BSTs can maintain an ordered structure of packages, their performance can become inefficient in skewed or unbalanced scenarios, requiring rebalancing algorithms to maintain optimal search times. I chose hash tables in the program because they offer consistent average-time complexity for insertions, deletions, and lookups, which is more suitable for the dynamic nature of package tracking and doesn't necessitate the ordered structure that BSTs provide.