C950 Task-2 WGUPS Write-Up

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

5/18/2025

C950 Data Structures and Algorithms II

Table of Contents

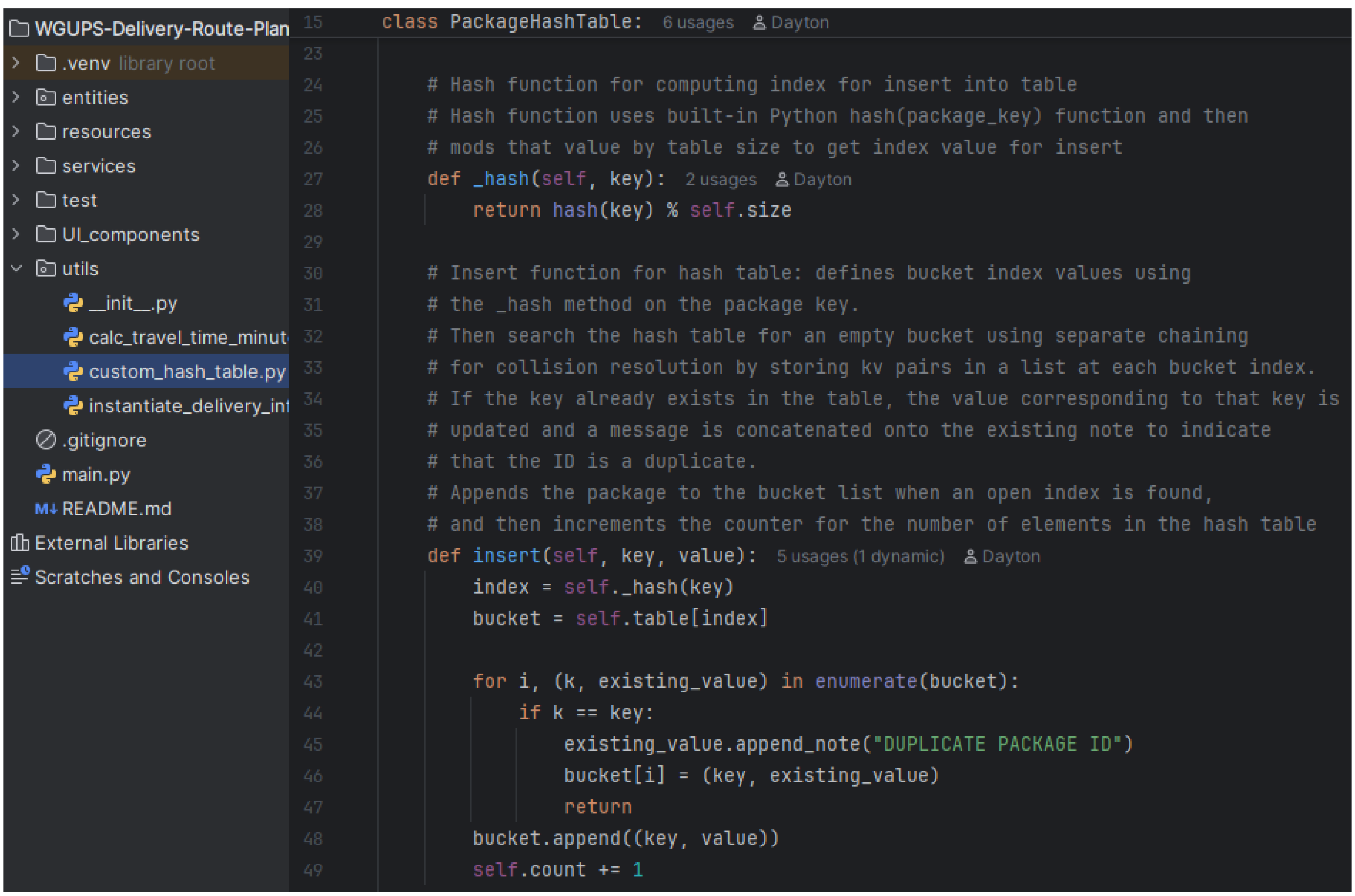
Table of Contents

A. Hash Table

The self-adjusting hash table is defined in the file: *WGUPS-Delivery-Route-Planning/utils/custom_hash_table.py*. The hash table consists of a constructor for initializing the hash table to a size that is relative to the input size (i.e., number of packages). The hash table constructor makes use of another function from within the PackageHashTable file that is not a method of the class. This function, _increment_for_prime, takes the number of packages, multiplies by two, and increments this value to the next greater prime. This self-adjusting approach for determining the size of the hash table ensures that the table is large enough to contain all of the package data without encountering collisions upon insertion into the table.

```
# defining static method to generate the next prime that is
# greater than argument 'num'
def _increment_for_prime(num):  1 usage  new *
    def is_prime(x):  new *
        if x <= 1:
            return False
        for i in range(2, x):
            if x % i == 0:
                return False
        return True
    while not is_prime(num):
        num += 1
    return num

class PackageHashTable:  6 usages  Dayton

    # The constructor for hash table to determine size and create table
    # Also, adds a count attribute that is incremented with each insertion
    def __init__(self, num_packages):  Dayton
        self.size = _increment_for_prime(num_packages * 2)
        self.table = [[] for _ in range(self.size)]
        self.count = 0
```

The PackageHashTable includes an insert method for inserting packages into the hash table. This method includes input parameters for the package ID as a key and the other package data as the corresponding value. In order to calculate the bucket placement for a particular package in the table, the built-in Python hash() function is called on the package key (package_id). The corresponding bucket for the insertion is then computed by taking the remainder of the hashed key value from dividing the hash value by the size of the hash table. The insert method itself also has a conditional block to check if there is a collision on insert, however, the hash table does not implement a collision resolution strategy. The conditional block here merely updates the package to the new insert value if there is a collision and appends a note to the package *notes*, which states that the package ID is a duplicate.



B. Look-Up Functions

The custom hash table includes two lookup methods that return package objects and all associated data for the package. The first of these methods takes a package ID as an argument, and the second takes an address as an argument. The *get_by_id* method takes the provided package ID, hashes it in the same way as the insertion method, and searches the hash table for this index. The method *get_by_id* then returns the value corresponding to that key if it is found, otherwise it returns *None* type if it is not found.

```
# get method for retrieving hash table elements' values by their key
def get_by_id(self, key):  18 usages (18 dynamic)  Dayton
    index = self._hash(key)
    for k, v in self.table[index]:
        if k == key:
            return v
    return None
```

The *get_by_address* method takes an *address* string as the input, initializes an empty list for packages, and searches all packages in the hash table for any packages with an address that contains the string value provided as the argument. For all packages that match this provided address, they are appended to the package_match list, which is returned by this method after it searches all buckets in the hash table. I created this additional look-up method to simplify the process of grouping packages that have same delivery address into the same delivery batch, so long as there are no other package constraints that prevent this.

```
    def get_by_address(self, address):  4 usages (4 dynamic)  Dayton
        package_match = []
        for bucket in self.table:
            for k, v in bucket:
                if address in v.address:
                    package_match.append(v)
        return package_match
```

C1. Identification Information

Project

- WGUPS-Delivery-Route-Planni
  - .venv library root
  - entities
  - resources
  - services
    - batch_truck_load_servic
    - delivery_service.py
    - distance_matrix_builder.
    - nearest_neighbor_path_
    - package_data_parser.py
    - package_priority_parsing
  - UI_components
  - utils
  - .gitignore
  - main.py
  - README.md
- External Libraries
- Scratches and Consoles

main.py

```python
# Dayton Abbott
# 011125353
line_format = ""

def get_fresh_routes():  1 usage  Dayton
    instantiate_delivery_infra()
    routes = start_delivery_service(package_keys, package_hash_table)
    return routes

def main():  1 usage  Dayton

    exit_delivery_monitor = False

    while not exit_delivery_monitor:

        main_menu()
        routes = get_fresh_routes()

        user_input = input()

        # Print all package status and mileage
        # when deliveries are completed
        if user_input == '1':
            query_delivery_service( time: "6:00 PM", routes, int(user_input))

        # Print single package info by ID
        elif user_input == '2':
            prompt_for_package_id()

        # Get single package status at a given time
        elif user_input == '3':
            user_time = input("Enter time in format: HH:MM AM/PM\n (space between time and AM/PM)\n")
            query_delivery_service(user_time, routes,  condition_code: 3)

        # Get all package statuses at given time
        elif user_input == '4':
            user_time = input("Enter time in format: HH:MM:AM/PM\n (space between time and AM/PM)\n")
            query_delivery_service(user_time, routes,  condition_code: 4)

        # print route data based on truck number
        elif user_input == '5':
            route_number = int(input("Enter route number (1-3)\n"))
            print(Truck.trucks_dict[route_number])
            print_route_data(routes, route_number - 1)

        # Exit program
        elif user_input == '6':
            exit_delivery_monitor = True

        else:
            print("Invalid input. Please try again.")

if __name__ == '__main__':
    main()
```

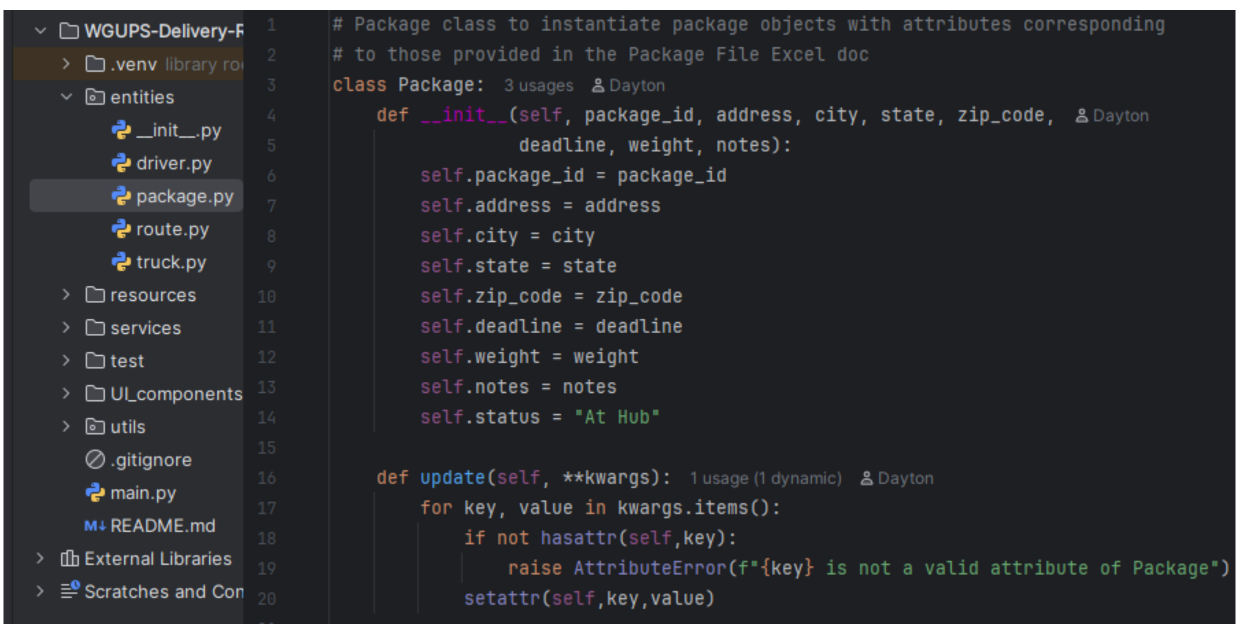
C2. Original Code with Process and Flow

For my implementation of the WGUPS route planning project, I modularized the architecture for clean organization and scalability. The core components of the application are divided into five modules: *entities*, *resources*, *services*, *UI_components*, and *utils*.

The entities package contains class definitions for *Driver*, *Package*, *Route*, and *Truck*. I will not go into detail on the *Driver* class here because it is not very important.

C2. Entities Module: Package

The *Package* class provides a means of instantiating package objects composed of the data extracted from the excel documents provided, as well as an additional field for the package status to indicate delivery status. The *Package* class also includes an *update* method that I created for correcting package number 9's address while the delivery service is running, a *__str__* method for formatting package print calls, and a method for appending to the notes attribute. The calls to *append_note* could have been modified to use the *update* method; I created them at different points and did not realize I was going to need a general update method while I was creating the append_note method. It did not seem worthwhile to refactor this since it works and *append_note* does serve a distinct purpose in its own right, which is to provide a notice when hash table inserts collide.



Package class continued:

```python
    # method for stringifying a Package object and adding min fields widths
    # for improved readability
    def __str__(self):
        package_label = f"Package {self.package_id}:"
        return (f"{package_label:<14} {self.status:<30} {self.address:50} "
                f"{self.city:20} {self.state:8} {self.zip_code:8} "
                f"{self.deadline:10} {self.weight:8}kg {self.notes:^65}")

    # method used in the custom_hash_table insert method to concat a duplicate
    # message to the package's note if the package ID has a duplicate attempt
    # to insert it into the hash table
    def append_note(self, new_note):
        if self.notes:
            self.notes = self.notes + "; " + new_note
```

C2. Entities Module: Route

The *Route* class contains all major route data for finalized routes, including: the *package_keys* for all packages in a given route, the package hash table of all packages in the route, total number of destinations in the route, dynamically generated distance matrices based on the route's hash table, the optimized sequence of delivery destinations and corresponding distances, total route distance, and route duration. I

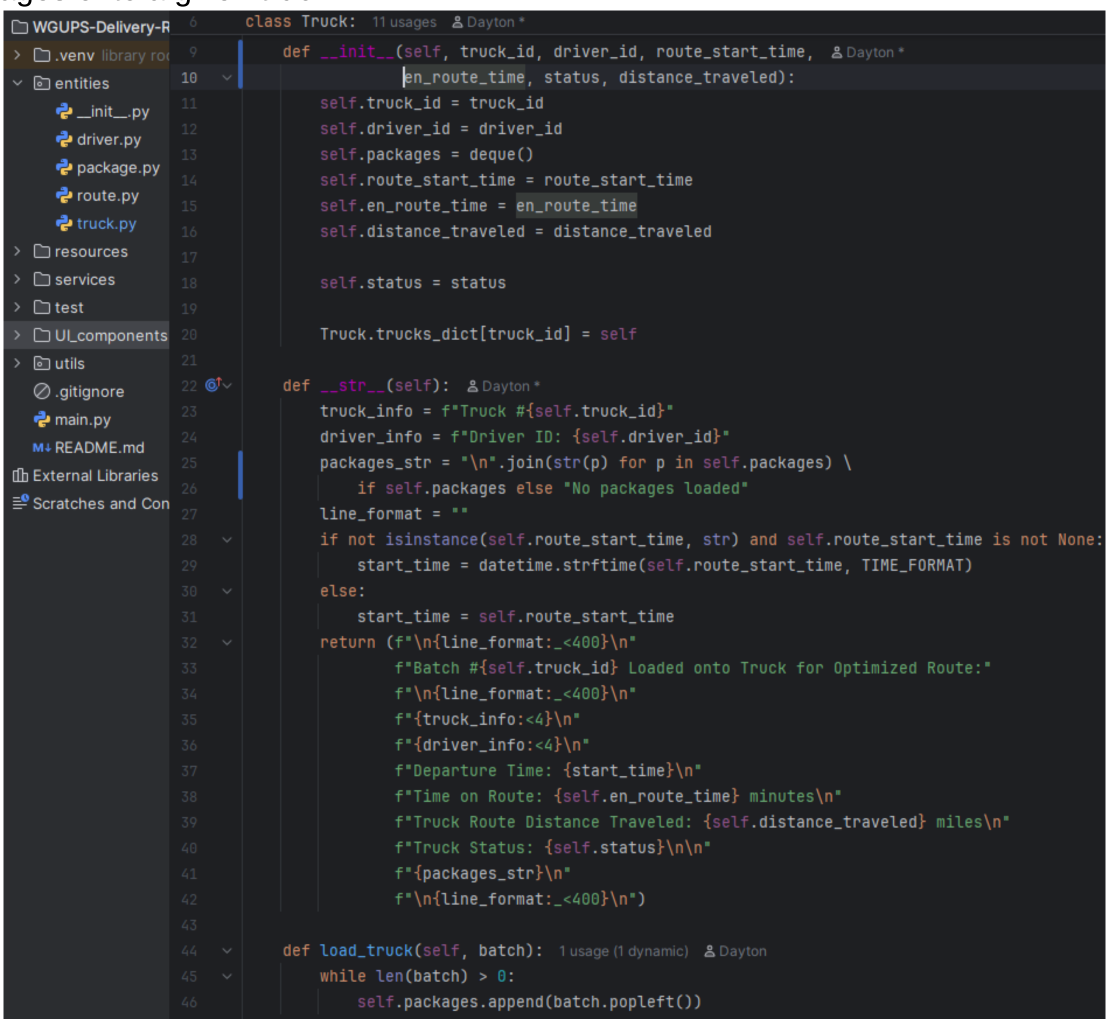
- driver.py
- package.py
- route.py
- truck.py
- resources
- services
- test
- UI_components
- utils
- .gitignore
- main.py
- README.md
- External Libraries
- Scratches and Consoles

```python
    def __init__(self, label=None, package_keys=None, package_table=None, num_destinations=None,
                 distance_matrix=None, metadata=None, total_distance=None, duration=None):
        self.label = label
        self.package_keys = package_keys
        self.package_table = package_table
        self.num_destinations = num_destinations
        self.distance_matrix = distance_matrix
        self.metadata = metadata
        self.total_distance = total_distance
        self.duration = duration

    def __str__(self):
        package_table = print_all_packages(self.package_keys, self.package_table, self.label)
        num_destinations = len(self.distance_matrix[0])
        return (f"{line_format:_<400}\n \n{package_table}\n"
                f"{print_dist_matrix(self.distance_matrix, num_destinations)}"
                f"\n\n"
                f"{line_format:_<400}"
                f"\n"
                f"NEAREST NEIGHBOR OPTIMIZED ROUTE DESTINATION DETAILS"
                f"\n"
                f"{print_route_metadata(self.metadata, self.num_destinations)}\n"
                f"Route Distance Total: {self.total_distance} miles\n"
                f"Route Duration: {self.duration} minutes\n")
```

created the *Route* class to simplify the access of data for a particular route given that different data elements are requested for different queries made in the UI.

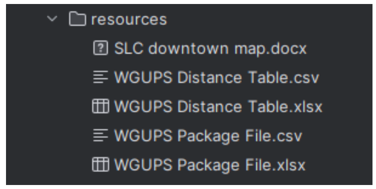
C2. Entities Module: Truck

The *Truck* class includes relevant status data with respect to queries that a user can make in the UI, data such as: *driver_id*, *packages*, *route_start_time*, *en_route_time*, *distance_traveled*, and *status*. Each of these attributes was created for extracting status data for presentation through the UI. For example, *route_start_time*, *en_route_time*, and *distance_traveled* are all used in determining the status of the truck itself when a user provides a time for the service to be queried, as well as the status of the packages contained on the truck. The *Truck* class also includes a *load* method for loading packages onto a given truck.



C2. Resources module

The *resources* module within the root folder for my project contains the materials provided for the project, as well as the .csv versions of these files.

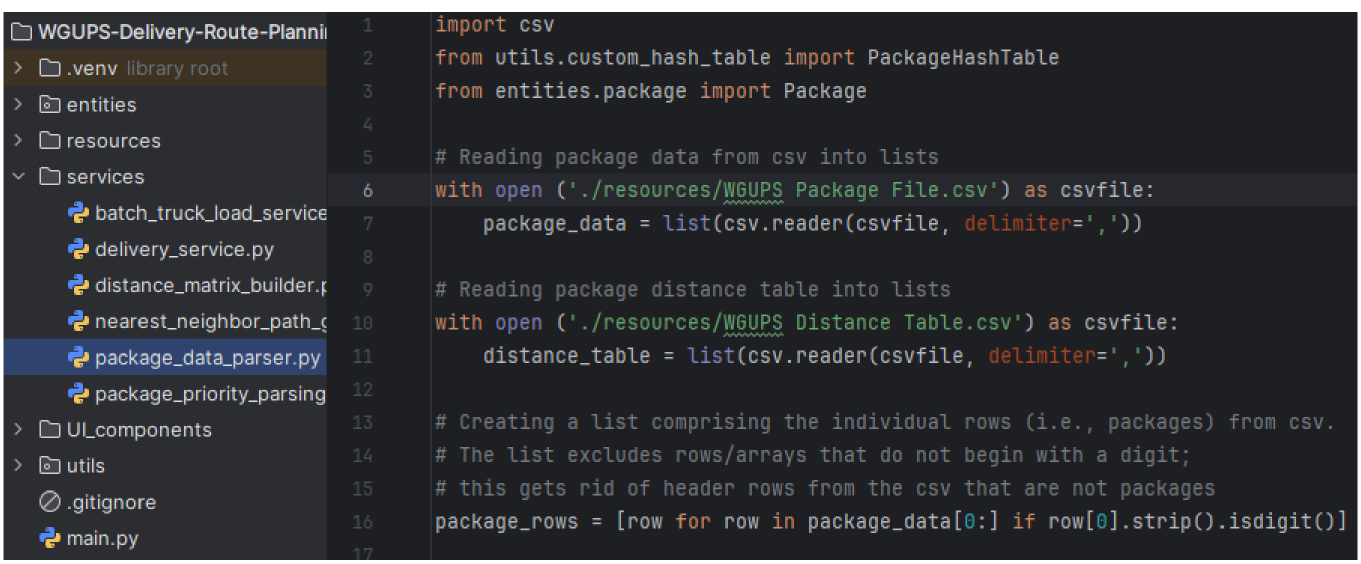


The *service* module contains distinct services for: extracting the provided data and inserting into operable data structures, determining package priorities, building distances matrices for individual routes, generating nearest neighbor paths for the delivery trucks, loading the packages onto the trucks, and executing the delivery service.

C2. Data import and organization

This process begins with the *package_data_parser.py* file, which takes the .csv files for package data and distance table, extracts the data, and inserts it into data structures that are used for access throughout the program. The package data is inserted into the package_hash_table, and the distance table data is used to construct a distance matrix for comparing path weights between destinations. The code here is explained within the comments.

C2. Hash table and distance table imports




```python
import csv
from utils.custom_hash_table import PackageHashTable
from entities.package import Package

# Reading package data from csv into lists
with open ('./resources/WGUPS Package File.csv') as csvfile:
    package_data = list(csv.reader(csvfile, delimiter=','))

# Reading package distance table into lists
with open ('./resources/WGUPS Distance Table.csv') as csvfile:
    distance_table = list(csv.reader(csvfile, delimiter=','))

# Creating a list comprising the individual rows (i.e., packages) from csv.
# The list excludes rows/arrays that do not begin with a digit;
# this gets rid of header rows from the csv that are not packages
package_rows = [row for row in package_data[0:] if row[0].strip().isdigit()]
```

C2. Building the hash table and distance matrix

```python
# Extracting only the rows that represent recipients from the distance table
# and placing them in an array for mapping to the matrix.
distance_table_rows = []
for row in distance_table[0:]:
    try:
        # if the value at index position 2 (distance from hub) of the row cannot be
        # converted to a float, then the row is not a recipient row and should not
        # be included in the recipient count
        float(row[2])

        # Adding each recipient row to the distance_table_rows to reconstruct the
        # recipient table, but with only the rows representing recipients
        distance_table_rows.append(row)

    # catching errors from the attempt to convert to float from above and continuing,
    # i.e. skipping that row if float conversion throws an error
    except(ValueError, TypeError):
        continue


# creating a variable that holds the number of recipients + 1 for defining matrix size
# the + 1 is required because there is one additional row and column required for the
# recipient names/identifiers
num_destinations = len(distance_table_rows) + 1


# Creating hash table using custom PackageHashTable class and _next_prime method
# to create a table that is of size (num packages in csv * 2) ++ to next prime)).
package_hash_table = PackageHashTable(len(package_rows))

distance_matrix = [[0 for _ in range(num_destinations)] for _ in range(num_destinations)]
```

C2. Populating hash table

```
# Creating a list of package keys for testing the package_hash_table population.
package_keys = []

# Looping through each row array and extracting data by array index position
# to assign it to the corresponding package attribute.
# Also, adding each package_id to package_keys so that I can test that the hash table
# has populated correctly later (it's in test_prints.py).
for row in package_rows:
    package_id = int(row[0])
    package_keys.append(package_id)
    address = row[1]
    city = row[2]
    state = row[3]
    zip_code = row[4]
    deadline = row[5]
    weight = float(row[6])
    notes = row[7]

# Creating a temp package from the attributes assigned in each row...
    temp_package = Package(package_id, address, city, state, zip_code, deadline, weight, notes)

# so that the temp package can be inserted into the package_hash_table
    package_hash_table.insert(package_id, temp_package)
```

C2. Building distance matrix continued

```
# Constructing a two-dimensional array/matrix from the distance table csv.
index = 1
distance_matrix[0][0] = 'RECIPIENT'
for row in distance_table_rows:

    # Parsing address/recipient/zipcode identifiers from csv rows and extracting the recipient name
    recipient = row[0].split('\n')
    recipient = recipient[1].strip()
    recipient = recipient.strip(',')

    # Extracting initial letters of recipient info to generate a string abbreviation for the recipient
    # name to make the top row of the matrix readable and allow column values to line up visually with
    # their corresponding distance values
    recipient_abrev = recipient[0:24]

    # Inserting abbreviated recipient names in corresponding top row index positions
    distance_matrix[0][index] = recipient_abrev

    # Inserting non-abbreviated recipient names along the corresponding rows in index 0
    distance_matrix[index][0] = recipient_abrev
```

C2. Populating the distance matrix

```python
    # Mapping distance values from the csv to the correct location within the matrix
    for col in range(1, num_destinations):
        # If the value cannot be converted to a float, then it is not a distance value.
        # So, I use a try/catch block to try to convert each value to a float.
        try:
            distance_matrix[index][col] = 'X'
            # If this throws an error, then the value is not a distance value and retains
            # a placeholder "X" for improved readability.
            dist = float(row[col + 1])

            # if the value can be converted to a float, then it is added to the matrix in the position
            # that corresponds to its distance between two locations
            distance_matrix[index][col] = dist
            distance_matrix[col][index] = dist

        # Catching errors from the attempt to convert to float from above and continuing i.e., skipping
        # that element if float conversion throws an error.
        # The try block sets every index value to X before attempting the float conversion, so
        # if the value fails the float conversion it retains the placeholder X
        except(ValueError, TypeError):
            continue

    index += 1
```

C2. Prioritization, Batching, Optimization, and Loading

With the package and distance data now accessible in the program, the delivery service execution flow moves onto the *package_priority_parsing_service*. This service begins with creating a copy of the package data hash table. A copy of the package data hash table is used in order to allow *status* update operations to be performed within individual queries made from the UI without modifying the core data structure, which in turn enables the main() loop to loop repeatedly and the user to perform multiple subsequent queries without *status* updates carrying over from previous queries. Moving through the *package_priority_parsing_service*, this service utilizes many of the other services from within this module to perform each of the steps involved in the optimization and routing process, beginning with identifying package priorities based on package deadlines and other constraints.

C2. Prioritization

At a high level, my approach to this part of the problem was to identify three categories of packages, which I initially tracked using lists of the package IDs. The first category I labeled as the *constrained_packages* and included any package that has a *notes* value. These packages were assigned to truck number two, since many of the *notes* state that it is required for the package in question to be delivered by truck two. Also, these packages were all able to be delivered by their respective delivery times, despite delaying the start time of this route until the late packages arrived at 9:05am.

```
WGUPS-Delivery-Route-P
  .venv library root
  entities
  resources
  services
    batch_truck_load_se
    delivery_service.py
    distance_matrix_bui
    nearest_neighbor_p
    package_data_parse
    package_priority_pa
  UI_components
  utils
  .gitignore
  main.py
  README.md
External Libraries
Scratches and Consoles
```

```python
def package_priority_parsing_service(some_package_keys, some_package_hash_table):  2 usages  Dayton *

    instanced_package_hash_table = copy.deepcopy(some_package_hash_table)
    not_corrected = True

    # Create arrays for three separate package priority classifications.
    priority_package_keys = []
    constrained_package_keys = []
    standard_package_keys = []

    # Use package_id from package_keys to loop through all packages to determine
    # package priority classification.
    for package_id in some_package_keys:
        temp_package = instanced_package_hash_table.get_by_id(package_id)
        # if a package has any delivery note, then it is a constrained package.
        if temp_package.notes and not "Wrong address" in temp_package.notes:
            constrained_package_keys.append(package_id)
        # If a package has a delivery deadline, and the package is not already
        # in constrained packages, then it is assigned to the priority packages array.
        elif not temp_package.notes and not temp_package.deadline[0].isalpha():
            if package_id not in constrained_package_keys and package_id not in standard_package_keys:
                association_check = instanced_package_hash_table.get_by_id(package_id)
                associated_packages = instanced_package_hash_table.get_by_address(association_check.address)
                if len(priority_package_keys) + len(associated_packages) <= 16:
                    for package in associated_packages:
                        if package.notes:
                            continue
                        if (package.package_id not in priority_package_keys
                                and package.package_id not in standard_package_keys):
                            priority_package_keys.append(package.package_id)
```

The second category of packages are the *priority_packages*, these are the packages that have a specific delivery deadline, but <u>do not</u> have a *notes* value. These packages are able to be loaded at the start of the delivery service window and immediately sent out for delivery because the limitation of not having a *notes* value excludes any packages that are late or have incorrect addresses. Together, the lists of keys for these two categories are built as follows:

The third category of packages is *standard_packages*, which are the packages that have no deadline, no notes, or a note that states that the address is wrong. Because this batch of packages is larger than the others and cannot fit onto a single delivery truck when the truck capacity is 16 packages, these packages are dispersed between the other two trucks when the *standard_package_keys* list reaches a length of 16.

The package with the incorrect address is included with this batch because it would otherwise delay priority packages and cause missed delivery deadlines. After this prioritization process is run, package number 9 is updated to reflect the correct address. The project instructions state that we can “assume the address will be updated at

```
        # If a package is neither assigned to constrained packages nor priority packages,
        # then it is a standard package.
        else:
            if package_id not in constrained_package_keys and package_id not in priority_package_keys:
                # There are too many standard packages to fit on one truck,
                # so standard packages are assigned to standard package array, so long as the
                # number of packages in standard packages does not exceed truck capacity.
                # When the standard package array reaches maximum size, the remaining standard
                # packages are dispersed between priority and constrained package arrays
                if ((len(standard_package_keys) < 16) and (package_id not in priority_package_keys)
                        and (package_id not in constrained_package_keys)):
                    standard_package_keys.append(package_id)
                elif ((len(priority_package_keys) < 16) and (package_id not in priority_package_keys)
                      and (package_id not in constrained_package_keys)):
                    priority_package_keys.append(package_id)
                elif ((len(constrained_package_keys) < 16) and (package_id not in priority_package_keys)
                      and (package_id not in constrained_package_keys)):
                    constrained_package_keys.append(package_id)

    if not_corrected:
        package_to_correct = instanced_package_hash_table.get_by_id(9)
        package_to_correct.update(
            address="410 S State St",
            city="Salt Lake City",
            state="UT",
            zip_code="84111",
            deadline="EOD",
            weight=2,
            notes="Address corrected at 10:20 AM"
        )
        not_corrected = False
```

10:20”, which seems to suggest that this address can be corrected before 10:20. Correcting the address later would have just resulted in a less optimal delivery route for this batch of packages. The *standard_package* batch population and correction of package 9 were accomplished like so:

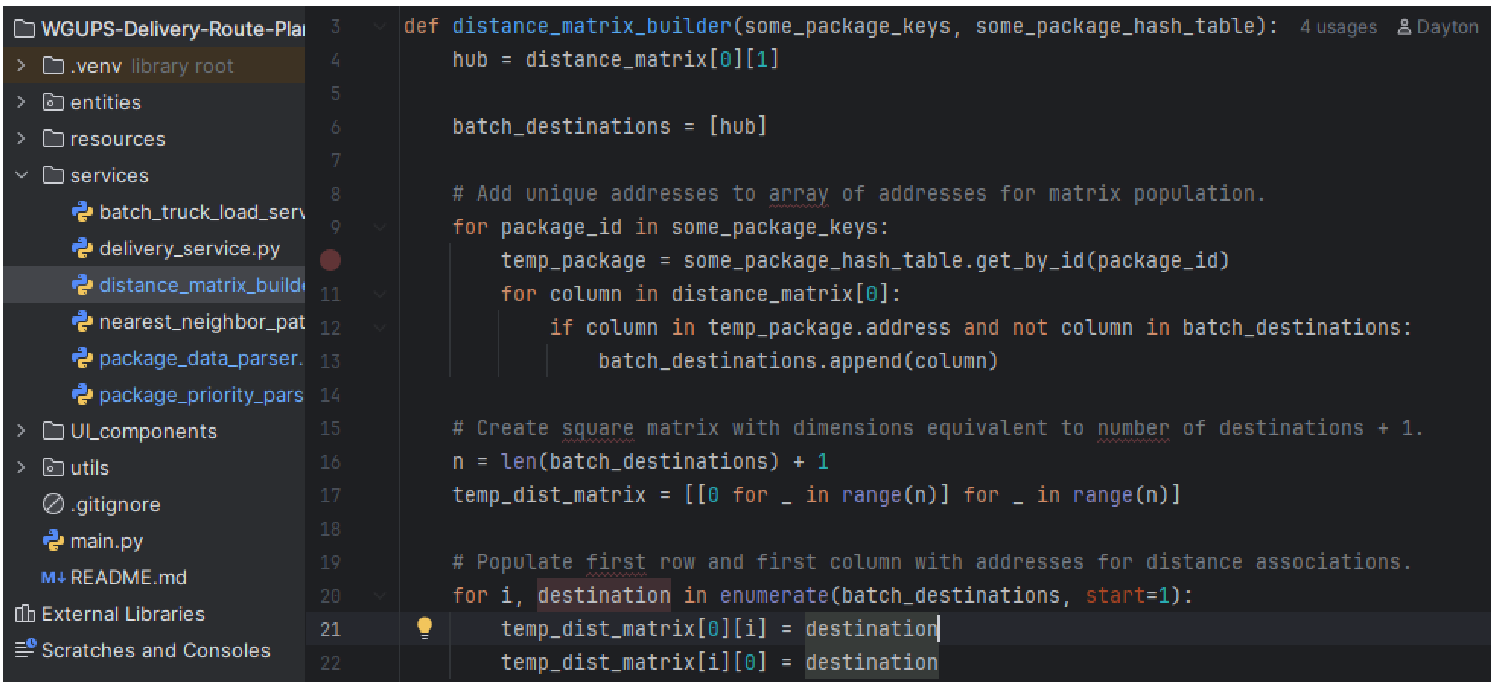
C2. Populating batch hash tables

Once the packages have been separated into their respective categories, hash tables for each batch can be generated based on the length of the list of keys for the batch, and then the tables can be populated:

```
def package_priority_parsing_service(some_package_keys, some_package_hash_table):

    # Create hash tables for each package classification array based on the size of the array
    constrained_delivery_package_table = PackageHashTable(len(constrained_package_keys))
    priority_delivery_package_table = PackageHashTable(len(priority_package_keys))
    standard_delivery_package_table = PackageHashTable(len(standard_package_keys))

    # Take package classification arrays and build distinct hash tables for each classification
    for package_id in priority_package_keys:
        priority_delivery_package_table.insert(package_id, instanced_package_hash_table.get_by_id(package_id))
    for package_id in constrained_package_keys:
        constrained_delivery_package_table.insert(package_id, instanced_package_hash_table.get_by_id(package_id))
    for package_id in standard_package_keys:
        standard_delivery_package_table.insert(package_id, instanced_package_hash_table.get_by_id(package_id))
```

C2. Generate batch distance matrix

The next step in this process is to dynamically generate distance matrices for each package batch, extracting only data elements that correspond to destinations in this batch from the original distance table. Here I will briefly redirect attention to the *distance_matrix_builder.py* file where the function for performing these operations is defined. The first steps of generating a distance matrix for each batch involve determining the size of the matrix, and extracting the row/column identifiers (addresses) from the original distance table for use in the new matrix:



With the distance matrix defined at an adequate size and the row/column address labels inserted, the matrix can be populated with distance values corresponding only to addresses in this batch.

```
    # Get distances from full distance matrix that correspond only to distances between
    # addresses in this distance matrix.
    # Insert the distances in to the matrix at the coordinates that correspond to the associated
    # addresses.
    for address in distance_matrix[0]:
        try:
            primary_index = batch_destinations.index(address) + 1
            parent_matrix_index = distance_matrix[0].index(address)
            for i, row_values in enumerate(distance_matrix[parent_matrix_index]):
                if distance_matrix[0][i] in batch_destinations:
                    secondary_index = batch_destinations.index(distance_matrix[0][i]) + 1
                    temp_dist_matrix[primary_index][secondary_index] = row_values
        except ValueError:
            continue

    # Return this distance matrix for assignment where called.
    return temp_dist_matrix
```

Switching focus back to the *package_priority_parsing_service.py*, the *distance_matrix_builder* function can now be called on the categorized package keys lists:

```
    # Take classification distinct package hash tables and generate distance matrices for packages
    # in that classification group
    priority_package_distance_matrix = distance_matrix_builder(priority_package_keys,
                                                               priority_delivery_package_table)
    constrained_package_distance_matrix = distance_matrix_builder(constrained_package_keys,
                                                                  constrained_delivery_package_table)
    standard_package_distance_matrix = distance_matrix_builder(standard_package_keys,
                                                               standard_delivery_package_table)
```

C2. Nearest Neighbor Heuristic and Package Deadline Prioritization

Now that the hash tables and distance matrices for each batch have been constructed, the next step is to optimize the delivery routes for each batch. I created a separate file for *nearest_neighbor_path_generator* to implement the nearest neighbor heuristic algorithm.

```
def nearest_neighbor_path_generator(some_package_keys, some_package_hash_table,
                                    some_distance_matrix):
    HUB_ADDRESS = some_distance_matrix[0][1]
    distance_traveled_array = []
    current_node_address = ''
    current_node_index = 0
    visited_nodes = []
    high_priority_packages = []
    high_priority_package_keys = []
    aggregate_time = 0.0
    destination_count = 0
    num_packages_delivered = 0
```

The process for generating optimized routes begins by comparing the priority of packages within a batch to see if any packages have greater priority than others. The implementation of these in-batch priority comparisons only compares deadlines to see if there are any packages with an earlier deadline than others, it does not prioritize packages with deadlines over other packages that do not have a specific deadline. I chose not to prioritize all packages with deadlines over non-deadline packages because otherwise doing so would essentially eliminate a large portion of the optimization achieved through the nearest neighbor algorithm by defining the route based on deadlines rather than nearest neighbors. This approach of only prioritizing packages with lesser/earlier deadlines enables more distance optimization while ensuring that packages that truly have priority over the other packages can have their delivery deadlines met. The code for this portion is on the next page.

```python
def nearest_neighbor_path_generator(some_package_keys, some_package_hash_table,

    # Isolate packages with specified delivery time deadlines into a separate array.
    deadline_package_keys = []
    for package_key in some_package_keys:
        if "EOD" not in some_package_hash_table.get_by_id(package_key).deadline:
            deadline_package_keys.append(package_key)

    if deadline_package_keys:
        # Loop to compare package deadline with the deadlines of other packages.
        for package_key in deadline_package_keys:
            package_to_check = some_package_hash_table.get_by_id(package_key)
            if "EOD" in package_to_check.deadline:
                continue
            # Convert given time string to a datetime object of format HH:MM AM/PM.
            this_deadline = datetime.strptime(package_to_check.deadline, time_format).time()
            # If this_deadline is less than any other deadline (i.e., deadline is sooner), then the package has priority.
            if any(
                    this_deadline < (datetime.strptime(some_package_hash_table.get_by_id(other_key).deadline,
                        time_format).time())
                    for other_key in deadline_package_keys
            ):
                # If the package has priority, then add it to the high_priority_packages array and the key array.
                high_priority_packages.append(package_to_check)
                high_priority_package_keys.append(package_key)
```

Before adding the priority packages to the delivery sequence, the *nearest_neighbor_path_generator* identifies the position of the *Hub* in the distance list to use as the starting point of the path traversal. It adds the hub address as a dictionary to the visited_nodes list at index position 0. I chose to implement the visited_nodes list as a list of dictionaries in order to store the address and associated packages together for referencing.

```python
    # Find the starting node i.e., the hub.
    for i, row in enumerate(some_distance_matrix[1:], start=1):
        if row[1] == 0.0:
            current_node_address = row[0]
            visited_nodes.append({
                "address": current_node_address,
                "associated_packages": None
            })
            current_node_index = i
        destination_count = i
```

With the starting point for the route set to the *Hub* address, the algorithm moves on to adding the priority packages for the batch to the *visited_nodes_array*, ordering the destinations by nearest neighbor comparisons starting from the *Hub*.

```python
    # Check if there are any packages in high priority package array for this batch.
    if high_priority_packages:

        # For all packages in the high_priority_packages_array (identified by keys array).
        for package_key in high_priority_package_keys:
            # Get package delivery address from hash table.
            this_address = some_package_hash_table.get_by_id(package_key).address
            # Get all other address associated packages.
            associated_packages = some_package_hash_table.get_by_address(
                some_package_hash_table.get_by_id(package_key).address)
            # Add the high_priority package address to visited nodes.
            visited_nodes.append({
                "address": this_address,
                "associated_packages": associated_packages
            })

            # Get the distance matrix index position for this address.
            for i, address in enumerate(some_distance_matrix[0][1:], start=1):
                if this_address in address:
                    this_address_index = i
                    # break from loop when found
                    break
            # Get the distance from the current address (e.g., the hub) to this high priority delivery address
            # and append it to the distance traveled array.
            distance_traveled_array.append(some_distance_matrix[this_address_index][current_node_index])
            # Set current_node_index and current_node_address for next loop iteration
            current_node_index = this_address_index
            current_node_address = this_address

            # Increment the number of packages delivered by the number of packages associated with this address
            num_packages_delivered += len(associated_packages)

        # Clear the list so that the next loop iteration can skip this block
        # because there are no more priority packages
        high_priority_packages.clear()
```

Once the *high_priority_packages* in a batch have been assigned priority positions in the delivery sequence, the *nearest_neighbor_path_generator* function moves onto the remaining packages in the batch. The delivery sequence is determined using a nearest neighbor algorithm that searches the batch specific distance table for the shortest distance from the current node to another node that has not yet been visited on this route. This portion of the algorithm also optimizes the route by searching the hash table for other packages in the batch that share an address with the current package.

C950 Task-2 WGUPS Write-Up

```python
def nearest_neighbor_path_generator(some_package_keys, some_package_hash_table,  4 usages  Dayton

    if not high_priority_packages:
        # Create an empty dictionary to hold the distances TO other destinations FROM current_node.
        neighbor_distances_array = {}
        for delivery_stop in range(len(high_priority_package_keys), destination_count + 1):
            # Add each neighbor distance for current_node to the distance dictionary.
            for row in some_distance_matrix[1:]:
                # First, check that the distance being added does not correspond to an already visited
                # destination.
                if not any(row[0] in node['address'] for node in visited_nodes):
                    # If the distance value does not correspond to a visited node, add it to distance
                    # dictionary, where the key is the node address and the value is the distance to
                    # that node from current_node.
                    neighbor_distances_array[row[0]] = row[current_node_index]
                else:
                    continue

            # Identify the nearest neighbor to be the dictionary element containing the smallest
            # distance value.
            nearest_neighbor = min(neighbor_distances_array.items(), key=lambda item: item[1])

            # Adjust current_node_index to be this nearest_neighbor in prep for next iteration.
            for i, row in enumerate(some_distance_matrix[1:], start=1):
                if row[0] == nearest_neighbor[0]:
                    current_node_index = i

            # Gather packages associated with the address of nearest_neighbor into a hash table.
            associated_packages = some_package_hash_table.get_by_address(nearest_neighbor[0])

            # Add associated_packages address to visited nodes if there are packages associated with
            # this address.
            current_node_address = nearest_neighbor[0]

            visited_nodes.append({
                "address": current_node_address,
                "associated_packages": associated_packages
            })
            # print(visited_nodes[0])

            path_time = calc_travel_time_minutes(nearest_neighbor[1])

            aggregate_time += path_time
            distance_traveled_array.append(nearest_neighbor[1])

            # Empty reusable data structure for next iteration.
            neighbor_distances_array.clear()

            # Aggregate the total number of packages delivered in this route for
            # termination upon delivery of all packages.
            num_packages_delivered += len(associated_packages)

            # Terminate route planning if all packages have been accounted for.
            if num_packages_delivered >= len(some_package_keys):
                break
```

The last step for the *nearest_neighbor_path_generator* function is to initialize a dictionary of dictionaries, with the primary dictionary's key being an integer representation of the destination's number/position in the delivery sequence. This integer key is then paired with the corresponding destination information such as, *address*, *associated_packages*, and the sum of the combined distances traveled from the *Hub* to arrive at this destination. Once these dictionaries for each destination have been inserted into the *nearest_neighbor_route* dictionary, *nearest_neighbor_path_generator* returns this dictionary of routes.

```
def nearest_neighbor_path_generator(some_package_keys, some_package_hash_table,  4 usages  Dayton
    # Create a dictionary to insert optimized route based on delivery sequence value (as key).
    nearest_neighbor_route = {}
    for d, destination in enumerate(visited_nodes):
        if d == 0:
            route_info = {
                "destination_number": d,
                "address": HUB_ADDRESS,
                "associated_packages": None,
                "distance": 0
            }
        elif 0 < d < len(visited_nodes):
            route_info = {
                "destination_number": d,
                "address": destination["address"],
                "associated_packages": destination["associated_packages"],
                "distance": sum(distance_traveled_array[0:d])
            }
        # Calculate return to hub distance from final delivery address.
        if d == len(visited_nodes) - 1:
            final_delivery_matrix_index = some_distance_matrix[0].index(route_info["address"])
            return_to_hub_distance = some_distance_matrix[final_delivery_matrix_index][1]

            route_termination_info = {
                "destination_number": d + 1,
                "address": HUB_ADDRESS,
                "associated_packages": None,
                "distance": sum(distance_traveled_array[0:d]) + return_to_hub_distance
            }
            nearest_neighbor_route[d + 1] = route_termination_info
        # Add dictionary value for each destination to route dictionary.
        nearest_neighbor_route[d] = route_info

        # Add return to HUB to delivery route if all deliveries for route are complete.


    # Return the route dictionary for assignment where called.
    return nearest_neighbor_route
```

C2. Load Trucks

The *nearest_neighbor_path_generator* is called from the *package_priority_parsing_service.py*, which allows for the route returned from the path generator to be assigned to a variable in *package_priority_parsing_service.py*. This routes are then passed to the *batch_load_truck_service* function, along with the package keys, hash table, and a route label associated with each route.

```
def package_priority_parsing_service(some_package_keys, some_package_hash_table):  2 usages  Dayton *
    # Take classification distinct package hash tables and generate distance matrices for packages
    # in that classification group
    priority_package_distance_matrix = distance_matrix_builder(priority_package_keys,
                                                               priority_delivery_package_table)
    constrained_package_distance_matrix = distance_matrix_builder(constrained_package_keys,
                                                                  constrained_delivery_package_table)
    standard_package_distance_matrix = distance_matrix_builder(standard_package_keys,
                                                               standard_delivery_package_table)

    # Pass the distinct package classification data sets (keys, hash table, distance matrix)
    # into the nearest neighbor path generator to create a greedy delivery route
    priority_delivery_route = nearest_neighbor_path_generator(priority_package_keys, priority_delivery_package_table,
                                                              priority_package_distance_matrix)

    constrained_delivery_route = nearest_neighbor_path_generator(constrained_package_keys,
                                                                 constrained_delivery_package_table,
                                                                 constrained_package_distance_matrix)

    standard_delivery_route = nearest_neighbor_path_generator(standard_package_keys, standard_delivery_package_table,
                                                              standard_package_distance_matrix)

    # Take delivery route and load packages into truck
    batch_truck_load_service(priority_delivery_route, priority_package_keys,
                             priority_delivery_package_table, label: "priority batch")

    batch_truck_load_service(constrained_delivery_route, constrained_package_keys, constrained_delivery_package_table,
                             label: "constrained batch")

    batch_truck_load_service(standard_delivery_route, standard_package_keys, standard_delivery_package_table,
                             label: "standard batch")
```

The *batch_truck_load_service* uses the *label* value passed as an argument in its function call to determine which truck the passed route needs to be loaded onto.

```python
def batch_truck_load_service(route_path, some_package_keys, some_package_hash_table, label):
    total_package_count = 0
    batch = deque()
    batch_package_count = 0
    package_match_count = 0
    carryover_packages = []

    if "priority batch" in label:
        truck_number = 1
    elif "constrained batch" in label:
        truck_number = 2
    elif "standard batch" in label:
        truck_number = 3
    else:
        truck_number = 4
```

Next, the *batch_truck_load_service* function loops through all of the destinations in the route path, or until the truck capacity is maxed out, to select all of the associated packages in this batch that can be loaded onto the designated truck. The loop starts with a condition check for carryover_packages, which only evaluates to true if the loop has already been executed. The carryover_packages list was necessary because of how the function checks if there is room on the truck for all of the packages associated with the current route destination. A truck may be able to accept more packages (i.e., it could have 15 or fewer packages), but if the destination has multiple associated packages, loading all of these packages onto the truck would result in overloading the truck. Which means that this destination that would require overloading the truck should not be visited by this truck. So, the packages must be carried over to the next truck operating on this route.

```python
def batch_truck_load_service(route_path, some_package_keys, some_package_hash_table, label):
    for delivery_stop in route_path.values():
        # Check whether there are carryover_packages from last batch; see ---->>>> below.
        # If carryover packages exist, add them to current batch.
        if carryover_packages:
            for carryover_package in carryover_packages:
                batch.append(carryover_package)
                batch_package_count += 1
                total_package_count += 1
            carryover_packages = []

        # Check hash table for other packages associated with this address so that they can be
        # delivered at the same time
        address_associated_packages = some_package_hash_table.get_by_address(delivery_stop["address"])
        package_match_count += len(address_associated_packages)

        # If there are no associated_packages (e.g., it is the hub)
        # then don't load packages for this address. Continue to loop condition.
        if len(address_associated_packages) == 0:
            continue

        # Compare what the size of the batch would be if associated_packages were added to the batch.
        # Do not attempt to load if batch size would be above truck capacity.
        if ((batch_package_count + len(address_associated_packages) <= 16) and total_package_count
                < len(some_package_keys)):
            for package in address_associated_packages:
                    batch.append(some_package_hash_table.get_by_id(package.package_id))
                    batch_package_count += 1
                    total_package_count += 1
            can_load_more = True
        else:
            can_load_more = False
            # ---->>>>
            # If this point is reached, then the number of associated_packages is too large to fit in
            # the current truck batch. But the loop iteration has already visited this address, so
            # we cannot just continue the loop or these packages will be skipped for truck loading.
            # Add the associated packages to carry_over_packages for inclusion in the next batch.
            for package in address_associated_packages:
                carryover_packages.append(some_package_hash_table.get_by_id(package.package_id))
```

Once the number of packages loaded from this batch has reached the maximum value that will fit on the current truck, the deque container for the packages is passed to the *load_truck* method of the *Truck* class.

```
def batch_truck_load_service(route_path, some_package_keys, some_package_hash_table, label):  4 usages  Dayton *
        # Once the number of packages in a batch reaches the maximum possible per constraints,
        # load the packages onto corresponding truck and clear batch to start new.
        if ((batch_package_count == 16) or (package_match_count == len(some_package_keys))
                or can_load_more == False):

            # print("\nTruck number: ", truck_number)
            # print("Batch", truck_number, "package count: ", batch_package_count)
            # if package_match_count == len(some_package_keys):
            #     print("\nTotal package count: ", total_package_count)

            Truck.trucks_dict.get(truck_number).load_truck(batch)

            batch.clear()
            address_associated_packages.clear()
            batch_package_count = 0
            can_load_more = True
```

C2. Instantiating Route objects

The final process carried out in the *package_priority_parsing_service* involves instantiating route objects which contain all of the route metadata necessary to return the desired information to the UI. This requires initializing variables to hold values for the max key (last destination), total distance, and duration of each route, and then assigning the appropriate values to all of the attributes for each route object. The *package_priority_parsing_service* returns a list of these route objects for use elsewhere.

```
    # Retrieve the max (last) key for each route to use for distance calculation
    priority_route_max_key = max(priority_delivery_route.keys())
    constrained_route_max_key = max(constrained_delivery_route.keys())
    standard_route_max_key = max(standard_delivery_route.keys())

    # Calculate total distances for routes
    priority_route_total_distance = priority_delivery_route[priority_route_max_key]["distance"]
    constrained_route_total_distance = constrained_delivery_route[constrained_route_max_key]["distance"]
    standard_route_total_distance = standard_delivery_route[standard_route_max_key]["distance"]

    # Calculate total route duration
    priority_route_duration = calc_travel_time_minutes(priority_route_total_distance)
    constrained_route_duration = calc_travel_time_minutes(constrained_route_total_distance)
    standard_route_duration = calc_travel_time_minutes(standard_route_total_distance)
```

C2. Run the Delivery Service

Main.py contains a function *get_fresh_routes*, which calls the *instantiate_delivery_infra* function from the *utils* module as well as the *start_delivery_service* function from the *delivery_service* file in the *services* module. *Instantiate_delivery_infra* instantiates Truck and Driver objects to be utilized in the *Truck* loading process. *Start_delivery_service* calls the *package_priority_parsing_service*, passing all of the package data from the imported csv file as well as the list of all package keys and returning a list of route_objects to be passed to the *query_delivery_service* function. The *main()* method is discussed in the next section.

delivery_service.py

```python
today = date.today()
TIME_FORMAT = "%I:%M %p"
fmt = "%H:%M"
TRUCK_SPEED = 18
temp_truck1_time = datetime.strptime("8:00 AM", TIME_FORMAT).time()
TRUCK1_START_TIME = datetime.combine(today, temp_truck1_time)
temp_truck2_time = datetime.strptime("09:05 AM", TIME_FORMAT).time()
TRUCK2_START_TIME = datetime.combine(today, temp_truck2_time)

# Generate route data for delivery service
def start_delivery_service(all_package_keys, all_package_hash_table):

    route_objects = package_priority_parsing_service(all_package_keys, all_package_hash_table)

    return route_objects
```

C2. Query the Delivery Service

The *query_delivery_service* function takes a *time* string, the *route_objects* list, and a *condition_code* as arguments. The *main()* function in *main.py* consists of various branches with execution that is conditional upon the user input. For the branches that call *query_delivery_service*, a *time* string will be provided either via user input or hardcoded into the branch logic. The *route_objects* passed to *query_delivery_service* reference a hash table that is a copy of the main hash table structure in order to allow repeated queries without changes made to package data to carry over between queries.

The *condition_code* parameter of this function is used to direct the output of the function to the output that corresponds to the user's request.

delivery_service.py

```python
# Function for running a simulation of delivery process and returning values depending on user input time
def query_delivery_service(time, route_objects, condition_code):  4 usages  Dayton
    try:
        temp_parsed_time = datetime.strptime(time, TIME_FORMAT).time()
        parsed_input_time = datetime.combine(today, temp_parsed_time)
    except ValueError:
        print(f"Time must be in format HH:MM AM/PM, e.g. '02:30 PM'.")
        return

    all_trucks = Truck.trucks_dict
    truck1 = all_trucks[1]
    truck2 = all_trucks[2]
    truck3 = all_trucks[3]

    all_trucks_elapsed_time = 0

    # CREATE COPIES OF ROUTE SPECIFIC HASH TABLES FOR STATUS ALTERATIONS
    this_temp_routes = route_objects.copy()

    # Calculate route completion times for trucks 1 and 2 to get truck 3 start time;
    # waiting for an available driver.
    constrained_route_complete_time = TRUCK2_START_TIME + timedelta(minutes=this_temp_routes[1].duration)
    priority_route_complete_time = TRUCK1_START_TIME + timedelta(minutes=this_temp_routes[0].duration)

    # Format complete times to remove the seconds/microseconds and round up by one minute
    constrained_route_complete_time = (constrained_route_complete_time.replace(second=0, microsecond=0)
                                       + timedelta(minutes=1))
    priority_route_complete_time = (priority_route_complete_time.replace(second=0, microsecond=0) +
                                    timedelta(minutes=1))
```

```python
    # Set truck3_start_time to lesser/earlier time between truck 1 and truck 2 completion times
    temp_truck3_time = min(constrained_route_complete_time, priority_route_complete_time) + timedelta(minutes=1)
    truck3_start_time = datetime.combine(today, (temp_truck3_time.time()))

    truck1.route_start_time = TRUCK1_START_TIME
    truck2.route_start_time = TRUCK2_START_TIME
    truck3.route_start_time = truck3_start_time

    standard_route_complete_time = truck3_start_time + timedelta(minutes=this_temp_routes[2].duration)
    standard_route_complete_time = (standard_route_complete_time.replace(second=0, microsecond=0)
                                    + timedelta(minutes=1))

    route_completion_times = [priority_route_complete_time, constrained_route_complete_time,
                              standard_route_complete_time]
```

Prior to using the condition code to determine the output of the query, the *query_delivery_service* function takes the user input *time* and determines the statuses of all *Packages* and *Trucks* at the given time, beginning with setting truck statuses and calculating *truck.en_route_time* and *distance_traveled*.

delivery_service.py

```python
def query_delivery_service(time, route_objects, condition_code):  4 usages  Dayton *
    # Calculate the amount of time each truck has been on route for delivery
    # from truck-specific start time, up to user input time
    for i, truck in enumerate(all_trucks.values(), start=0):
        time_dif = parsed_input_time - truck.route_start_time
        truck.en_route_time = time_dif.total_seconds() / 60.0

        # If the truck en_route_time (or time_dif) is less than zero, then the truck has not left the hub
        if 0 < truck.en_route_time:
            truck.status = "en route"
            # Set truck driver based on route number
            if i == 0 or i == 2:
                truck.driver_id = 1
            elif i == 1:
                truck.driver_id = 2
            # If the duration of the route is greater than zero but
            # less than the elapsed time since the truck left the hub,
            # then the truck has already traveled the total route distance
            if this_temp_routes[i].duration <= truck.en_route_time:
                truck.distance_traveled = this_temp_routes[i].total_distance
                truck.en_route_time = this_temp_routes[i].duration
            # Otherwise, if the route duration is greater than the amount of time the truck
            # has been on route, then it has only partially completed the route.
            else:
                # Truck distance traveled can be computed using the route time in hours, times the truck speed
                truck.distance_traveled = (truck.en_route_time / 60) * TRUCK_SPEED

        # If input time is after the completion time for the route, then the route has been completed
        # and the truck returned to the hub.
        if parsed_input_time > route_completion_times[i]:
            truck.status = "Returned to Hub"
            all_trucks_elapsed_time = all_trucks_elapsed_time + truck.en_route_time
            truck.driver_id = None

        # If the en route time for the truck is less than zero, then it has not left the hub yet.
        elif time_dif <= timedelta(0):
            truck.distance_traveled = 0
            truck.en_route_time = 0
```

With these values calculated for each truck at the given time, the next step is to calculate the status of each package on each truck at this time.

delivery_service.py

```python
def query_delivery_service(time, route_objects, condition_code):  4 usages  Dayton *
    # Calculate the statuses of all packages at user input time.
    # Loop through all Trucks.
    for i, truck in enumerate(all_trucks.values(), start=0):
        # If the truck has left the Hub
        if truck.en_route_time > 0:
            # Loop through all destinations for a given truck
            for destination in this_temp_routes[i].metadata.values():
                if destination["distance"] <= truck.distance_traveled:
                    # Loop through all packages on this truck to find all packages associated with this address
                    for package in truck.packages:
                        # If, by user input time, the given truck has traveled far enough to have reached the
                        # package of this loop iteration, then the package has been delivered.
                        if destination["address"] in package.address:
                            # Calculate the elapsed time from truck departure to delivery of packages for this address.
                            en_route_time_to_delivery = calc_travel_time_minutes(destination["distance"])
                            # Calculate the delivery time of packages associated with this address.
                            package_delivery_time = (truck.route_start_time + timedelta(minutes=en_route_time_to_delivery))
                            # Set the package status of each package at this address to the calculated delivery time.
                            package.status = "Delivered: " + datetime.strftime(package_delivery_time, TIME_FORMAT)
                # If the truck has not traveled far enough to reach this loop iteration destination, the package
                # has not been delivered yet.
                # If this point in the control structure has been reached, then the package is out for delivery.
                # Set the status of the packages associated with this destination to en route
                elif destination["distance"] > truck.distance_traveled:
                    for package in truck.packages:
                        if destination["address"] in package.address:
                            package.status = "Package en route: Truck #" + str(truck.truck_id)
```

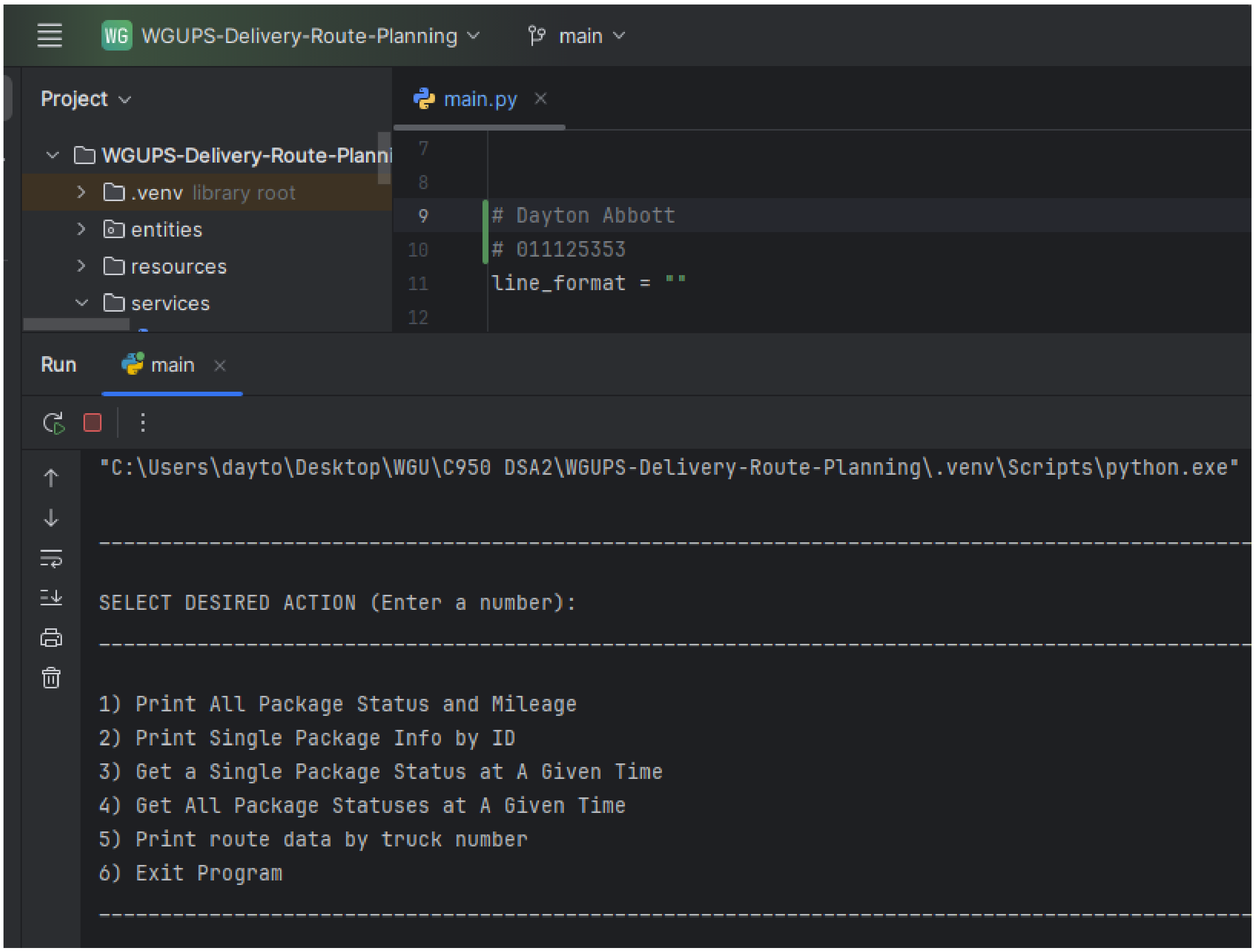
Once the statuses for *Trucks* and *Packages* have all been modified to reflect the statuses at the user's input time, the condition code is evaluated to determine which data has been requested and then output the corresponding data.

delivery_service.py

```python
def query_delivery_service(time, route_objects, condition_code):  4 usages  Dayton *
    # Check input condition_code to determine query response
    # IF CONDITION_CODE == 1
    if condition_code == 1:

        all_truck_hours = int(all_trucks_elapsed_time / 60)
        all_truck_mins = int(all_trucks_elapsed_time % 60)

        for truck in all_trucks.values():
            print(truck)
        print("TOTAL DELIVERY MILEAGE: ", truck1.distance_traveled + truck2.distance_traveled
              + truck3.distance_traveled, "miles\n")
        print("TOTAL DRIVER TIME (CONCURRENT): ", all_truck_hours, "hours and", all_truck_mins, "minutes\n")

    # IGNORE CONDITION_CODE 2, ITS HANDLED ELSEWHERE
    if condition_code == 2:
        return

    # IF CONDITION_CODE == 3
    if condition_code == 3:
        requested_package = int(input("Input package number (1-40):\n"))
        for truck in all_trucks.values():
            for package in truck.packages:
                if requested_package == package.package_id:
                    print("Package requested: \n", package)

    # IF CONDITION_CODE == 4
    if condition_code == 4:
        for truck in all_trucks.values():
            print(truck)

    # IGNORE CONDITION_CODE 5, ITS HANDLED ELSEWHERE
    if condition_code == 5:
        return

    if condition_code == 6:
        exit(1)
```

D. Interface

Delivery service running in PyCharm with user interface in the terminal.



D1. First Status Check

Trucks 1 and 2 loaded and en route at 9:15 AM

```
Run - WGUPS-Delivery-Route-Planning
main ×
------------------------------------------------------------------------------------------------------------------------------------

4
Enter time in format: HH:MM:AM/PM
 (space between time and AM/PM)
09:15 AM

------------------------------------------------------------------------------------------------------------------------------------
Batch #1 Loaded onto Truck for Optimized Route:
------------------------------------------------------------------------------------------------------------------------------------
Truck #1
Driver ID: 1
Departure Time: 08:00 AM
Time on Route: 75.0 minutes
Truck Route Distance Traveled: 22.5 miles
Truck Status: en route

Package 34:    Delivered: 08:11 AM          4580 S 2300 E                       Holladay          UT      84117    10:30 AM      2.0kg
Package 15:    Delivered: 08:11 AM          4580 S 2300 E                       Holladay          UT      84117    9:00 AM       4.0kg
Package 29:    Delivered: 08:28 AM          1330 2100 S                         Salt Lake City    UT      84106    10:30 AM      2.0kg
Package 33:    Delivered: 08:33 AM          2530 S 500 E                        Salt Lake City    UT      84106    EOD           1.0kg
Package 1:     Delivered: 08:38 AM          195 W Oakland Ave                   Salt Lake City    UT      84115    10:30 AM     21.0kg
Package 40:    Delivered: 08:42 AM          380 W 2880 S                        Salt Lake City    UT      84115    10:30 AM     45.0kg
Package 31:    Delivered: 08:47 AM          3365 S 900 W                        Salt Lake City    UT      84119    10:30 AM      1.0kg
Package 35:    Delivered: 09:02 AM          1060 Dalton Ave S                   Salt Lake City    UT      84104    EOD          88.0kg
Package 27:    Delivered: 09:02 AM          1060 Dalton Ave S                   Salt Lake City    UT      84104    EOD           5.0kg
Package 39:    Delivered: 09:08 AM          2010 W 500 S                        Salt Lake City    UT      84104    EOD           9.0kg
Package 13:    Delivered: 09:08 AM          2010 W 500 S                        Salt Lake City    UT      84104    10:30 AM      2.0kg
Package 37:    Package en route: Truck #1   410 S State St                      Salt Lake City    UT      84111    10:30 AM      2.0kg
Package 30:    Package en route: Truck #1   300 State St                        Salt Lake City    UT      84103    10:30 AM      1.0kg

------------------------------------------------------------------------------------------------------------------------------------



------------------------------------------------------------------------------------------------------------------------------------
Batch #2 Loaded onto Truck for Optimized Route:
------------------------------------------------------------------------------------------------------------------------------------
Truck #2
Driver ID: 2
Departure Time: 09:05 AM
Time on Route: 10.0 minutes
Truck Route Distance Traveled: 3.0 miles
Truck Status: en route

Package 14:    Delivered: 09:11 AM          4300 S 1300 E                       Millcreek         UT      84117    10:30 AM     88.0kg
Package 16:    Package en route: Truck #2   4580 S 2300 E                       Holladay          UT      84117    10:30 AM     88.0kg
Package 25:    Package en route: Truck #2   5383 South 900 East #104            Salt Lake City    UT      84117    10:30 AM      7.0kg    De
Package 20:    Package en route: Truck #2   3595 Main St                        Salt Lake City    UT      84115    10:30 AM     37.0kg
Package 28:    Package en route: Truck #2   2835 Main St                        Salt Lake City    UT      84115    EOD           7.0kg    De
Package 32:    Package en route: Truck #2   3365 S 900 W                        Salt Lake City    UT      84119    EOD           1.0kg    De
Package 6:     Package en route: Truck #2   3060 Lester St                      West Valley City  UT      84119    10:30 AM     88.0kg    De
Package 36:    Package en route: Truck #2   2300 Parkway Blvd                   West Valley City  UT      84119    EOD          88.0kg
Package 18:    Package en route: Truck #2   1488 4800 S                         Salt Lake City    UT      84123    EOD           6.0kg
Package 38:    Package en route: Truck #2   410 S State St                      Salt Lake City    UT      84111    EOD           9.0kg
Package 3:     Package en route: Truck #2   233 Canyon Rd                       Salt Lake City    UT      84103    EOD           2.0kg
```

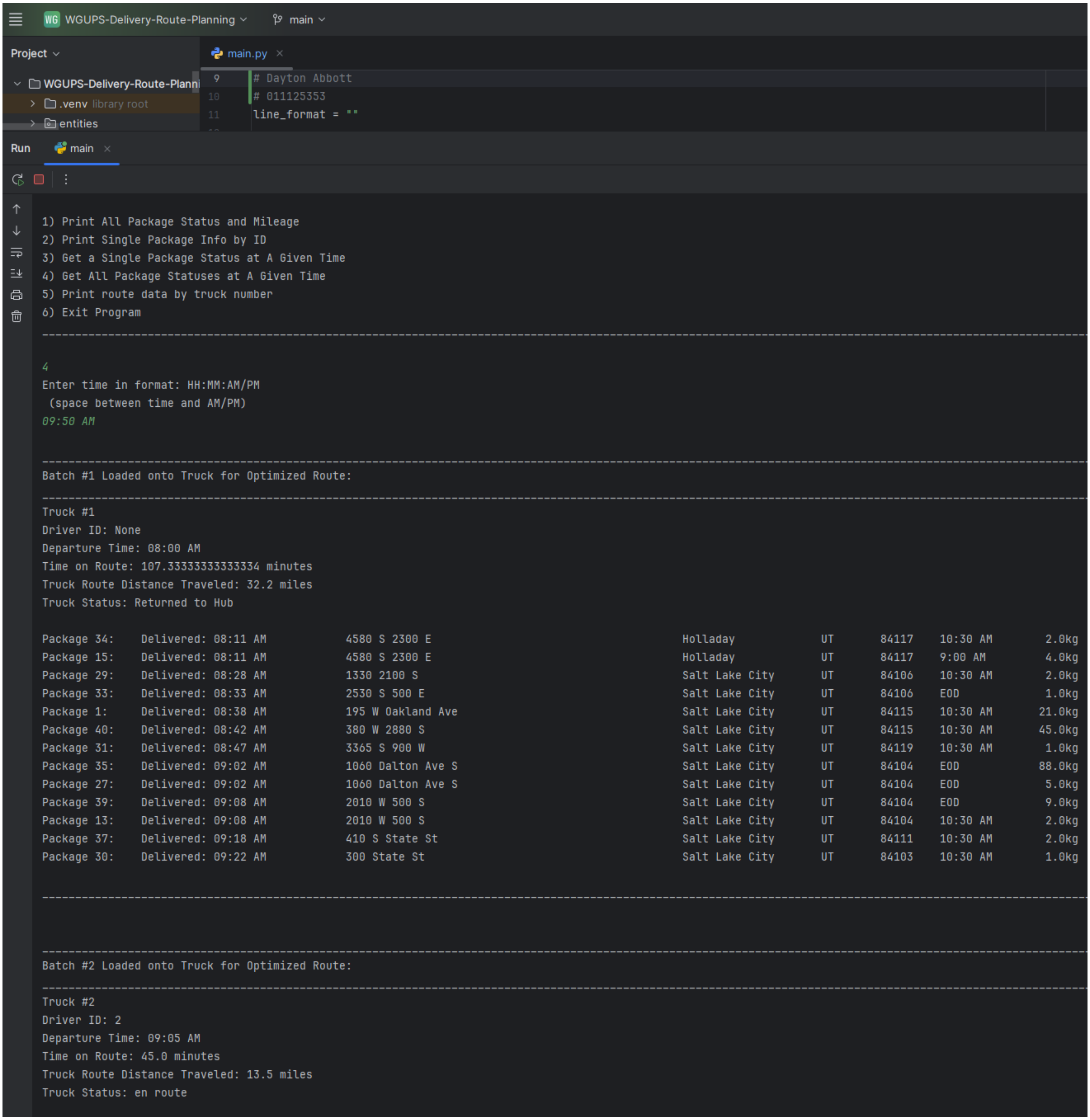
Truck 3 loaded and at Hub at 09:15 AM

(Can't fit all trucks in one screenshot, truck 2 stats show same time)

```
Run - WGUPS-Delivery-Route-Planning
main

------------------------------------------------------------------------------------------------------------------------


------------------------------------------------------------------------------------------------------------------------
Batch #2 Loaded onto Truck for Optimized Route:
------------------------------------------------------------------------------------------------------------------------
Truck #2
Driver ID: 2
Departure Time: 09:05 AM
Time on Route: 10.0 minutes
Truck Route Distance Traveled: 3.0 miles
Truck Status: en route

Package 14:   Delivered: 09:11 AM          4300 S 1300 E                           Millcreek         UT     84117   10:30 AM    88.0kg
Package 16:   Package en route: Truck #2   4580 S 2300 E                           Holladay          UT     84117   10:30 AM    88.0kg
Package 25:   Package en route: Truck #2   5383 South 900 East #104                Salt Lake City    UT     84117   10:30 AM     7.0kg   Del
Package 20:   Package en route: Truck #2   3595 Main St                            Salt Lake City    UT     84115   10:30 AM    37.0kg
Package 28:   Package en route: Truck #2   2835 Main St                            Salt Lake City    UT     84115   EOD          7.0kg   Del
Package 32:   Package en route: Truck #2   3365 S 900 W                            Salt Lake City    UT     84119   EOD          1.0kg   Del
Package 6:    Package en route: Truck #2   3060 Lester St                          West Valley City  UT     84119   10:30 AM    88.0kg   Del
Package 36:   Package en route: Truck #2   2300 Parkway Blvd                       West Valley City  UT     84119   EOD         88.0kg
Package 18:   Package en route: Truck #2   1488 4800 S                             Salt Lake City    UT     84123   EOD          6.0kg
Package 38:   Package en route: Truck #2   410 S State St                          Salt Lake City    UT     84111   EOD          9.0kg
Package 3:    Package en route: Truck #2   233 Canyon Rd                           Salt Lake City    UT     84103   EOD          2.0kg

------------------------------------------------------------------------------------------------------------------------


------------------------------------------------------------------------------------------------------------------------
Batch #3 Loaded onto Truck for Optimized Route:
------------------------------------------------------------------------------------------------------------------------
Truck #3
Driver ID: None
Departure Time: 09:49 AM
Time on Route: 0 minutes
Truck Route Distance Traveled: 0 miles
Truck Status: At Hub

Package 21:   At Hub                       3595 Main St                            Salt Lake City    UT     84115   EOD          3.0kg
Package 19:   At Hub                       177 W Price Ave                         Salt Lake City    UT     84115   EOD         37.0kg
Package 12:   At Hub                       3575 W Valley Central Station bus Loop  West Valley City  UT     84119   EOD          1.0kg
Package 7:    At Hub                       1330 2100 S                             Salt Lake City    UT     84106   EOD          8.0kg
Package 2:    At Hub                       2530 S 500 E                            Salt Lake City    UT     84106   EOD         44.0kg
Package 4:    At Hub                       380 W 2880 S                            Salt Lake City    UT     84115   EOD          4.0kg
Package 17:   At Hub                       3148 S 1100 W                           Salt Lake City    UT     84119   EOD          2.0kg
Package 24:   At Hub                       5025 State St                           Murray            UT     84107   EOD          7.0kg
Package 26:   At Hub                       5383 South 900 East #104                Salt Lake City    UT     84117   EOD         25.0kg
Package 22:   At Hub                       6351 South 900 East                     Murray            UT     84121   EOD          2.0kg
Package 11:   At Hub                       2600 Taylorsville Blvd                  Salt Lake City    UT     84118   EOD          1.0kg
Package 23:   At Hub                       5100 South 2700 West                    Salt Lake City    UT     84118   EOD          5.0kg
Package 10:   At Hub                       600 E 900 South                         Salt Lake City    UT     84105   EOD          1.0kg
Package 5:    At Hub                       410 S State St                          Salt Lake City    UT     84111   EOD          5.0kg
Package 9:    At Hub                       410 S State St                          Salt Lake City    UT     84111   EOD            2kg
Package 8:    At Hub                       300 State St                            Salt Lake City    UT     84103   EOD          9.0kg
```

D2. Second Status Check

Truck 1 status shot at 9:50 AM



Trucks 2 and 3 status shot at 9:50 AM

```
WG WGUPS-Delivery-Route-Planning ˅   main ˅

ject ˅                         main.py ×
WGUPS-Delivery-Route-Planni     9   # Dayton Abbott
 > .venv library root          10   # 011125353
 > entities                    11   line_format = ""

main ×

Batch #2 Loaded onto Truck for Optimized Route:
------------------------------------------------------------------------------------------------------------------------------
Truck #2
Driver ID: 2
Departure Time: 09:05 AM
Time on Route: 45.0 minutes
Truck Route Distance Traveled: 13.5 miles
Truck Status: en route

Package 14:    Delivered: 09:11 AM          4300 S 1300 E                              Millcreek          UT     84117    10:30 AM
Package 16:    Delivered: 09:18 AM          4580 S 2300 E                              Holladay           UT     84117    10:30 AM
Package 25:    Delivered: 09:29 AM          5383 South 900 East #104                   Salt Lake City     UT     84117    10:30 AM
Package 20:    Delivered: 09:42 AM          3595 Main St                               Salt Lake City     UT     84115    10:30 AM
Package 28:    Delivered: 09:46 AM          2835 Main St                               Salt Lake City     UT     84115    EOD
Package 32:    Package en route: Truck #2   3365 S 900 W                               Salt Lake City     UT     84119    EOD
Package 6:     Package en route: Truck #2   3060 Lester St                             West Valley City   UT     84119    10:30 AM
Package 36:    Package en route: Truck #2   2300 Parkway Blvd                          West Valley City   UT     84119    EOD
Package 18:    Package en route: Truck #2   1488 4800 S                                Salt Lake City     UT     84123    EOD
Package 38:    Package en route: Truck #2   410 S State St                             Salt Lake City     UT     84111    EOD
Package 3:     Package en route: Truck #2   233 Canyon Rd                              Salt Lake City     UT     84103    EOD

------------------------------------------------------------------------------------------------------------------------------



------------------------------------------------------------------------------------------------------------------------------
Batch #3 Loaded onto Truck for Optimized Route:
------------------------------------------------------------------------------------------------------------------------------
Truck #3
Driver ID: 1
Departure Time: 09:49 AM
Time on Route: 1.0 minutes
Truck Route Distance Traveled: 0.3 miles
Truck Status: en route

Package 21:    Package en route: Truck #3   3595 Main St                               Salt Lake City     UT     84115    EOD
Package 19:    Package en route: Truck #3   177 W Price Ave                            Salt Lake City     UT     84115    EOD
Package 12:    Package en route: Truck #3   3575 W Valley Central Station bus Loop     West Valley City   UT     84119    EOD
Package 7:     Package en route: Truck #3   1330 2100 S                                Salt Lake City     UT     84106    EOD
Package 2:     Package en route: Truck #3   2530 S 500 E                               Salt Lake City     UT     84106    EOD
Package 4:     Package en route: Truck #3   380 W 2880 S                               Salt Lake City     UT     84115    EOD
Package 17:    Package en route: Truck #3   3148 S 1100 W                              Salt Lake City     UT     84119    EOD
Package 24:    Package en route: Truck #3   5025 State St                              Murray             UT     84107    EOD
Package 26:    Package en route: Truck #3   5383 South 900 East #104                   Salt Lake City     UT     84117    EOD
Package 22:    Package en route: Truck #3   6351 South 900 East                        Murray             UT     84121    EOD
Package 11:    Package en route: Truck #3   2600 Taylorsville Blvd                     Salt Lake City     UT     84118    EOD
Package 23:    Package en route: Truck #3   5100 South 2700 West                       Salt Lake City     UT     84118    EOD
Package 10:    Package en route: Truck #3   600 E 900 South                            Salt Lake City     UT     84105    EOD
Package 5:     Package en route: Truck #3   410 S State St                             Salt Lake City     UT     84111    EOD
Package 9:     Package en route: Truck #3   410 S State St                             Salt Lake City     UT     84111    EOD
Package 8:     Package en route: Truck #3   300 State St                               Salt Lake City     UT     84103    EOD
```

D3. Third Status Check

Truck 1 status shot at 12:15 PM

```
WGUPS-Delivery-Route-Planning   main

main.py
9   # Dayton Abbott
10  # 011125353
11  line_format = ""

main

1) Print All Package Status and Mileage
2) Print Single Package Info by ID
3) Get a Single Package Status at A Given Time
4) Get All Package Statuses at A Given Time
5) Print route data by truck number
6) Exit Program
------------------------------------------------------------------------------------------------------------------------

4
Enter time in format: HH:MM:AM/PM
 (space between time and AM/PM)
12:15 PM

------------------------------------------------------------------------------------------------------------------------
Batch #1 Loaded onto Truck for Optimized Route:
------------------------------------------------------------------------------------------------------------------------
Truck #1
Driver ID: None
Departure Time: 08:00 AM
Time on Route: 107.33333333333334 minutes
Truck Route Distance Traveled: 32.2 miles
Truck Status: Returned to Hub

Package 34:   Delivered: 08:11 AM      4580 S 2300 E              Holladay         UT    84117   10:30 AM     2.0kg
Package 15:   Delivered: 08:11 AM      4580 S 2300 E              Holladay         UT    84117   9:00 AM      4.0kg
Package 29:   Delivered: 08:28 AM      1330 2100 S                Salt Lake City   UT    84106   10:30 AM     2.0kg
Package 33:   Delivered: 08:33 AM      2530 S 500 E               Salt Lake City   UT    84106   EOD          1.0kg
Package 1:    Delivered: 08:38 AM      195 W Oakland Ave          Salt Lake City   UT    84115   10:30 AM    21.0kg
Package 40:   Delivered: 08:42 AM      380 W 2880 S               Salt Lake City   UT    84115   10:30 AM    45.0kg
Package 31:   Delivered: 08:47 AM      3365 S 900 W               Salt Lake City   UT    84119   10:30 AM     1.0kg
Package 35:   Delivered: 09:02 AM      1060 Dalton Ave S          Salt Lake City   UT    84104   EOD         88.0kg
Package 27:   Delivered: 09:02 AM      1060 Dalton Ave S          Salt Lake City   UT    84104   EOD          5.0kg
Package 39:   Delivered: 09:08 AM      2010 W 500 S               Salt Lake City   UT    84104   EOD          9.0kg
Package 13:   Delivered: 09:08 AM      2010 W 500 S               Salt Lake City   UT    84104   10:30 AM     2.0kg
Package 37:   Delivered: 09:18 AM      410 S State St             Salt Lake City   UT    84111   10:30 AM     2.0kg
Package 30:   Delivered: 09:22 AM      300 State St               Salt Lake City   UT    84103   10:30 AM     1.0kg

------------------------------------------------------------------------------------------------------------------------


------------------------------------------------------------------------------------------------------------------------
Batch #2 Loaded onto Truck for Optimized Route:
------------------------------------------------------------------------------------------------------------------------
Truck #2
Driver ID: None
Departure Time: 09:05 AM
Time on Route: 139.0 minutes
Truck Route Distance Traveled: 41.7 miles
Truck Status: Returned to Hub
```

Trucks 2 and 3 status shot at 12:15 PM

WG WGUPS-Delivery-Route-Planning main

ect

WGUPS-Delivery-Route-Planni

.venv library root

entities

main.py

```
# Dayton Abbott
# 011125353
line_format = ""
```

main

```
--------------------------------------------------------------------------------------------------------------------------------------------
Batch #2 Loaded onto Truck for Optimized Route:

--------------------------------------------------------------------------------------------------------------------------------------------
Truck #2
Driver ID: None
Departure Time: 09:05 AM
Time on Route: 139.0 minutes
Truck Route Distance Traveled: 41.7 miles
Truck Status: Returned to Hub

Package 14:   Delivered: 09:11 AM       4300 S 1300 E                            Millcreek          UT     84117    10:30 AM     88.0kg
Package 16:   Delivered: 09:18 AM       4580 S 2300 E                            Holladay           UT     84117    10:30 AM     88.0kg
Package 25:   Delivered: 09:29 AM       5383 South 900 East #104                 Salt Lake City     UT     84117    10:30 AM      7.0kg    De
Package 20:   Delivered: 09:42 AM       3595 Main St                             Salt Lake City     UT     84115    10:30 AM     37.0kg
Package 28:   Delivered: 09:46 AM       2835 Main St                             Salt Lake City     UT     84115    EOD           7.0kg    De
Package 32:   Delivered: 09:56 AM       3365 S 900 W                             Salt Lake City     UT     84119    EOD           1.0kg    De
Package 6:    Delivered: 10:01 AM       3060 Lester St                           West Valley City   UT     84119    10:30 AM     88.0kg    De
Package 36:   Delivered: 10:06 AM       2300 Parkway Blvd                        West Valley City   UT     84119    EOD          88.0kg
Package 18:   Delivered: 10:20 AM       1488 4800 S                              Salt Lake City     UT     84123    EOD           6.0kg
Package 38:   Delivered: 10:55 AM       410 S State St                           Salt Lake City     UT     84111    EOD           9.0kg
Package 3:    Delivered: 10:58 AM       233 Canyon Rd                            Salt Lake City     UT     84103    EOD           2.0kg

--------------------------------------------------------------------------------------------------------------------------------------------


--------------------------------------------------------------------------------------------------------------------------------------------
Batch #3 Loaded onto Truck for Optimized Route:

--------------------------------------------------------------------------------------------------------------------------------------------
Truck #3
Driver ID: 1
Departure Time: 09:49 AM
Time on Route: 146.0 minutes
Truck Route Distance Traveled: 43.8 miles
Truck Status: en route

Package 21:   Delivered: 09:55 AM       3595 Main St                             Salt Lake City     UT     84115    EOD           3.0kg
Package 19:   Delivered: 09:57 AM       177 W Price Ave                          Salt Lake City     UT     84115    EOD          37.0kg
Package 12:   Delivered: 10:02 AM       3575 W Valley Central Station bus Loop   West Valley City   UT     84119    EOD           1.0kg
Package 7:    Delivered: 10:21 AM       1330 2100 S                              Salt Lake City     UT     84106    EOD           8.0kg
Package 2:    Delivered: 10:26 AM       2530 S 500 E                             Salt Lake City     UT     84106    EOD          44.0kg
Package 4:    Delivered: 10:32 AM       380 W 2880 S                             Salt Lake City     UT     84115    EOD           4.0kg
Package 17:   Delivered: 10:39 AM       3148 S 1100 W                            Salt Lake City     UT     84119    EOD           2.0kg
Package 24:   Delivered: 10:55 AM       5025 State St                            Murray             UT     84107    EOD           7.0kg
Package 26:   Delivered: 11:00 AM       5383 South 900 East #104                 Salt Lake City     UT     84117    EOD          25.0kg
Package 22:   Delivered: 11:05 AM       6351 South 900 East                      Murray             UT     84121    EOD           2.0kg
Package 11:   Delivered: 11:27 AM       2600 Taylorsville Blvd                   Salt Lake City     UT     84118    EOD           1.0kg
Package 23:   Delivered: 11:29 AM       5100 South 2700 West                     Salt Lake City     UT     84118    EOD           5.0kg
Package 10:   Delivered: 12:04 PM       600 E 900 South                          Salt Lake City     UT     84105    EOD           1.0kg
Package 5:    Delivered: 12:10 PM       410 S State St                           Salt Lake City     UT     84111    EOD           5.0kg
Package 9:    Delivered: 12:10 PM       410 S State St                           Salt Lake City     UT     84111    EOD             2kg
Package 8:    Delivered: 12:13 PM       300 State St                             Salt Lake City     UT     84103    EOD           9.0kg
```

E. Screenshot of Code Execution

Screenshots (1 of 2) showing all deliveries completed, times, and distances traveled.

```
WGUPS-Delivery-Route-Planning    main

main.py
# Dayton Abbott
# 011125353
line_format = ""

main

1) Print All Package Status and Mileage
2) Print Single Package Info by ID
3) Get a Single Package Status at A Given Time
4) Get All Package Statuses at A Given Time
5) Print route data by truck number
6) Exit Program
--------------------------------------------------------------------------------

1

--------------------------------------------------------------------------------
Batch #1 Loaded onto Truck for Optimized Route:
--------------------------------------------------------------------------------
Truck #1
Driver ID: None
Departure Time: 08:00 AM
Time on Route: 107.33333333333334 minutes
Truck Route Distance Traveled: 32.2 miles
Truck Status: Returned to Hub

Package 34:    Delivered: 08:11 AM    4580 S 2300 E          Holladay         UT    84117    10:30 AM    2.0kg
Package 15:    Delivered: 08:11 AM    4580 S 2300 E          Holladay         UT    84117    9:00 AM     4.0kg
Package 29:    Delivered: 08:28 AM    1330 2100 S            Salt Lake City   UT    84106    10:30 AM    2.0kg
Package 33:    Delivered: 08:33 AM    2530 S 500 E           Salt Lake City   UT    84106    EOD         1.0kg
Package 1:     Delivered: 08:38 AM    195 W Oakland Ave      Salt Lake City   UT    84115    10:30 AM    21.0kg
Package 40:    Delivered: 08:42 AM    380 W 2880 S           Salt Lake City   UT    84115    10:30 AM    45.0kg
Package 31:    Delivered: 08:47 AM    3365 S 900 W           Salt Lake City   UT    84119    10:30 AM    1.0kg
Package 35:    Delivered: 09:02 AM    1060 Dalton Ave S      Salt Lake City   UT    84104    EOD         88.0kg
Package 27:    Delivered: 09:02 AM    1060 Dalton Ave S      Salt Lake City   UT    84104    EOD         5.0kg
Package 39:    Delivered: 09:08 AM    2010 W 500 S           Salt Lake City   UT    84104    EOD         9.0kg
Package 13:    Delivered: 09:08 AM    2010 W 500 S           Salt Lake City   UT    84104    10:30 AM    2.0kg
Package 37:    Delivered: 09:18 AM    410 S State St         Salt Lake City   UT    84111    10:30 AM    2.0kg
Package 30:    Delivered: 09:22 AM    300 State St           Salt Lake City   UT    84103    10:30 AM    1.0kg

--------------------------------------------------------------------------------


--------------------------------------------------------------------------------
Batch #2 Loaded onto Truck for Optimized Route:
--------------------------------------------------------------------------------
Truck #2
Driver ID: None
Departure Time: 09:05 AM
Time on Route: 139.0 minutes
Truck Route Distance Traveled: 41.7 miles
Truck Status: Returned to Hub

Package 14:    Delivered: 09:11 AM    4300 S 1300 E            Millcreek        UT    84117    10:30 AM    88.0kg
Package 16:    Delivered: 09:18 AM    4580 S 2300 E            Holladay         UT    84117    10:30 AM    88.0kg
Package 25:    Delivered: 09:29 AM    5383 South 900 East #104 Salt Lake City   UT    84117    10:30 AM    7.0kg
Package 20:    Delivered: 09:42 AM    3595 Main St             Salt Lake City   UT    84115    10:30 AM    37.0kg
```

Screenshots (2 of 2) showing all deliveries completed, times, and distances traveled.

```
WGUPS-Delivery-Route-Planning   main
main.py
9   # Dayton Abbott
10  # 011125353
11  line_format = ""

main

Truck Route Distance Traveled: 41.7 miles
Truck Status: Returned to Hub

Package 14:   Delivered: 09:11 AM      4300 S 1300 E                           Millcreek         UT    84117    10:30 AM
Package 16:   Delivered: 09:18 AM      4580 S 2300 E                           Holladay          UT    84117    10:30 AM
Package 25:   Delivered: 09:29 AM      5383 South 900 East #104                Salt Lake City    UT    84117    10:30 AM
Package 20:   Delivered: 09:42 AM      3595 Main St                            Salt Lake City    UT    84115    10:30 AM
Package 28:   Delivered: 09:46 AM      2835 Main St                            Salt Lake City    UT    84115    EOD
Package 32:   Delivered: 09:56 AM      3365 S 900 W                            Salt Lake City    UT    84119    EOD
Package 6:    Delivered: 10:01 AM      3060 Lester St                          West Valley City  UT    84119    10:30 AM
Package 36:   Delivered: 10:06 AM      2300 Parkway Blvd                       West Valley City  UT    84119    EOD
Package 18:   Delivered: 10:20 AM      1488 4800 S                             Salt Lake City    UT    84123    EOD
Package 38:   Delivered: 10:55 AM      410 S State St                          Salt Lake City    UT    84111    EOD
Package 3:    Delivered: 10:58 AM      233 Canyon Rd                           Salt Lake City    UT    84103    EOD

---------------------------------------------------------------------------------------------------------------------



---------------------------------------------------------------------------------------------------------------------
Batch #3 Loaded onto Truck for Optimized Route:
---------------------------------------------------------------------------------------------------------------------
Truck #3
Driver ID: None
Departure Time: 09:49 AM
Time on Route: 170.0 minutes
Truck Route Distance Traveled: 51.0 miles
Truck Status: Returned to Hub

Package 21:   Delivered: 09:55 AM      3595 Main St                            Salt Lake City    UT    84115    EOD
Package 19:   Delivered: 09:57 AM      177 W Price Ave                         Salt Lake City    UT    84115    EOD
Package 12:   Delivered: 10:02 AM      3575 W Valley Central Station bus Loop  West Valley City  UT    84119    EOD
Package 7:    Delivered: 10:21 AM      1330 2100 S                             Salt Lake City    UT    84106    EOD
Package 2:    Delivered: 10:26 AM      2530 S 500 E                            Salt Lake City    UT    84106    EOD
Package 4:    Delivered: 10:32 AM      380 W 2880 S                            Salt Lake City    UT    84115    EOD
Package 17:   Delivered: 10:39 AM      3148 S 1100 W                           Salt Lake City    UT    84119    EOD
Package 24:   Delivered: 10:55 AM      5025 State St                           Murray            UT    84107    EOD
Package 26:   Delivered: 11:00 AM      5383 South 900 East #104                Salt Lake City    UT    84117    EOD
Package 22:   Delivered: 11:05 AM      6351 South 900 East                     Murray            UT    84121    EOD
Package 11:   Delivered: 11:27 AM      2600 Taylorsville Blvd                  Salt Lake City    UT    84118    EOD
Package 23:   Delivered: 11:29 AM      5100 South 2700 West                    Salt Lake City    UT    84118    EOD
Package 10:   Delivered: 12:04 PM      600 E 900 South                         Salt Lake City    UT    84105    EOD
Package 5:    Delivered: 12:10 PM      410 S State St                          Salt Lake City    UT    84111    EOD
Package 9:    Delivered: 12:10 PM      410 S State St                          Salt Lake City    UT    84111    EOD
Package 8:    Delivered: 12:13 PM      300 State St                            Salt Lake City    UT    84103    EOD

---------------------------------------------------------------------------------------------------------------------

TOTAL DELIVERY MILEAGE:  124.9 miles

TOTAL DRIVER TIME (CONCURRENT):  6 hours and 56 minutes
```


Table of Contents

F1. Strengths of the Chosen Algorithm

There are two main benefits to the nearest neighbor greedy algorithm that I used for this program: the simplicity of the implementation and the speed of the algorithm compared to alternatives.

With regards to both the speed and the simplicity of this algorithm, a more complex algorithm that iterates through all possible destinations in different sequences in order to find the shortest route would require more logical complexity to implement, and it would have a higher time complexity. The scan of a distance matrix for the nearest neighbor algorithm has a worst case time complexity of $O(N^2)$. An alternative like a brute force algorithm that checks all possible permutations would require $O(N!)$ time (Salonen).

F2. Verification of Algorithm

As is evident in the section E screenshots, the algorithm is able to generate routes that meet the delivery deadline criteria for all packages. The deadline constraints are met while using the constant truck speed of 18mph, which is assigned in *delivery_service.py* for the travel time calculations, and by assigning priority packages to earlier truck routes or by prioritizing the higher priority packages within a batch for earlier delivery.

The maximum mileage constraint is met by applying nearest neighbor the heuristic to the delivery routes to achieve an ideal aggregate travel distance. The total distance traveled by all trucks for all deliveries is 124.9 miles, including the mileage to return to the Hub after deliveries are complete.

The constraints for truck capacities are also met by this algorithm, as a truck reaches capacity while packages are being assigned to it, loading can be paused and then resumed on the next available truck. My algorithm implementation ensures that no truck is loaded with more than 16 packages.

Truck 1 leaves the Hub at exactly 8:00 AM. Truck 2 contains all of the packages that can only be delivered by truck 2 as well as all of the packages that must be delivered with other packages. Truck two leaves the Hub at 9:05 AM carrying the delayed packages that do not arrive at the depot until 9:05 AM.

The address for package 9 is corrected and the delivery to the correct address is not made until 12:10PM, after the correct address becomes available at 10:20AM. Truck 3 does not leave the hub until 9:49, after driver 1 returns from the delivery route for truck 1. This means that only two drivers are operating the three trucks throughout the delivery simulation and at most two trucks are on route at a given time.

All packages are loaded onto trucks at the hub prior to departure.

F3. Other Possible Algorithms

Two other possible algorithms for solving the delivery routing problem are the Clarke and Wright Savings algorithm and the Nearest Insertion Heuristic. While implementation of these algorithms differs, they are all greedy algorithms.

F3a. Algorithm Differences

The Clarke and Wright Savings algorithm differs from the Nearest Neighbor algorithm that I used in a few ways. A Clarke and Wright algorithm begins by creating individual routes for every delivery (Tunnisaki and Sutarman). These routes start at the Hub, go to the destination, and return to the Hub. In order to create an optimized combination of these “mini-routes”, the algorithm combines nodes based on the maximum “distance savings” value of traveling from one node to another node rather than returning to the hub, as long as combining nodes would not violate delivery constraints (Tunnisaki and Sutarman).

The Nearest Insertion Heuristic differs in the following ways. It initially selects two nodes from the destination nodes (“Some Important Heuristics for the TSP”). In the case of the delivery routing problem, the two start nodes would both be the delivery Hub. Next, the algorithm would search for the nearest neighbor to the Hub and insert that between the start/end nodes. From this point, the algorithm selects the nearest unvisited destination node whose distance to any node in the route is the smallest. The route then checks for the optimal position to insert this node based on which position would increase the total distance of the route by the least amount (“Some Important Heuristics for the TSP”).

G. Different Approach

I think that if I were to do this project again, the implementation could be refined by combining various components and/or steps to reduce the complexity of the program. While the batch-specific hash tables and distance matrices provide a level of separation that initially seemed valuable for preventing errors in route data and distances, in hindsight, it does not seem necessary to have generated these separate data structures. The primary distance tables and routes could have been used with condition checks to exclude addresses or packages that are not included in a given batch. Eliminating the functions for creating these data structures would have greatly reduced the overhead in developing the program and would have reduced the memory required for the program as well as the execution time. That being said, I think it was a valuable experience for me to learn about data processing through dynamically generating these structures.

I also would have liked to implement a collision resolution strategy for the hash table or just use a built-in hash table structure from Python that includes collision resolution to ensure that package data are not being overwritten due to some error in the program logic.

The other main thing that I would change would be to persist the package and route data in a database. For a theoretical approach for a course project, the data structures used for containing the data are sufficient. However, if I were creating this program as an actual delivery service, I think that the architecture and access of data would be much cleaner if these data were managed by a relational database rather than just being contained in data structures that are passed around between functions.

H. Verification of Data Structure

The data structure meets the project requirement for a self-adjusting data structure because the size of the package hash table is dynamic in that the size of the table changes in relation to the input size, i.e. the number of packages.

The hash table also includes an insertion function for inserting packages into the structure, as well as look-up functions that use either the package ID or an address as inputs in order to return the corresponding packages' information. These insertion and look-up functions retrieve all of the specified information including package ID, delivery address, delivery deadline, delivery city, delivery zip code, package weight, and delivery status. They also retrieve the package notes.

H1. Other Data Structures

Two alternative data structures are a list of lists, and a balanced binary tree like an AVL tree.

H1a. Data Structure Differences

Since the package IDs are numbered sequentially, a list of lists would have worked as an alternative data structure. The package ID could be inserted into the corresponding index position by subtracting 1 from all IDs in order to insert them into the list. Then the attributes of each package could be identified by their index position within each list. This would result in a constant time complexity for access by index, similar to the hash table, although not as well suited to the purpose as this would involve multiple abstractions of keys for accessing data, which could get confusing to try to write the code for.

Another alternative would be a balanced binary search tree, such as an AVL tree. Each node in the tree could be a package ID/key with a pointer to the package data. An AVL or other balanced binary tree would have slower search times for packages though, because searching a binary search tree takes an amount of time that is proportional to its height and balanced trees are kept at a height of O(logn) (Lysecky et al., 6.1). Meaning that the search time of a BST is also O(logn), which is greater than constant time O(1) for hash tables or searching lists by index (Lysecky et al., 6.1).

I. Sources

Lysecky, Roman, Frank Vahid, and Evan Olds. C949: Data Structures and Algorithms I. zyBooks, 2022. ISBN: 979-8-203-31094-1. https://learn.zybooks.com/zybook/WGUC949DataStructuresv4/.

Salonen, Antti. "4.4.2. NP-hard Problems." *Learn Programming 1.0*, 2018, https://progbook.org/tsp.html. Accessed 18 May 2025.

"Some Important Heuristics for the TSP." *Logistical and Transportation Planning Methods*, lecture 16, MIT OpenCourseWare, 27 Nov. 2002, https://ocw.mit.edu/courses/1-203j-logistical-and-transportation-planning-methods-fall-2006/03634d989704c2607e6f48a182d455a0_lec16.pdf. Accessed 19 May 2025.

Tunnisaki, Fadlah, and Sutarman. "Clarke and Wright Savings Algorithm as Solutions Vehicle Routing Problem with Simultaneous Pickup Delivery (VRPSPD)." *Journal of Physics: Conference Series*, vol. 2421, no. 1, IOP Publishing, Jan. 2023, p. 012045. https://doi.org/10.1088/1742-6596/2421/1/012045. Accessed 19 May 2025.