WGUPS Routing Program

Dalton J Riley

WGU

C950 Data Structures and Algorithms 2

Sidney Ruby

7/7/2022

# 

# WGUPS Routing System

# Section A

# Algorithm Identification

For this program, I chose the nearest neighbor algorithm to complete the packages' delivery.

**Section B**

# Logic Comments

Set current address variable with current address

Set shortest distance variable to a value larger than every distance

Initiliaze closest address variable

Initialize closest\_id variable

Loop through all packages on a truck

Set next address variable containing the current package’s address

Set a distance check variable containing the distance from the current address to next address

If the value of distance check is less than or equal to the shortest distance variable

Set shortest distance with value of distance check

Set closest address variable with the value from next address

Set closest id variable with the id of the current package

Return shortest distance, closest address, and closest id

# 

# Development Environment

The software used was Python 3.9.2 PyCharm 2021.2.3 and the hardware used was my local machine, an Asus ROG Strix Scar 15 running Windows 10.

# 

# Space-Time and Big-O

This can be found within the comments of the program.

# Scalability and Adaptability

The two scalable elements in my project are the self-adjusting algorithm (deliver\_packages()) and the hashtable. Both can take N amount of variables and I believe both will scale reasonably well. Since the hashtable runs in polynomial time and the algorithm runs in exponential time, the hash table is more scalable. The self-adjusting algorithm would be much more scalable if the complexity were taken down to O(N).

# 

# Software Efficiency and Maintainability

My program’s space-time complexity is O(N^2). Since it runs in exponential time it is not efficient. The program is somewhat maintainable. The functions and variables are clearly named and I have written comments explaining the purpose of all major sections. In the future, I could make it more maintainable by compartmentalizing the functions into classes with their own files. This would make the structure of the program much more understandable and it would clean up the main.py file.

# 

# Self-Adjusting Data Structures

The self-adjusting data structure in this program is the hash table. It can take n amount of elements and organize them into respective buckets. It can also search for a specific element and return that element efficiently. The fact that we can have an unlimited amount of items in the hashtable and that we are able to search for them efficiently are some of the strengths. The following are weaknesses I would like to improve in the future: creating an update function to manipulate the data once inside. It could also be improved if a function would allow seeing the entire hashtable without having to call each element, one at a time.

# 

**Section C**

These requirements can be found within the program.

**Section D**

**Explanation of Data Structure**

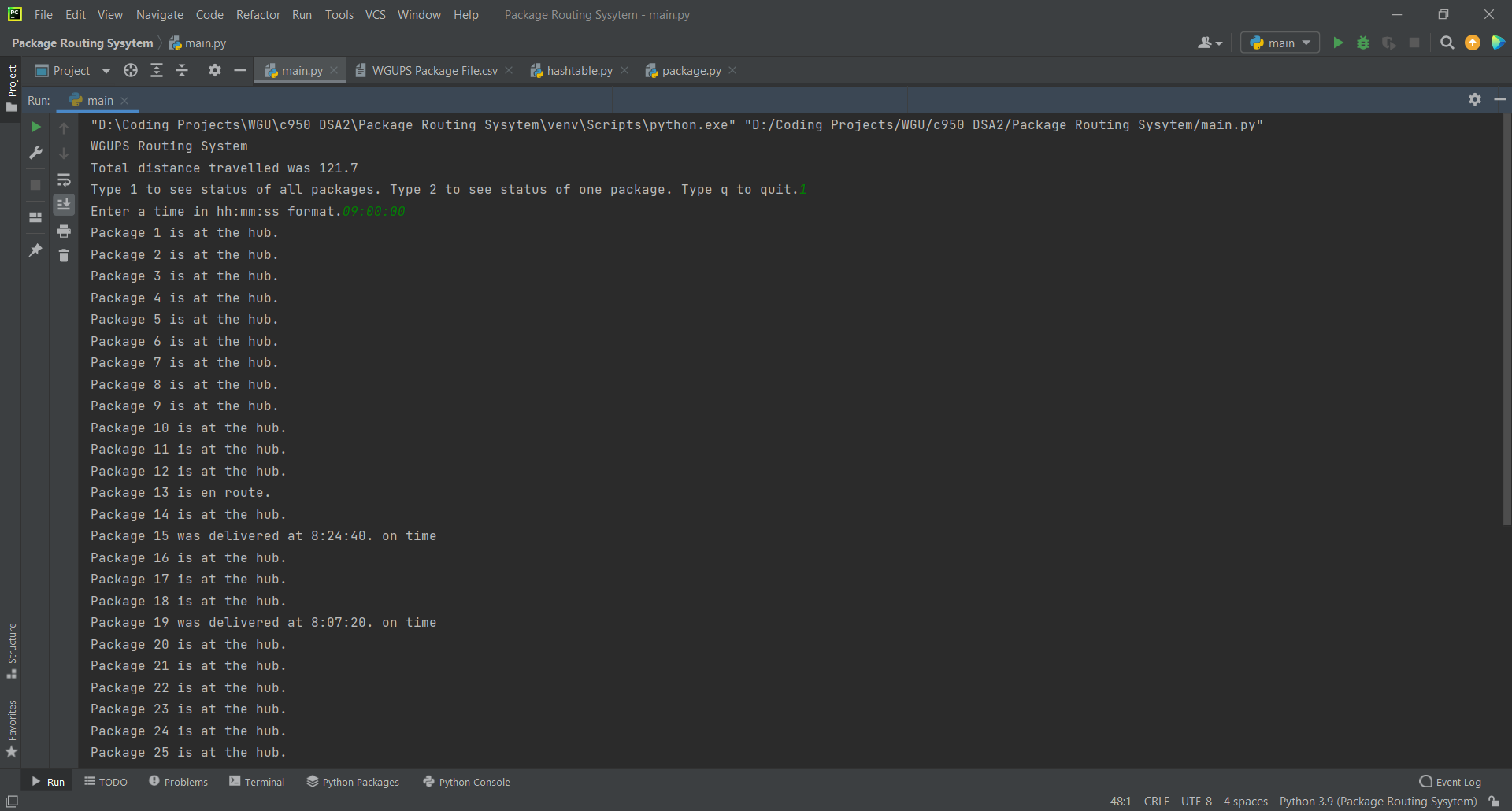
The data structure that I used was a hash table. It creates a table with a specified amount of buckets. If an amount isn’t specified, 10 buckets are created. It places item-key (package object is item, package id is key) pairs into the buckets using the modulo function. The result of key modulo 10 is the bucket it is placed in, appending the list. It can then use the key to search or remove the desired item.

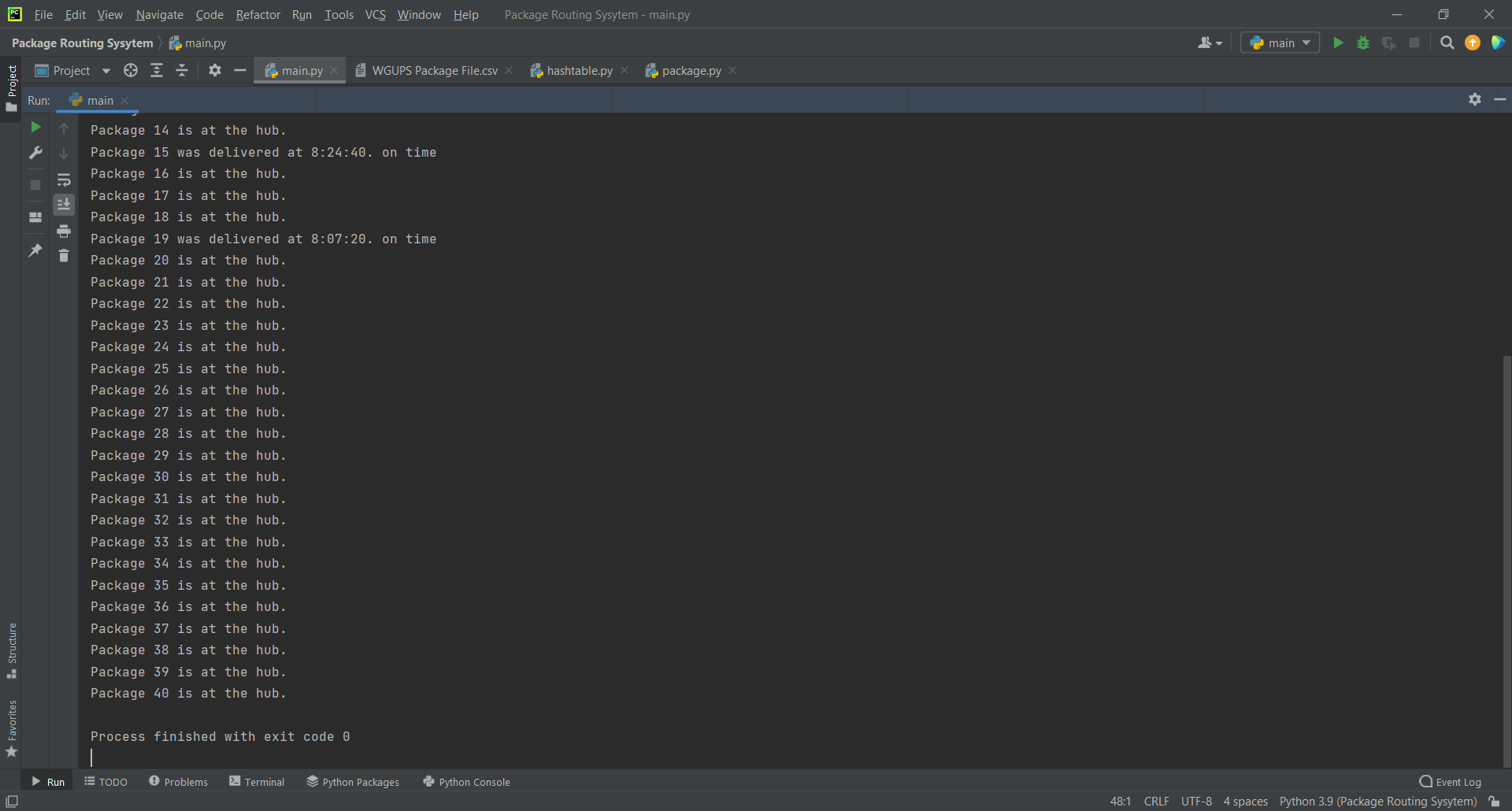
**Sections E and F**

These requirements can be found within the program.

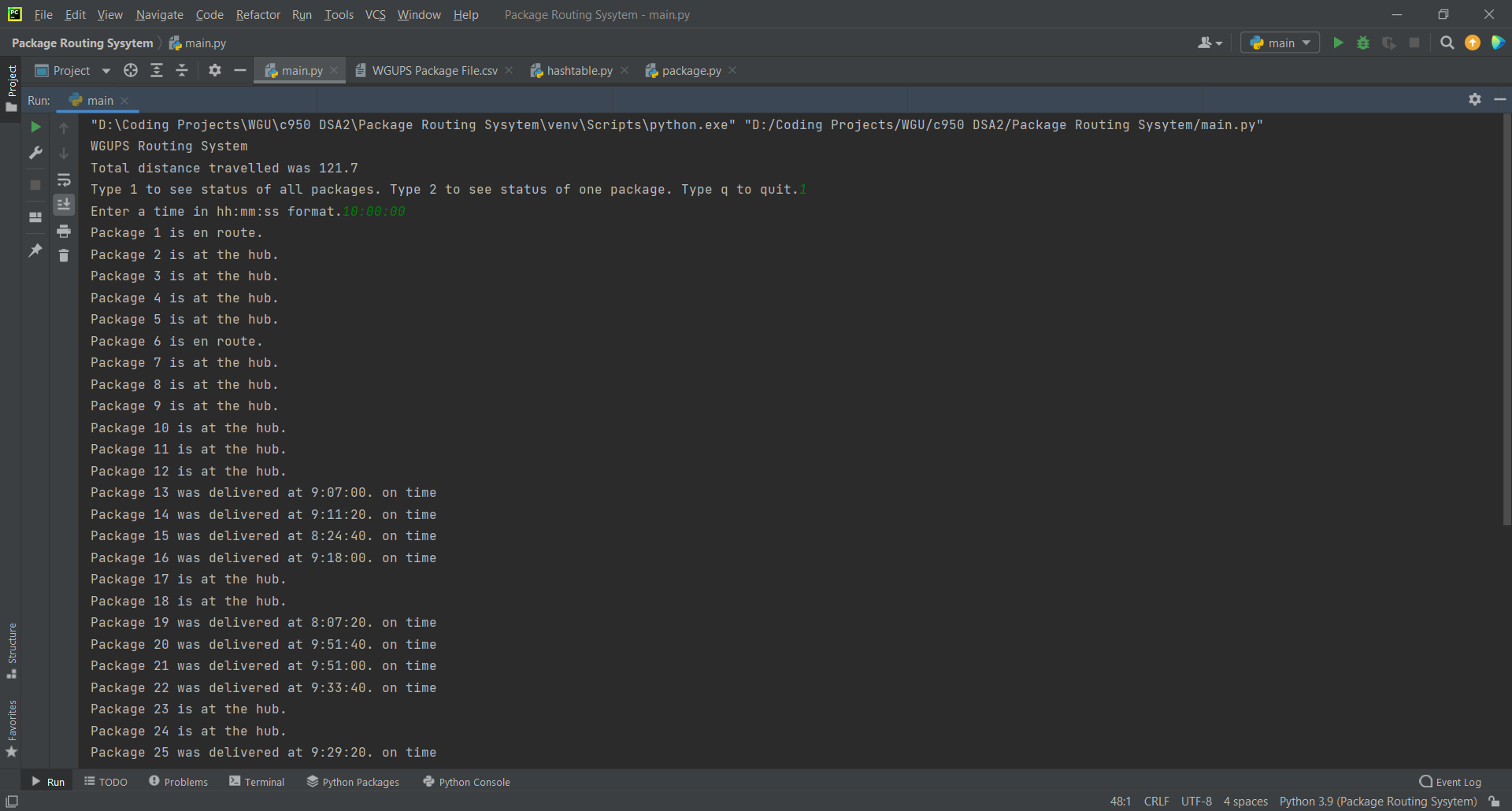
**Section G**

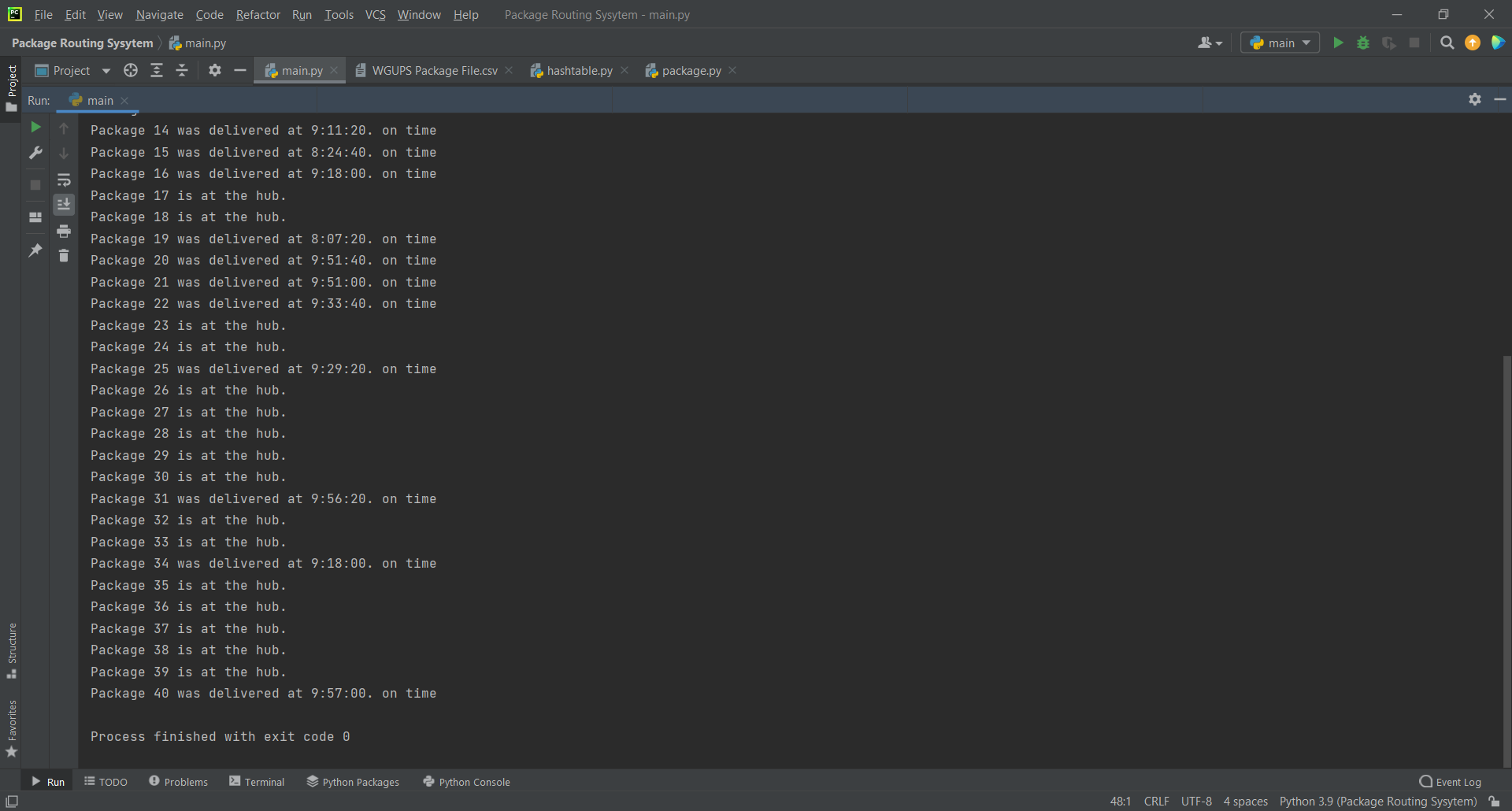
**First Status Check**

****

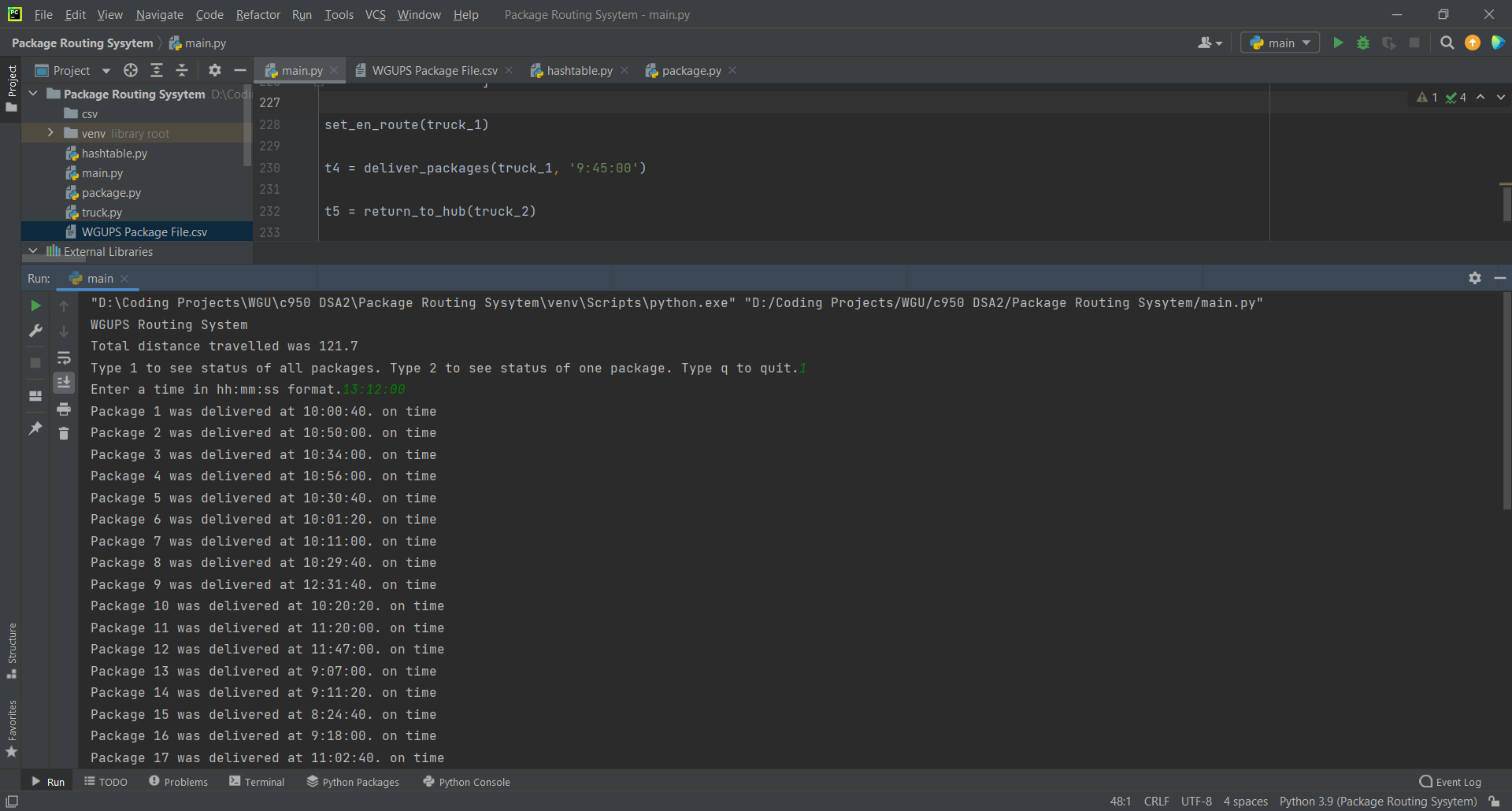
****

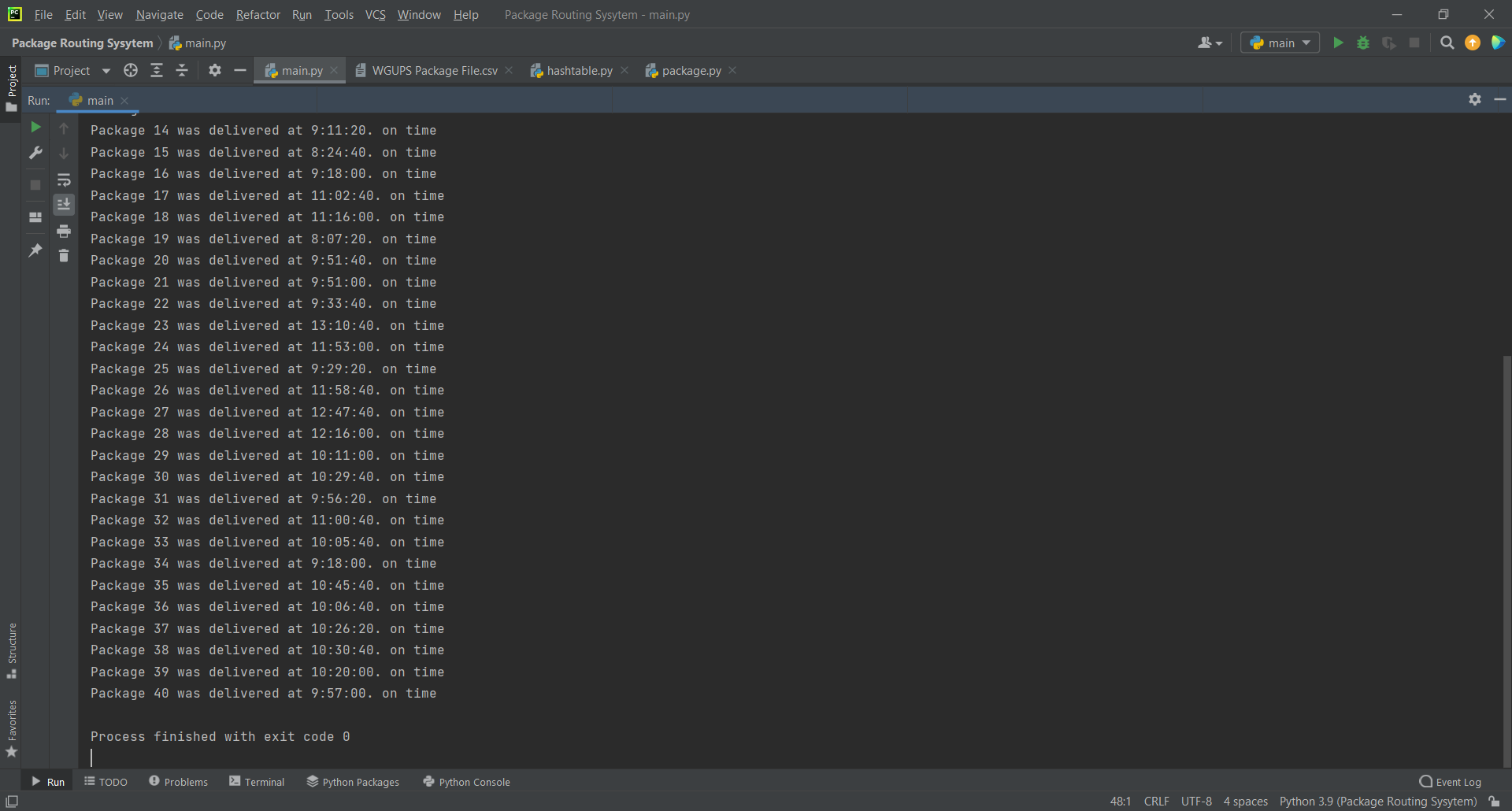
**Second Status Check**

****

****

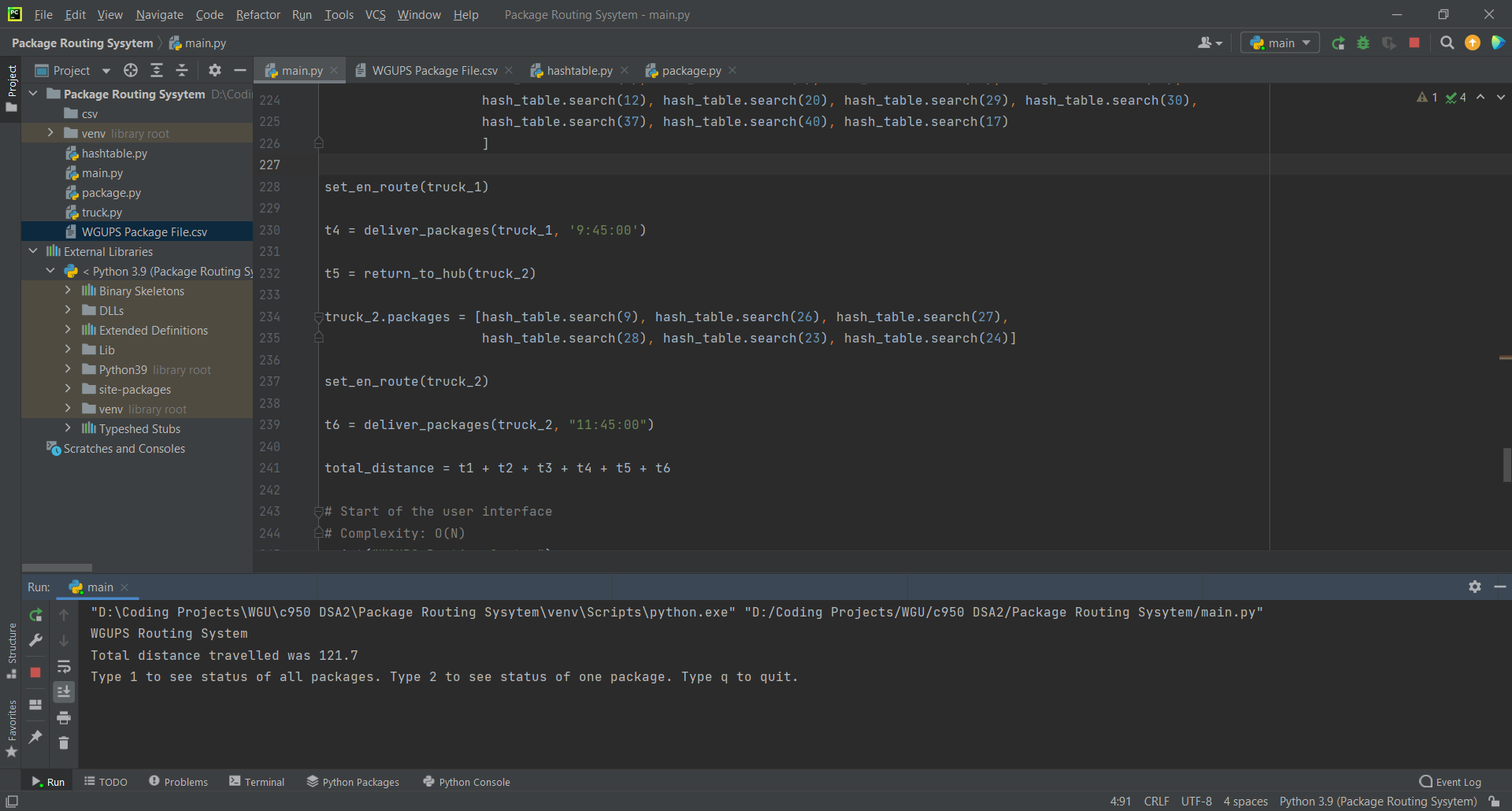
**Third Status Check**

****

****

**Section H**

**Code Execution**

****

**Section I**

**Strengths of Chosen Algorithm**

One strength of the algorithm I chose is that it is adaptable. It can n amount of packages and deliver them effectively. Another strength is that it keeps track of the time packages were loaded and delivered, as well as the distance traveled for each package.

**Verification of Algorithm**

The total miles traveled by all trucks using this algorithm was 121.7. All packages were delivered on time and within their specifications. This can be found by using the user interface or by looking at the screenshots above.

**Other Possible Algorithms**

Two other algorithms that would be possible are nearest insertion and random insertion.

***Algorithm Differences***

The nearest insertion algorithm starts with two locations and adds the closest location next to them making a route. It then continues finding the closest location and adding them to the route so that the route remains as short as it can be until no locations are left. This algorithm would find a more optimal solution and the complexity is still O(N^2). The random insertion algorithm starts with two locations and randomly adds other locations into the route until non are left. This algorithm might produce a more optimal solution or might not because of its random nature. The time complexity is also O(N^2).

**Section J**

**Different Approach**

I would change the way I loaded the distance and address files, and the way I loaded the trucks. For ease, I loaded everything manually, but this is not sustainable if it were to scale larger. I would create functions to read the CSV files and automatically load the contents of them into lists. I would also make a function that takes a truck as a parameter and automatically loads packages until the truck is full, checking package constraints along the way.

**Section K**

**Verification of Data Structure**

The total combined miles for all trucks was 121.7. All packages were delivered on time and according to package specifications. An efficient hashtable that can look up items is present and the package status and info can be found through the user interface or the screenshots above.

***Efficiency***

The look-up function for the hash table has a complexity of O(N). Adding more items will affect the time linearly. If it takes one second with one item, it will take 10 seconds with 10 items.

***Overhead***

The same is true of space complexity. The look-up function runs in O(N) so the space required is directly affected by the number of items being stored.

***Implications***

In this program adding additional trucks and cities would have no impact on the hashtable. The only thing stored in the hashtable is the packages.

**Other Data Structures**

Another possible data structure could be a hashtable with an update function. Another could be a python dictionary.

***Data Structure Differences***

The difference for the hashtable with the update function would be that we could make the hashtable private, preventing manipulation without a method being used. This would also make the code cleaner, having a method call instead of directly changing the package objects.

The difference with using a dictionary would be that the key-item pairs would be stored in one large list instead of different buckets. This would decrease scalability.

# 

# References

WGU. (2020, November 17). *Lets Go Hashing* [Video]. Panopto. <https://wgu.hosted.panopto.com/Panopto/Pages/Viewer.aspx?id=f08d7871-d57a-496e-a6a1-ac7601308c71>

WGU. *C950 WGUPS Project Implementation Steps - Example*

https://srm--c.na127.visual.force.com/apex/coursearticle