**Custom Package Routing CLI Program for WGUPS**

*Python Version 3.8*

By Sidak Sohi

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# Algorithm Overview

# Stated Problem

# Using the latest version of Python, use an algorithm to create a route for various trucks at ‘WGUPS’. The algorithm should result in a total distance of less than 145 miles and deliver all 40 packages on time, according to each package’s specifications. The output of this algorithm is a ‘route’, in which the driver both starts and ends at the hub. Each element in the route is a list, like so: [‘Distance’, \*\*‘Package ID’, ‘Start Location’, ‘End Location]

# *\*\*The last element in the route ends at the hub, and thus has ‘None’ as the Package ID*

1. **Algorithm Overview**

I created my algorithm from scratch, and did not reference or copy/paste any pathfinding algorithms.

Although I knew of various pathfinding algorithms, I decided to make my own for a couple reasons: 1) To see if I could make a functioning algorithm in the first place, and 2) to see which known algorithm mine would resemble and how I could have further optimized it.

My algorithm is very simple, but it got the job done. Here’s how it works, given a list of packages:

1. Find the package in the list which is closest to the hub
2. Remove that package from the list, add it to the route
3. Find the package, of packages remaining, that is closest to the last package in the route
4. Repeat steps 2, 3 until there is only one package left
5. For the last package, find the distance to the hub and add it to the route

On review of the algorithm and looking at algorithms, I found that my algorithm very closely resembles the 1st nearest neighbor algorithm. In essence, the algorithm takes an input of a point on a graph, and will find you the point nearest to the input, which is exactly what I am doing. In my algorithm, it repeats the 1st nearest neighbor for the entire list of packages, with the starting and ending locations for the route both being the hub. Here is an example of an input and output of my algorithm:

*Input; a list of packages [‘Package ID’, ‘Address’, ‘City’, ‘State’, ‘Zip’, …]:*

['3', '233 Canyon Rd', 'Salt Lake City', 'UT', '84103', 'EOD', '2', 'Can only be on truck 2', 'AT\_HUB']

['13', '2010 W 500 S', 'Salt Lake City', 'UT', '84104', '10:30:00', '2', 'None', 'AT\_HUB']

['14', '4300 S 1300 E', 'Millcreek', 'UT', '84117', '10:30:00', '88', 'Must be delivered with 15 & 19', 'AT\_HUB']

['15', '4580 S 2300 E', 'Holladay', 'UT', '84117', '9:00:00', '4', 'None', 'AT\_HUB']

['16', '4580 S 2300 E', 'Holladay', 'UT', '84117', '10:30:00', '88', 'Must be delivered with 13 & 19', 'AT\_HUB']

['19', '177 W Price Ave', 'Salt Lake City', 'UT', '84115', 'EOD', '37', 'None', 'AT\_HUB']

['20', '3595 Main St', 'Salt Lake City', 'UT', '84115', '10:30:00', '37', 'Must be delivered with 13 & 15', 'AT\_HUB']

['36', '2300 Parkway Blvd', 'West Valley City', 'UT', '84119', 'EOD', '88', 'Can only be on truck 2', 'AT\_HUB']

['38', '410 S State St', 'Salt Lake City', 'UT', '84111', 'EOD', '9', 'Can only be on truck 2', 'AT\_HUB']

['1', '195 W Oakland Ave', 'Salt Lake City', 'UT', '84115', '10:30:00', '21', 'None', 'AT\_HUB']

['29', '1330 2100 S', 'Salt Lake City', 'UT', '84106', '10:30:00', '2', 'None', 'AT\_HUB']

['30', '300 State St', 'Salt Lake City', 'UT', '84103', '10:30:00', '1', 'None', 'AT\_HUB']

['31', '3365 S 900 W', 'Salt Lake City', 'UT', '84119', '10:30:00', '1', 'None', 'AT\_HUB']

['34', '4580 S 2300 E', 'Holladay', 'UT', '84117', '10:30:00', '2', 'None', 'AT\_HUB']

['37', '410 S State St', 'Salt Lake City', 'UT', '84111', '10:30:00', '2', 'None', 'AT\_HUB']

['40', '380 W 2880 S', 'Salt Lake City', 'UT', '84115', '10:30:00', '45', 'None', 'AT\_HUB']

*Output; a route. [‘Distance’, ‘Package ID’, ‘Start Location’, ‘End Location’]:*

[1.9, '14', 'Western Governors University', '4300 S 1300 E']

[2.0, '15', '4300 S 1300 E', '4580 S 2300 E']

[0.0, '16', '4580 S 2300 E', '4580 S 2300 E']

[0.0, '34', '4580 S 2300 E', '4580 S 2300 E']

[5.0, '20', '4580 S 2300 E', '3595 Main St']

[0.5, '19', '3595 Main St', '177 W Price Ave']

[1.7, '40', '177 W Price Ave', '380 W 2880 S']

[1.1, '1', '380 W 2880 S', '195 W Oakland Ave']

[2.8, '29', '195 W Oakland Ave', '1330 2100 S']

[4.3, '37', '1330 2100 S', '410 S State St']

[0.0, '38', '410 S State St', '410 S State St']

[1.0, '3', '410 S State St', '233 Canyon Rd']

[0.6, '30', '233 Canyon Rd', '300 State St']

[4.2, '13', '300 State St', '2010 W 500 S']

[4.0, '36', '2010 W 500 S', '2300 Parkway Blvd']

[5.8, '31', '2010 W 500 S', '3365 S 900 W']

[3.7, None, '3365 S 900 W', 'Western Governors University']

1. **Operation Time**

Operation time for the algorithm varies, as in this project I was not able to use a dictionary to store the packages. If I was able to use a dictionary to store packages, the algorithm’s worst case time complexity would be drastically reduced. This is because searching for a value in a list vs a dictionary is O(n) vs O(1). There are many instances in my algorithm which involve iterating through a data structure to find a certain value. Overall, the worst case time complexity for this algorithm came out to O(n^4), which is not ideal.

1. **Pseudo Code Analysis**

**Given a start location and end location, find the distance between the two locations**

**Returns the distance, package id, start location, and end location**

**Worst case time complexity: O(n^2)**

**Space Complexity: O(n)**

distance(start, end, id):

packageDistanceItem = []

search\_check = False

for i in distance\_dict.keys():

if start in i:

for index, item in enumerate(distance\_dict[i]):

if end in str(item):

search\_check = True

packageDistanceItem.append(distance\_dict[i].\_\_getitem\_\_(index + 1))

packageDistanceItem.append(id)

packageDistanceItem.append(start)

packageDistanceItem.append(end)

return packageDistanceItem.copy()

if not search\_check:

for i in distance\_dict.keys():

if end in i:

for index, item in enumerate(distance\_dict[i]):

if start in str(item):

packageDistanceItem.append(distance\_dict[i].\_\_getitem\_\_(index + 1))

packageDistanceItem.append(id)

packageDistanceItem.append(start)

packageDistanceItem.append(end)

return packageDistanceItem

**Given a list of packages, this method will find the distances to each package from the hub**

**It will then sort the packages by descending order, and then return the list**

**Using pop() on the returned list will get you the package closest to the hub.**

**Worst case time complexity: O(n^3)**

**Space Complexity: O(n)**

closestPackageFromHub(packages, start):

routeHiToLow = []

for index, item in enumerate(packages):

routeHiToLow.append(distance(start, item[1], item[0]))

routeHiToLow.sort(reverse=True)

return routeHiToLow

**This method is very similar to the one above it, however instead of finding the distance to each package from the hub, it finds the shortest distance to the next package, from the given package's delivery address**

**Worst case time complexity: O(n^3)**

**Space Complexity: O(n)**

closestPackage(packages, start):

routeHiToLow = []

for index, item in packages:

routeHiToLow.append(distance(start, item[3], item[1]))

routeHiToLow.sort(reverse=True)

return routeHiToLow

**Using the three methods above, this method takes a list of unsorted packages as input, and returns a 'route'**

**First, it finds the package closest to the hub**

**Then, it uses that package's delivery address to find the package closest to that**

**It repeats the above step with the remaining packages**

**Once there is only one package, it will find the distance from the last package to the hub, creating a 'route'**

**Worst case time complexity: O(n^4)**

**Space Complexity: O(n)**

route(packages):

sorted = []

sortedTemp = []

for index, item in packages:

if index == 0:

sortedTemp = closestPackageFromHub(packages, 'Western Governors University')

sorted.append(sortedRouteTemp.pop())

if 0 < index < (length(packages)-1):

sortedTemp = closestPackage(sortedTemp, packages[index-1][3])

sorted.append(sortedTemp.pop())

if index == (length(packages)-1):

sorted.append(sortedTemp.pop())

sorted.append(distance((packages[index][3]), 'Western Governors University', None))

return sorted

**Application of Programming Models**

The data for this program is stored locally on .csv files. Since everything is run locally, a communication protocol does not exist. There are also no interaction semantics, due to everything being run locally; the local machine does not have to interact with anything else in order to run the program. The target host environment is PyCharm, with Python 3.8.

**Adaptability**

The algorithm used for this program has a worst case time complexity of O(n^4). Because of this, it is not very scalable and cannot handle a growing amount of data in certain situations. In this scenario, the trucks could only hold a maximum of 16 packages at a time, and the distance table was stored using a dictionary. Because the input has a constraint of a maximum of 16 packages, and the data structure used to find the distance has a very fast access time, it could hypothetically scale quite well. But, it would have to be in accordance with the 16 package limit, i.e.: 100 lists of 16 packages -> 1000 routes is fine, but 1 list of 1600 packages -> 1 huge route would not be efficient, due to the algorithm's time complexity. In conclusion, it can scale well for this particular company and their constraints for delivering packages, assuming it doesn’t change. If WGUPS were to change their truck capacity, then it would not scale well.

**Software Efficiency and Maintainability**

The software’s efficiency is almost non-existent in general terms, due to the algorithm's shortcomings in regards to its time complexity of O(n^4). But specifically for this company, it is efficient *enough* to work, all thanks to their constraints which limit the strain on the algorithm.

Maintainability wise, the program is short and sweet, and includes detailed comments on the flow and logic of the program. Due to this, it would be simple to maintain by programmers other than myself. The program also has a logical structure and it would be easy to locate code when necessary.

**Self-Adjusting Data Structures**

The data structure we use for the list of packages is a hash table, specifically one where we use a list of lists for each package. In python, a list is represented as an array, so adding to the end of the list is O(1), meaning it could easily adapt if the operator needed to add packages to the hash table. Removing a package would take a lot longer however, due to every element after the deleted element having to be moved back, which would result in a time complexity of O(n). In our program, there are many cases in which we have to iterate through the hash table to find a specific element, and iteration in a list has a time complexity of O(n). If the list of packages were to hypothetically increase by a multitude of 1,000, then it would take much longer to iterate through the list each time we want to find an element. For this program specifically, it seems that the program only takes packages that are limited to the same city and day, which means that the list of packages would never be too high.

**Explanation of Data Structure**

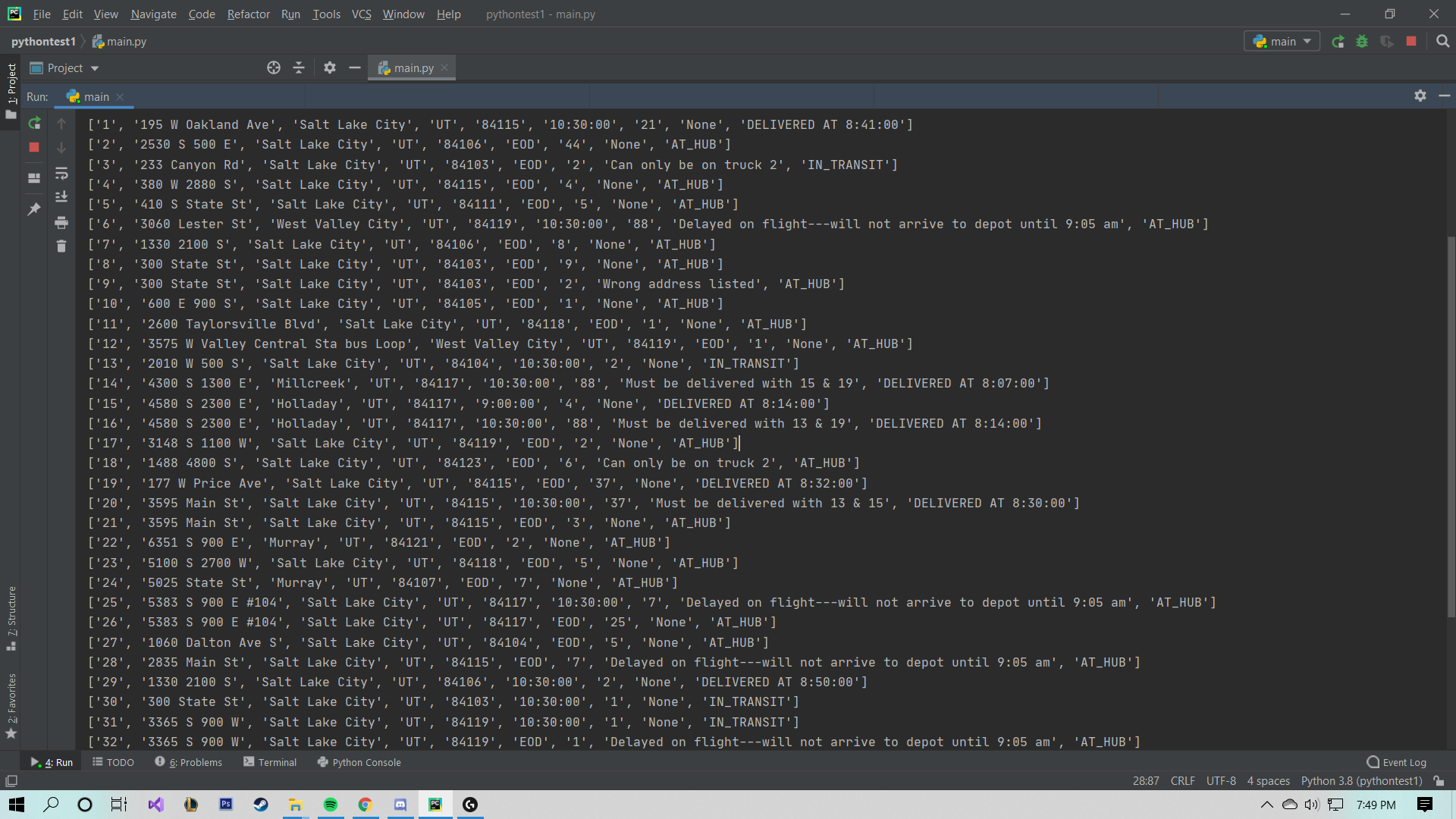
The data structure I use to create the hash table to store the packages in is a list, with each element in the list being another list. For example, this is how I store package #1:

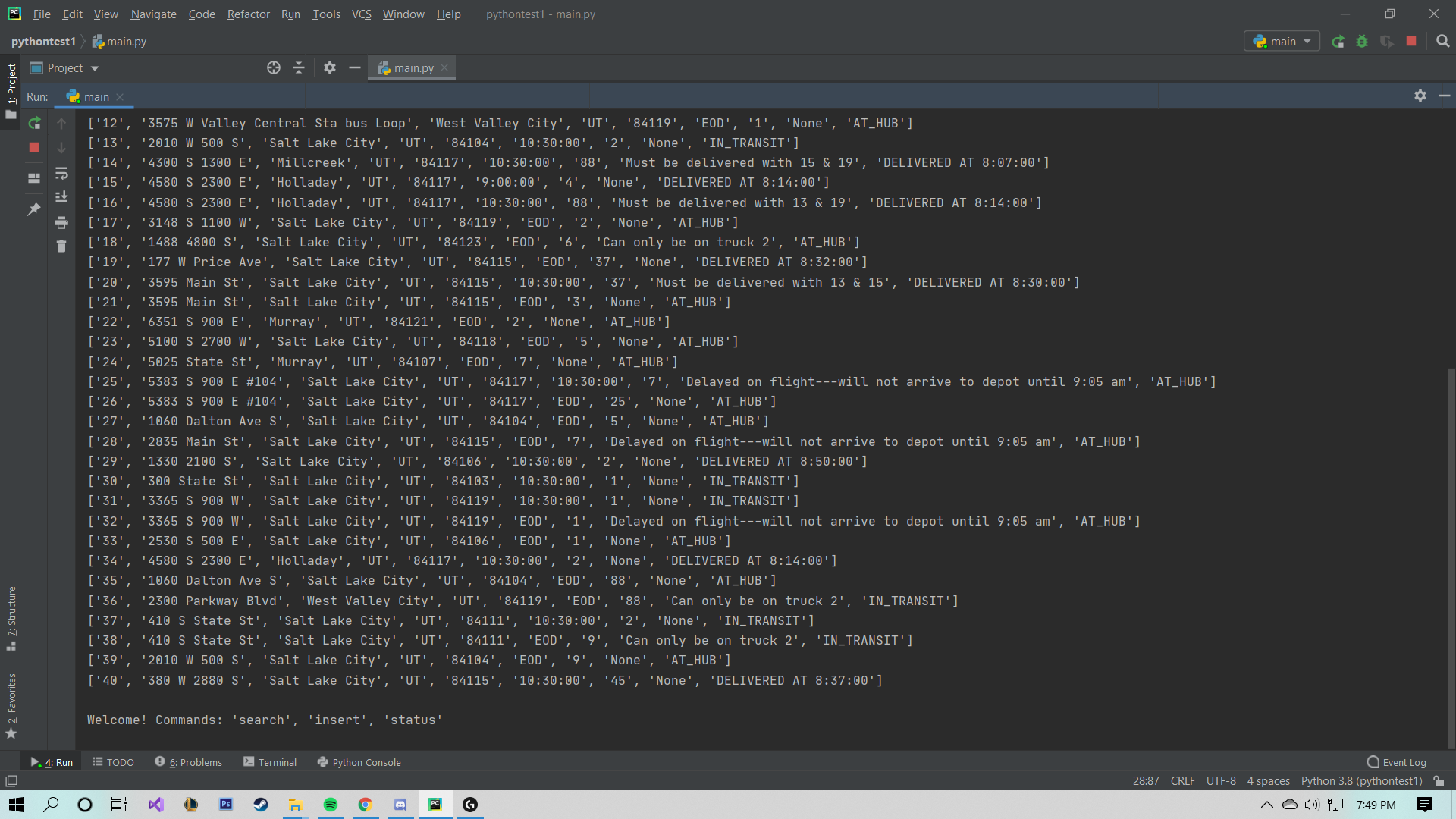
['1', '195 W Oakland Ave', 'Salt Lake City', 'UT', '84115', '10:30:00', '21', 'None', 'AT\_HUB']

Since every package has a unique package ID, it is easy to keep track of packages. Each data point is unique thanks to the package IDs, making it easy to access and locate specific packages. The program also includes an insert method and a search method, to add a package or view a package. Because the data structure is in essence a list, the insert method always adds the package to the end of the list, so as to not have to reassign storage to each element in the list. The search function is simple; given a package ID it will iterate through the hash table until a matching package ID is found, if it is, then it will return the package.

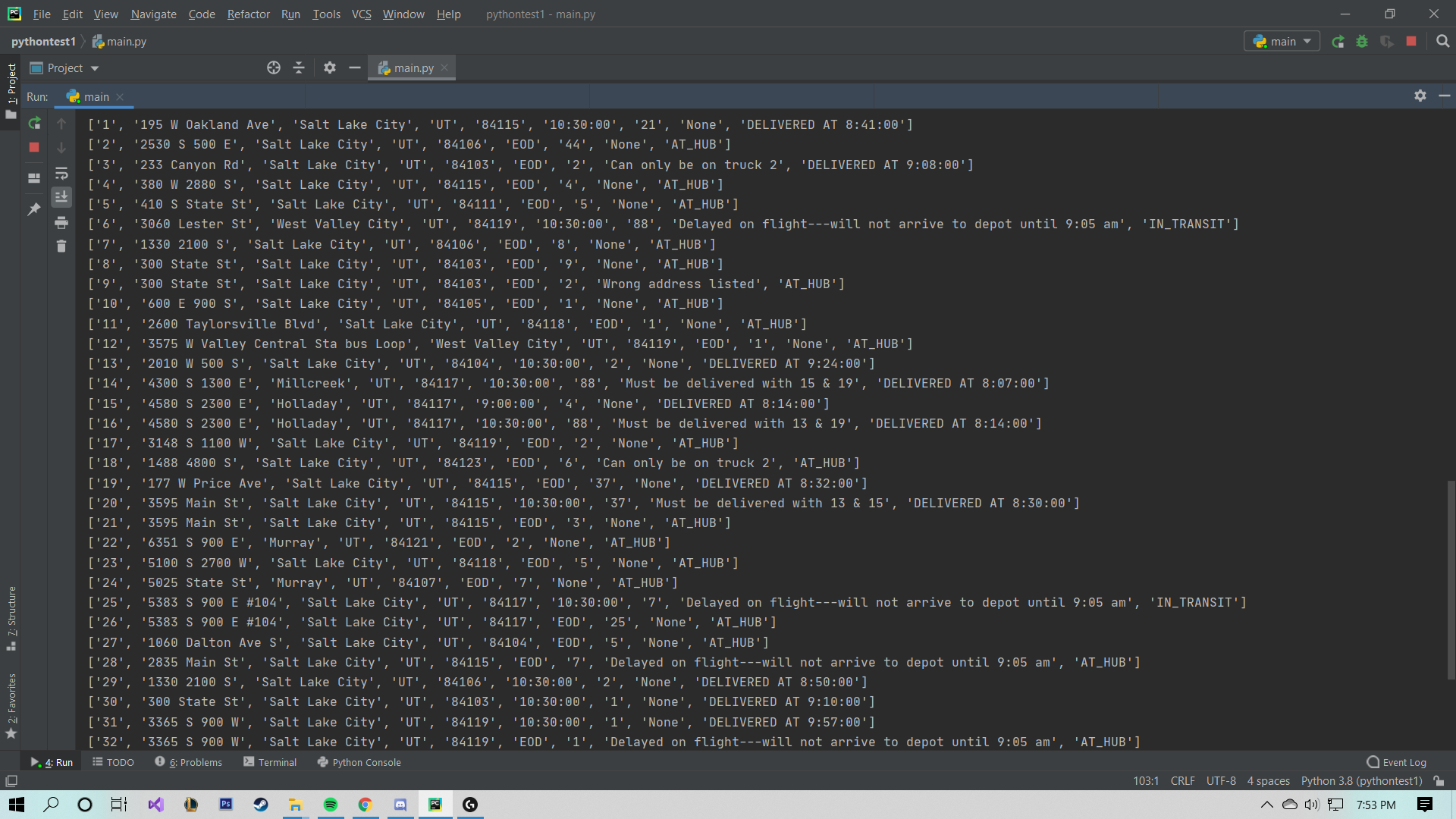
**Status Checks**

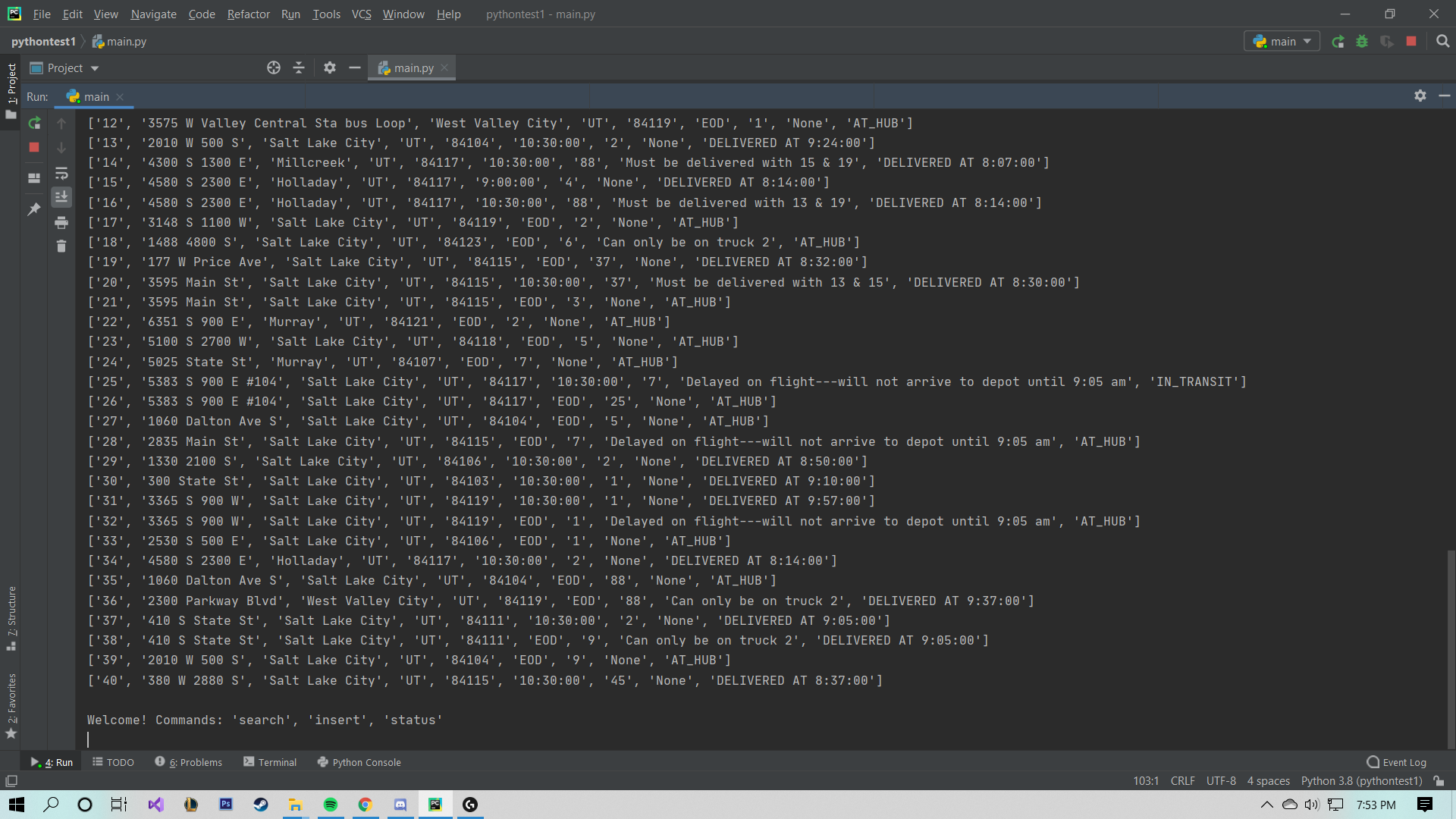
* **9:00 a.m.**

****

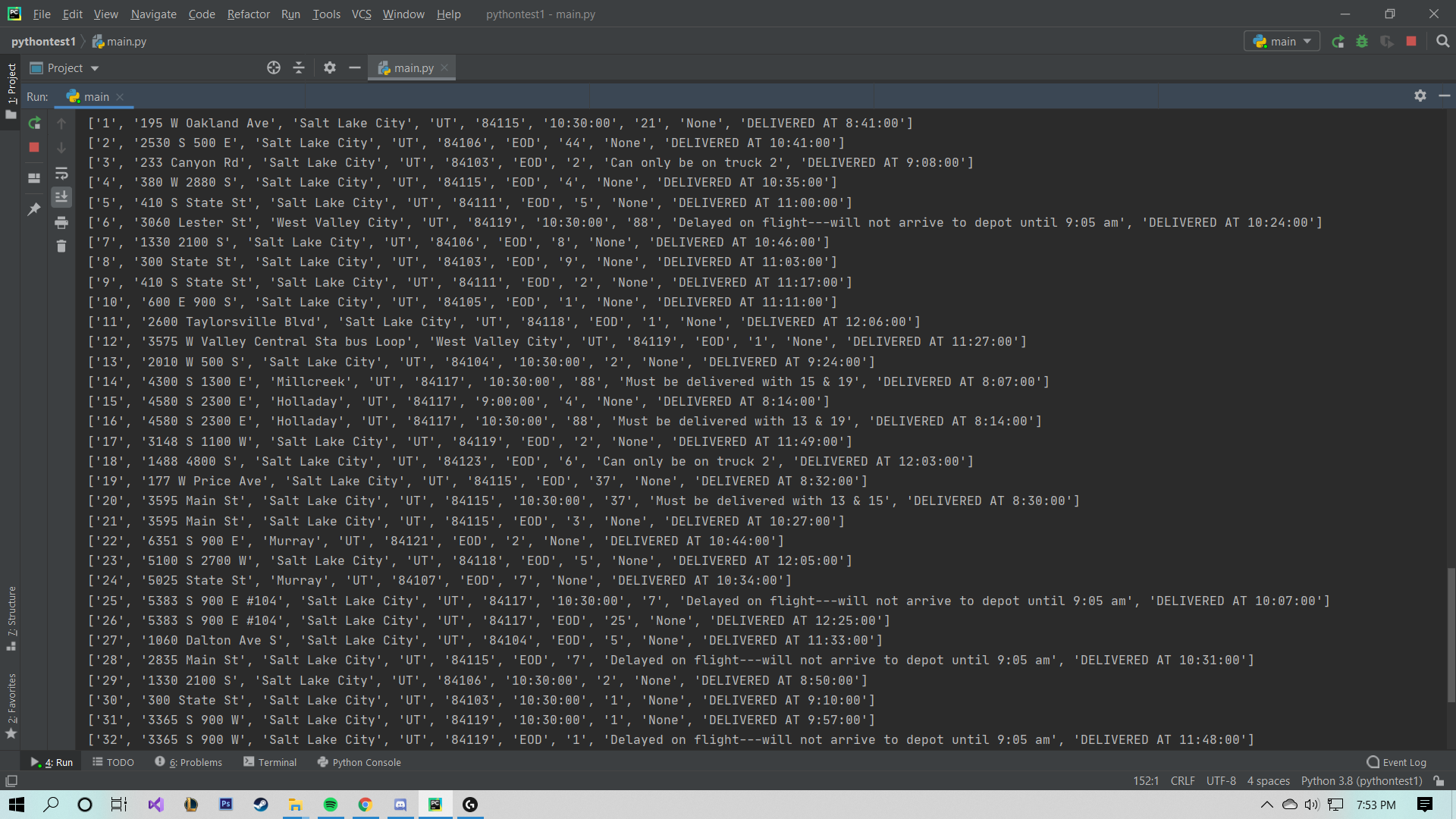
****

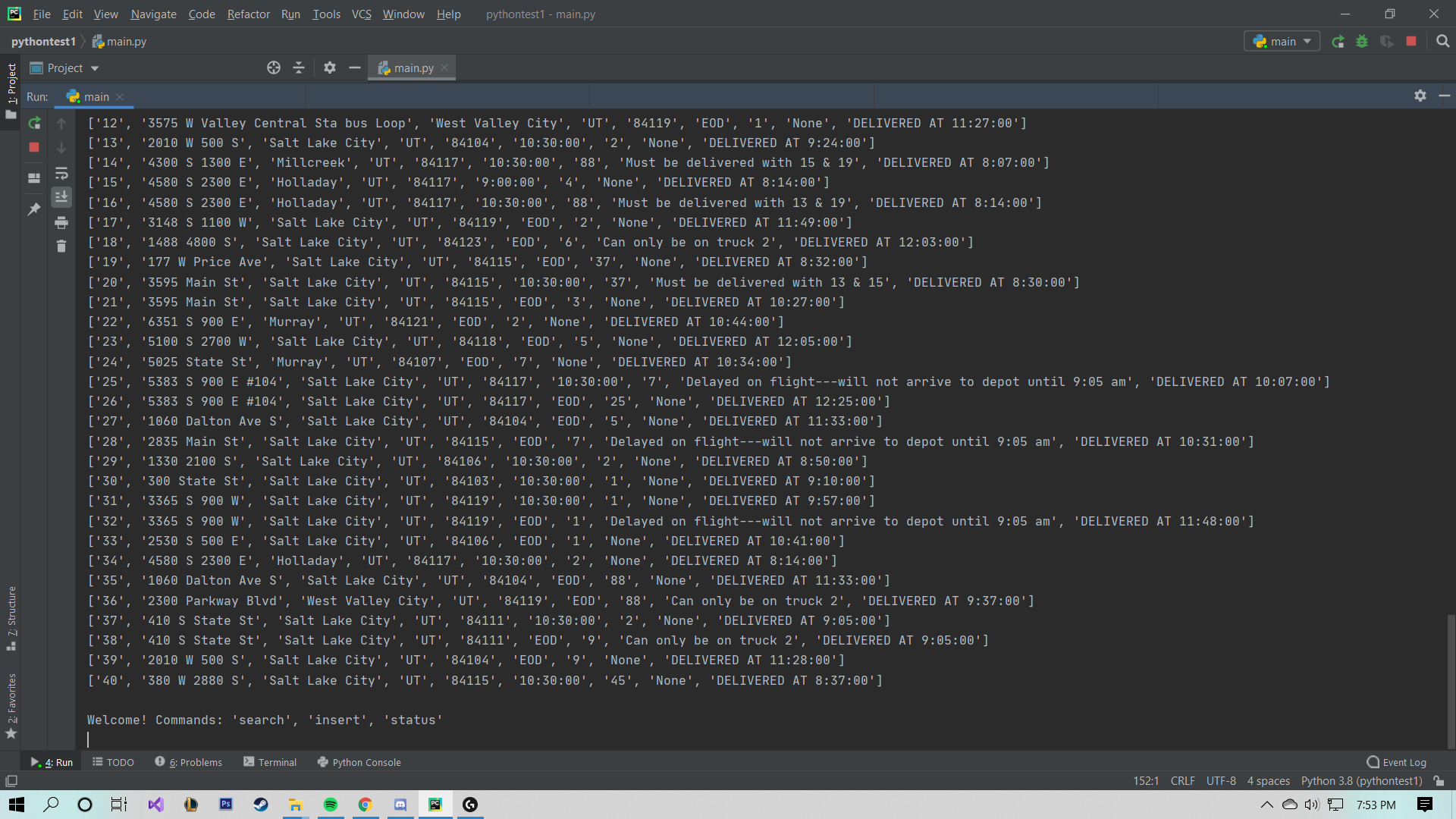
* **10:00 a.m.**

****

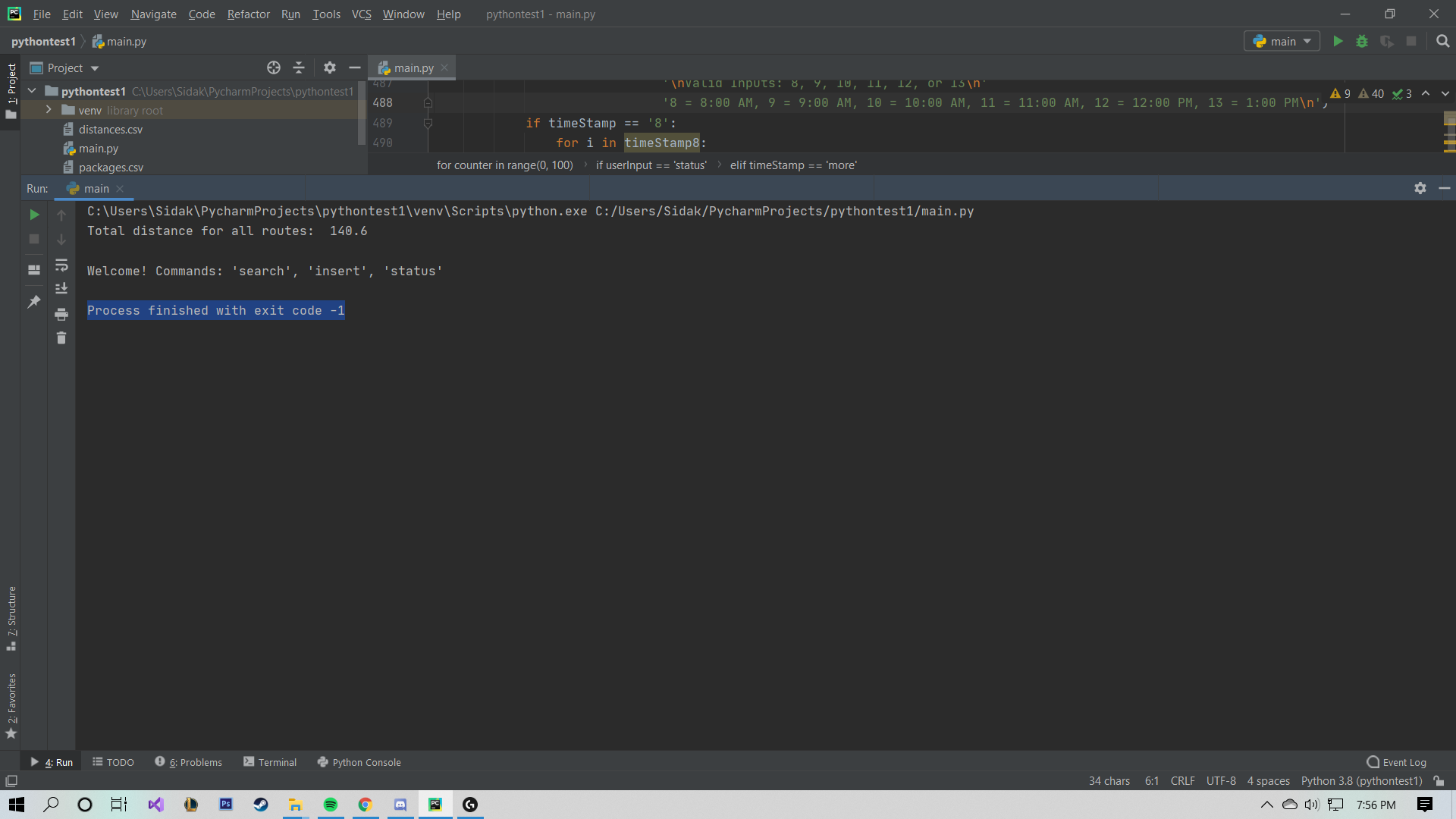
****

* **1:00 p.m.**





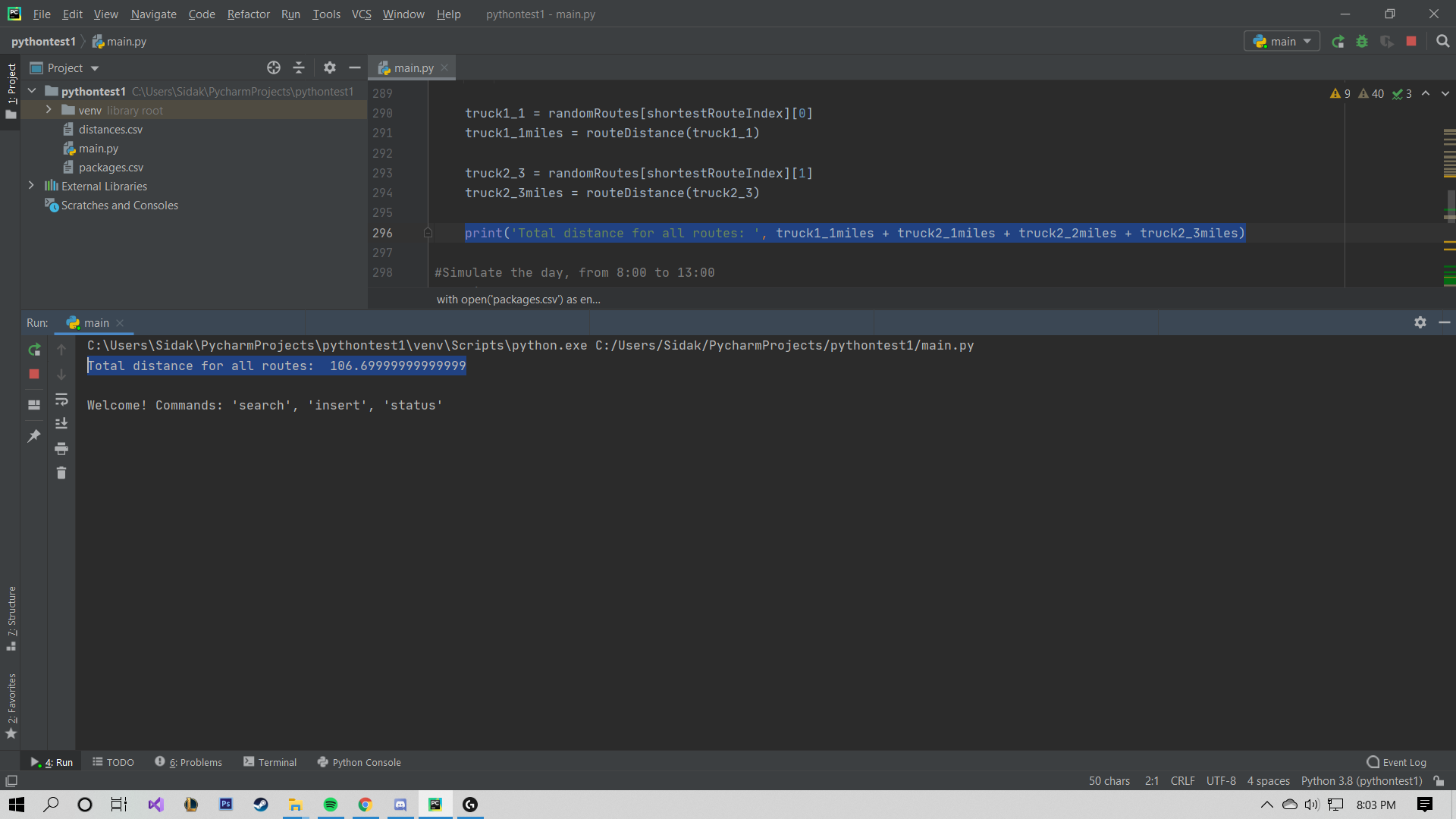
**Screenshot of Code Execution**



# Strengths of Chosen Algorithm

The chosen algorithm has two strengths that I noticed: it automatically accounts for the route starting and ending at the hub, and it also has no errors when two packages of the same address are input to be routed. If two or more packages have the same delivery address, it will deliver them all at the same time, which may be common sense but I am glad that my algorithm was able to do that.

**Verification of Algorithm**



**Other Possible Algorithms & Algorithm Differences**

I could have used Djikstra’s algorithm, or the A\* algorithm, instead of my chosen algorithm.

*Dijkstra's Algorithm:*

1. Faster runtime
2. Would find shorter paths i.e: if A->B->C is 5 miles but A->C is 6 miles it would find such paths
3. Better solution
4. Cleaner code
5. Does not support returning to start distance, would need to tweak

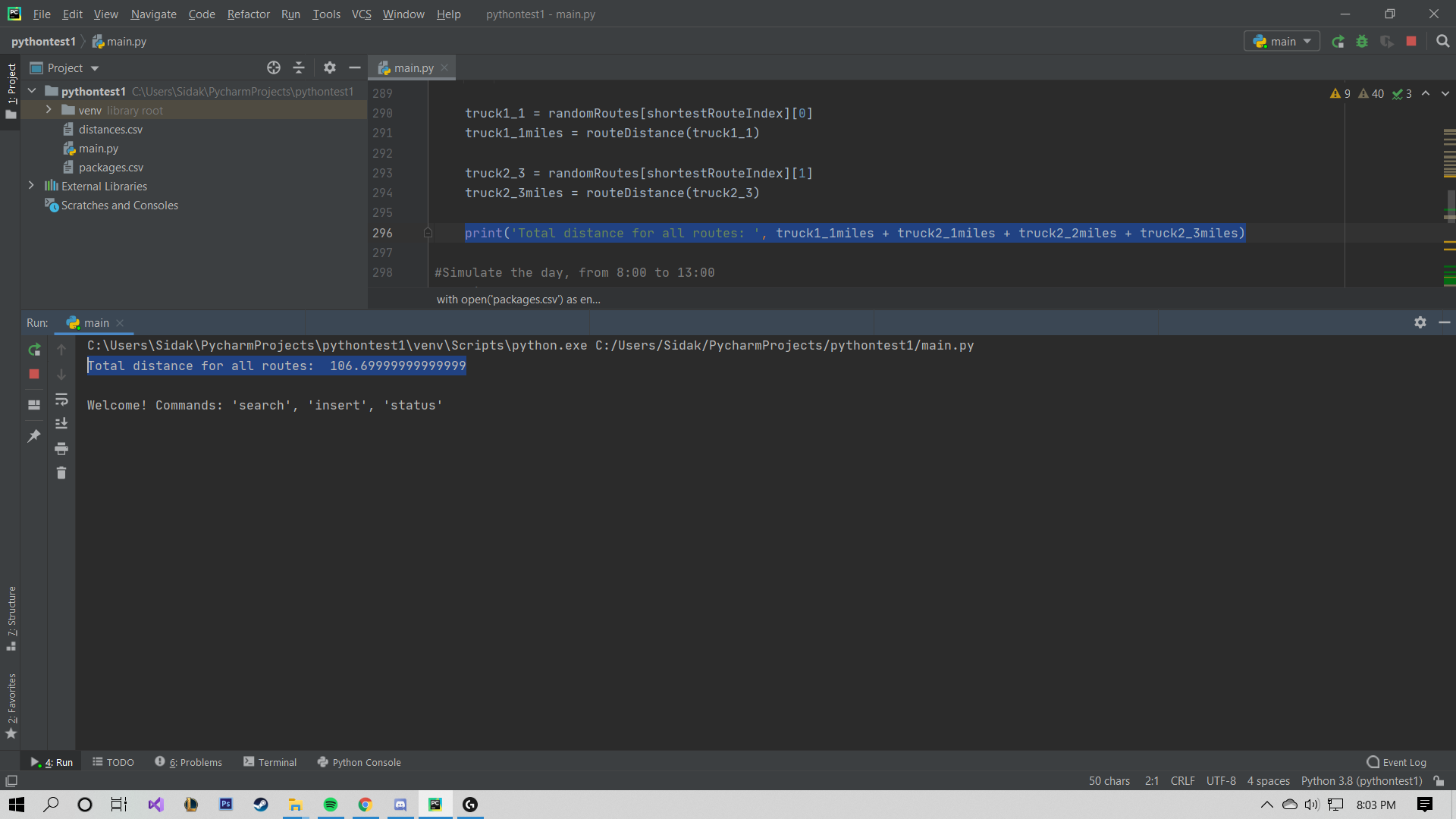
*A\* Algorithm:*

1. High space complexity
2. Uses weighted graphs
3. Cleaner code
4. Better solution (than Dijkstra’s algorithm as well)
5. Does not support returning to start distance, would need to tweak

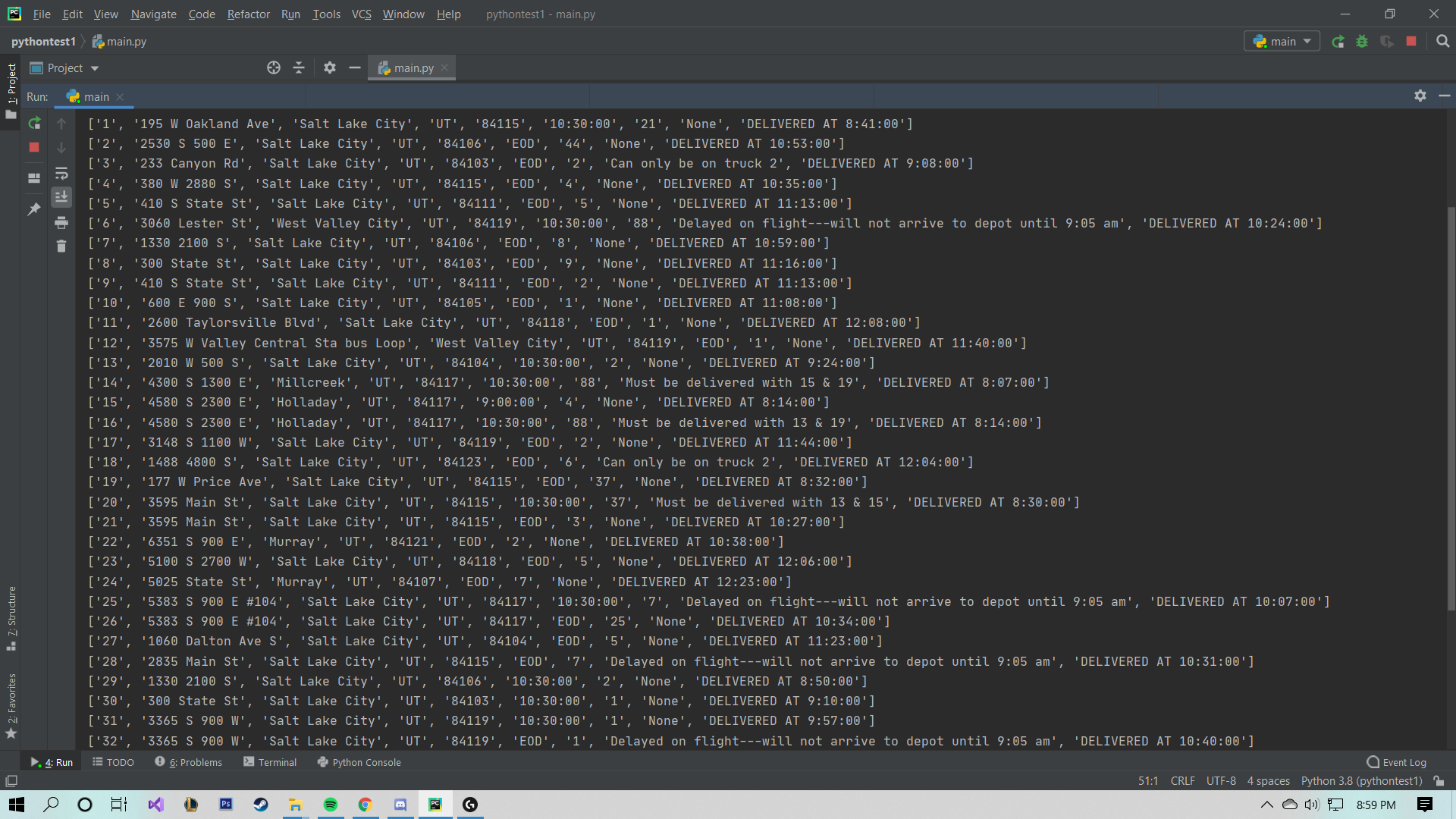
**Different Approach**

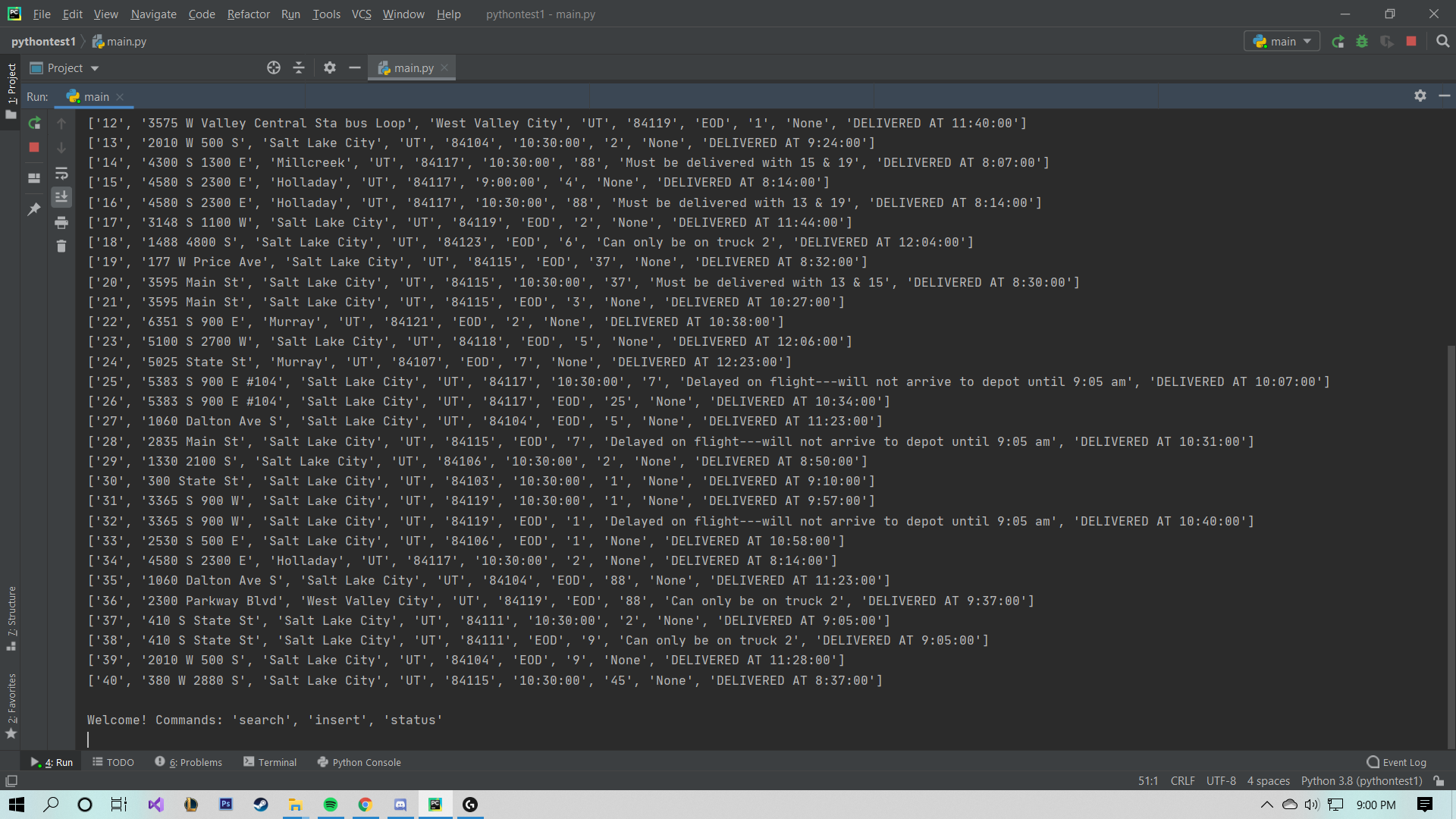
If I had to redo this project again, I would definitely plan things better, as well as research various algorithms more. To plan, I would create a diagram of how the entire program worked on a small scale and then add to it until it was a functioning program. Algorithm wise, I would definitely have scoured more research papers, online forums, and other sources to find which algorithm would work the best for this specific program. If I could go back in time, I would have even implemented various algorithms just to get used to them as well as seeing which one was really the best. Another thing I would change is how I used data structures here: I would look into using a priority queue as well as other ways to optimize time and space complexity.

**Verification of Data Structure and Solution**

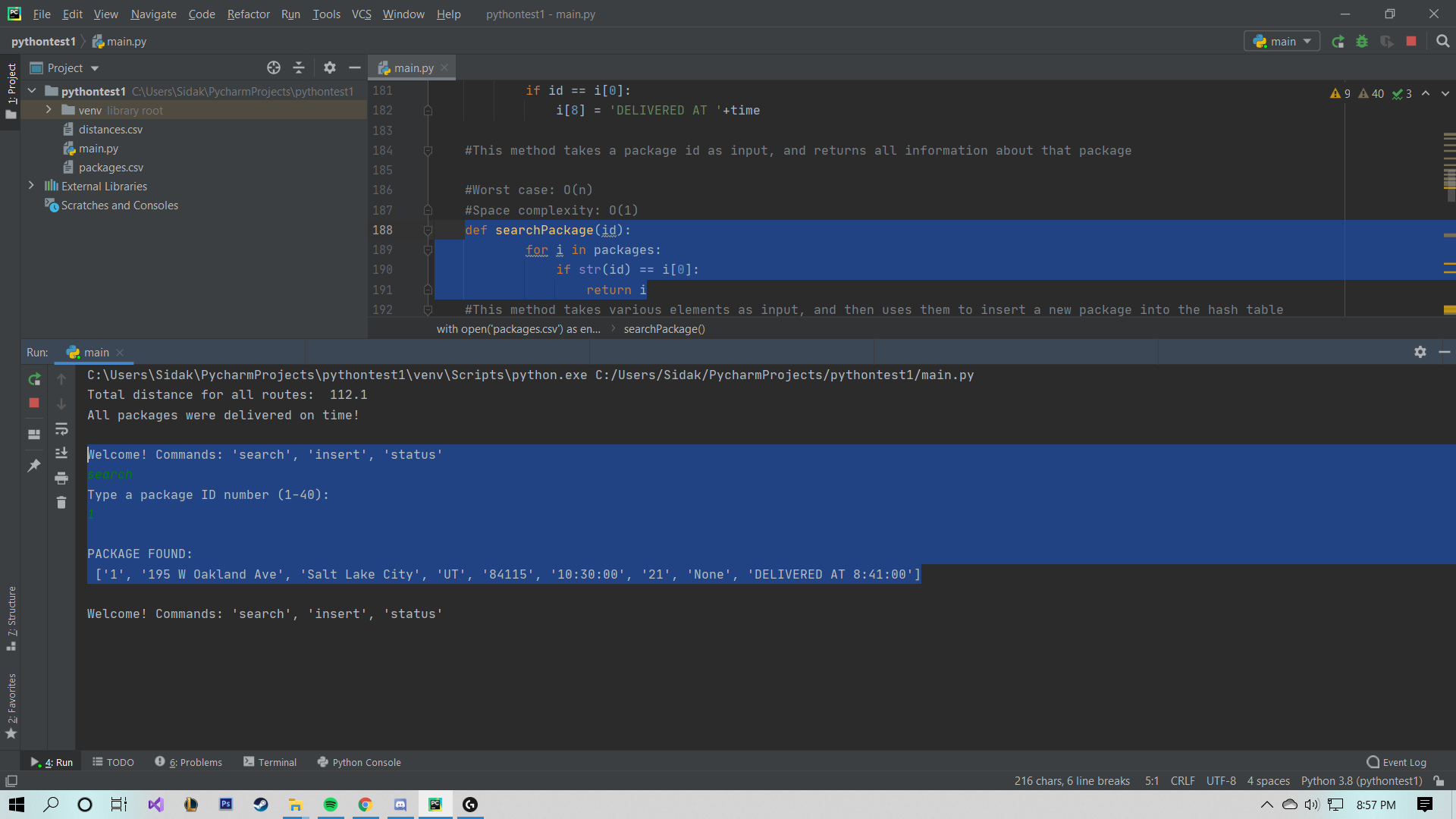


The miles will vary, but averages around 118 miles total, and it will never be over 145 miles. This is because after delivering all the packages with constraints, the rest are loaded randomly in 250 simulated routes, and the best two routes are chosen for the last two trucks. The mileage is accurate, because I simply add the distance that each truck’s route has, as shown in the highlighted line above.





All packages are delivered on time according to their constraints. The packages with constraints are hand sorted into trucks so no matter they will always be delivered on time.



Here is the lookup function for the hash table of packages. It takes the package ID as input and returns the specified package and all of its details.

**Efficiency**

The data structure used in the solution is a hash table, which was created from a list, in which each element is a list. The data is used to hold all information about the packages, and the data structure is accessed several times throughout the program in order to load trucks, update packages, and create routes. The type of data within the package is all stored as a string, and each list (or package) holds all data relevant to the package, including package status (AT\_HUB, IN\_TRANSIT, DELIVERED).

**Overhead**

The overhead includes an explanation of computational time, memory, and bandwidth when handling data. Because this program is run locally on a machine, the explanation of memory and bandwidth when handling data are not applicable to this scenario. Had this program been connected to a database hosted on a server, then it would. The computational time of the hash table varies depending on the action. Adding to the hash table is O(1), while removing a package would take a lot longer, due to every element after the deleted element having to be moved back, which would result in a time complexity of O(n). In our program, there are many cases in which we have to iterate through the hash table to find a specific element, and iteration in a list has a time complexity of O(n).

**Implications**

If the number of packages, trucks, or cities increased, then the code would need to be revised as it does not accommodate any changes. This is because the program was hard coded to work with the inputs given, and was not intended to be dynamic enough to handle a different dataset. If the number of trucks increased, there still wouldn’t be a change, because we would need the number of drivers to increase. Assuming there are more drivers, trucks, packages, and cities, then a complete revamp of the program would be required. The algorithm could still be used, but the program would need to be able to handle any city and any addresses, given the distances. It would also need to be connected to a database so that all WGUPS data would be stored in a central location, allowing for data analysis and remote operators to check progress. In that situation, each city would have their own routing program, and then there would be another program in which operators could check routes from multiple cities. The two programs would be accessed via the same method, however a login page would direct city-wide operators and region-wide operators to different programs.

**Other Data Structures**

A different data structure could have been a *Set*, or if I were to sort the lists prior to adding them to the .csv file, it could have been a *Deque*.

**Data Structure Differences**

*Set:*

1. Faster to see if an element is in the data structure
2. Slower when iterating over
3. Is not ordered
4. All elements must be unique

*Deque:*

1. Item access times are slower
2. Adding an element to the beginning would not require you to reallocate all elements after it