Core Algorithm Overview

**Stated Problem:**

The purpose of this project is to create an effective and efficient sorting algorithm using the python programming language. The program will read three csv files, one with the delivery package information, one with the delivery addresses, and one with a distance matrix for the delivery addresses. The delivery package information is separated into three lists which represent the three delivery trucks and is sorted by the distance value and is added to an optimized list for each truck. The program will output the total distance of miles for all three trucks and display a menu in the terminal. The user will have the ability to view timestamps for all the packages at the user’s selected time. The user will also be able to search packages by package ID, weight, status, address, zip code, city, and delivery deadline.

**Algorithm Overview:**

The greedy algorithm is created by doing the following:

1. An optimized list of packages is passed in along with the associated truck number and the current location.
2. The current location is compared to the locations of all the packages in the truck to determine the nearest location.
3. The lowest value is chosen after all the items in the truck have been compared and the value is popped from the truck list and the truck goes to the new address location.
4. The current address value is appended to an optimized location list and the algorithm updates the current location with the new address and calls the function again with the smaller truck list.

The space-time complexity of this self-adjusting greedy algorithm has a worst-case runtime O(N^2) and a best-case runtime of O(1). The worst case is almost always guaranteed as the best case is only possible when the list of packages being loaded onto a truck is empty.

Pseudocode for the sorting algorithm:

**Function sort\_packages(truck\_distance\_list, truck\_number, current\_location):**

**If length of truck\_distance\_list is empty:**

**Return the empty list**

**Else:**

**Set lowest\_value to 10.0**

**Set new\_location to 0**

**for each index in truck\_distance\_list:**

**if function current\_distance less than 0:**

**set lowest\_value to current\_distance value**

**set new\_location to current\_distance index**

**for each index in truck\_distance\_list:**

**if function current\_distance is equal to the lowest\_value:**

**if truck\_number equals one:**

**append index to optimized first truck list**

**append index[1] to optimized first truck list**

**set current package index to a variable**

**pop current package from truck\_distance\_list**

**set current location to new\_location**

**function sort\_packages(truck\_distance\_list, truck\_number, current\_location)**

**if truck\_number equals two:**

**append index to optimized second truck list**

**append index[1] to optimized second truck list**

**set current package index to a variable**

**pop current package from truck\_distance\_list**

**set current location to new\_location**

**function sort\_packages(truck\_distance\_list, truck\_number, current\_location)**

**if truck\_number equals three:**

**append index to optimized third truck list**

**append index[1] to optimized third truck list**

**set current package index to a variable**

**pop current package from truck\_distance\_list**

**set current location to new\_location**

**function sort\_packages(truck\_distance\_list, truck\_number, current\_location)**

The total time complexity of the algorithm is as follows:

A. O(1) or constant time

B. O(1) or constant time

C. O(N)

D. O(N^2)

Total: O(1) + O(1) + O(N) + O(N^2) = O(N^2)

Below is a table breakdown of the worst-case space-time complexity of each file in the Python application:

**hashtable.py**

|  |  |  |  |
| --- | --- | --- | --- |
| **Line Number** | **Method** | **Space Complexity** | **Time Complexity** |
| 12 | \_\_init\_\_ | O(1) | O(1) |
| 20 | \_get\_hash | O(1) | O(1) |
| 26 | insert | O(N) | O(N) |
| 42 | update | O(1) | O(N) |
| 53 | get\_value | O(N) | O(N) |
| 63 | delete | O(N) | O(N) |
| **Total** |  | 3N + 3 = O(N) | 4N + 2 = O(N) |

**packages.py**

|  |  |  |  |
| --- | --- | --- | --- |
| **Line Number** | **Method** | **Space Complexity** | **Time Complexity** |
| 16 | None | O(N) | O(N) |
| 68 | get\_hash\_map | O(1) | O(1) |
| 72 | get\_first\_truck | O(1) | O(1) |
| 76 | get\_second\_truck | O(1) | O(1) |
| 80 | get\_first\_truck\_reload | O(1) | O(1) |
| **Total** |  | N + 4 = O(N) | N + 4 = O(N) |

**deliveries.py**

|  |  |  |  |
| --- | --- | --- | --- |
| **Line Number** | **Method** | **Space Complexity** | **Time Complexity** |
| 44 | None | O(1) | O(N) |
| 52 | None | O(N^2) | O(N^2) |
| 66 | None | O(N) | O(N) |
| 82 | None | O(1) | O(N) |
| 89 | None | O(N^2) | O(N^2) |
| 101 | None | O(N) | O(N) |
| 118 | None | O(1) | O(N) |
| 125 | None | O(N^2) | O(N^2) |
| 139 | None | O(N) | O(N) |
| 153 | total\_distance | O(1) | O(1) |
| **Total** |  | 3N^2 + 3N + 3 = O(N^2) | 3N^2 + 6N + 1 = O(N^2) |

**main.py**

|  |  |  |  |
| --- | --- | --- | --- |
| **Line Number** | **Method** | **Space Complexity** | **Time Complexity** |
| 21 | None | O(1) | O(N) |
| **Total** |  | 1 | O(N) |

**distance.py**

|  |  |  |  |
| --- | --- | --- | --- |
| **Line Number** | **Method** | **Space Complexity** | **Time Complexity** |
| 28 | check\_distance | O(1) | O(1) |
| 38 | current\_distance | O(1) | O(1) |
| 51 | first\_truck\_time\_func | O(N) | O(N) |
| 65 | second\_truck\_time\_func | O(N) | O(N) |
| 79 | third\_truck\_time\_func | O(N) | O(N) |
| 93 | get\_address | O(1) | O(1) |
| 119 | sort\_package | O(N^2) | O(N^2) |
| 154 | get\_first\_opt\_truck\_index | O(1) | O(1) |
| 158 | get\_first\_opt\_truck\_distance\_list | O(1) | O(1) |
| 162 | get\_second\_opt\_truck\_index | O(1) | O(1) |
| 166 | get\_second\_opt\_truck\_distance\_list | O(1) | O(1) |
| 170 | get\_third\_opt\_truck\_index | O(1) | O(1) |
| 174 | get\_third\_optimized\_truck\_distance\_list | O(1) | O(1) |
| **Total** |  | N^2 + 3N + 9 = O(N^2) | N^2 + 3N + 9 = O(N^2) |

**menu.py**

|  |  |  |  |
| --- | --- | --- | --- |
| **Line Number** | **Method** | **Space Complexity** | **Time Complexity** |
| 4 | print\_timestamp | **O(N)** | **O(N)** |
| 30 | search\_by\_id | O(1) | O(1) |
| 52 | search\_by\_address | O(N) | O(N) |
| 79 | search\_by\_deadline | O(N) | O(N) |
| 102 | search\_by\_city | O(N) | O(N) |
| 125 | search\_by\_zip | O(N) | O(N) |
| 151 | search\_by\_weight | O(N) | O(N) |
| 174 | search\_by\_status | O(N) | O(N) |
| 199 | print\_packages | O(1) | O(1) |
| Total |  | 7N + 2 = O(N) | 7N + 2 = O(N) |

Memory and computational time stay almost linear throughout the entire application. This allows the available set of inputs to scale without being overloaded by memory availability constraints. Bandwidth is not an issue in the current application as the application is managed on a local machine that does not require network resources.

**Advantages of chosen Algorithm**

This greedy algorithm performs all the necessary functions to meet the constraints and delivers the packages within the range of 90 miles. The greatest strength of this algorithm is its capability to quickly find the best path for a given truck. Another advantage of the greedy algorithm approach is that it can scale with any set of data and addresses provided to it. Additionally, the user interface has the ability for a user to see all the package details and their delivery status at a given time. It can also search for an individual package based on its package ID, delivery address, delivery zip code, delivery city, package weight, delivery deadline, and package status.

An algorithmic approach I could have used to optimize the packages was a dynamic programming approach. “Dynamic programming is a problem-solving technique that splits a problem into smaller subproblems” (Zybooks, 3.5). The advantage of using a dynamic approach is that I can store paths along a route and check if there is a quicker way to get to a location by first traveling to another location. Storing these different paths may have created a large space complexity but it could have yielded a shorter path. In the sorting algorithm I used a broad approach instead of attempting to break the program down into smaller functions. A second algorithm I could have implemented for the ideal route is a self-adjusting heuristic. This approach would start at the hub then determine the closest path to the hub. It would then determine all packages that needed to be delivered and load them into the truck. From there it would start at the new location and determine the shortest path from that location. If a location has already been visited by that truck, then it would move on the to the next closest location until all 40 packages had been allocated to a truck and a path was set. My sorting algorithm and the self-adjusting heuristic share a similar attribute in the sense that the shortest available path is constantly chosen.

**Programming Models:**

The programming model for this application is limited as it is currently hosted on a local machine. The application is written using Python 3.8.3 and is executed in the visual studio IDE. There is no communication protocol present as the application pulls data from CSV files that are in the project folder on the local machine. To do this we use the csv.reader function from the csv library in Python. Thus, data exchanges are limited to the communications of the application and the local machine. Additionally, there is no target host environment for the application as a network connection is not needed or established. In terms of information semantics, the current software requirements are organized in a functional manner. Defining a set of interaction semantics to determine the flow of connection, data, and the resulting information would not be required with the application being hosted on a local machine.

**Ability to Adapt**

The core functions of the application are intended to be able to scale and address changes in the number of packages, the number of trucks, and the number of locations. Minor changes are required to scale with any of these components. For example, another set of location csv files can be entered into the application to calculate a new route and determine an optimal path. Additional packages can be added, and the program will determine where to place the packages. This approach also allows a great deal of control when implementing several sub-applications as the design allows the input set to be altered easily. A possible problem with future scaling in this application is the approach I take to load the packages into the truck. It is currently done manually to meet all the package constraints and deadlines. Given the opportunity to improve this project, I would work on developing a heuristic approach to determine what packages go onto which truck. Having this process automated would benefit both the ability for the application to adapt to new environments and potentially provide a better path set for the greedy algorithm to work with. However, the core functions of the application have great capability to scale with changing markets as the greedy algorithm handles limited data sets very quickly.

**Efficiency and Maintainability**

Overall, the software is very efficient, with two comparisons that have a time efficiency of O(N^2). While this may not be the best time complexity, it scales well with the 16-package limit per truck. It is also very supportable as much of the software is the same core functions that have been modified for use case. Debugging is easier because you can always refer to another instance of the function to determine where potential errors might be.

**Data Structures**

The primary data structure I implemented throughout the application is a list of lists. I chose this data structure because it is flexible and easy to work with. It also works well with a hash table and data retrieval. Additionally, the project is required to be able to have quick access to many attributes of a package. Using a list allowed me to create a hash table with chaining resulting in very fast lookup, insertion, deletion, and access to specific data elements. In fact, most operations involving the data structure had a constant time complexity of O(1). The biggest advantage for me was it allowed me to validate the accuracy of my trucks and packages along the route with very little extra overhead. One weakness I identified while working through this project is that using a list of lists is hard to utilize in an object-oriented way. For example, I had to apply the same operations to each list to deliver the packages for a given truck. Having a package factory class and a truck object would have shortened the coding and made it easier to manage scaling.

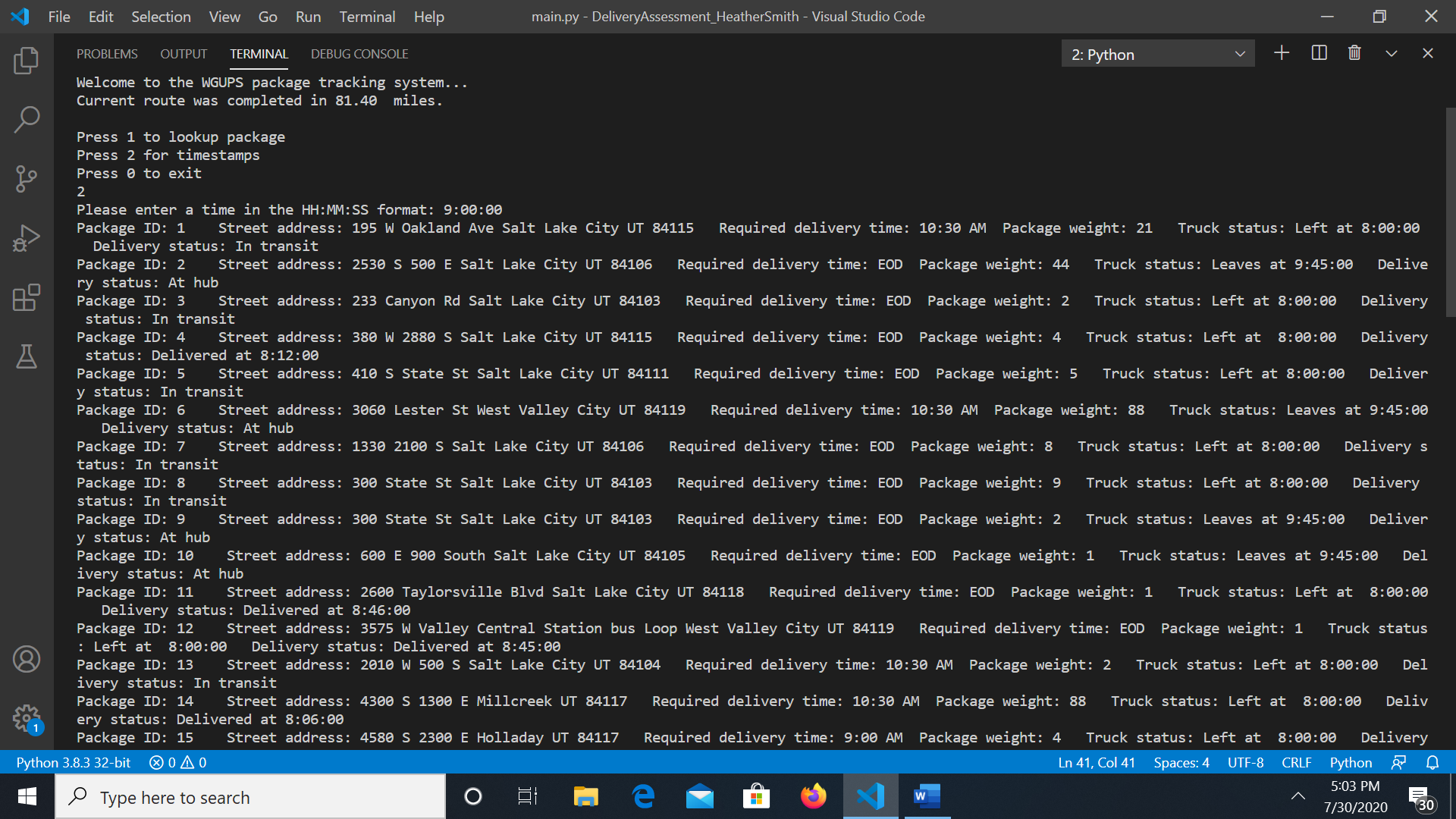
Looking back, I could have used various data structures to fit the requirements of this project. One popular data structure would be a binary search tree. The BST would be different in because I can presort the packages based on an attribute and quickly access them through a tree. Another data structure I could have implemented was a weighted graph. A huge advantage of using a weighted graph is that I could have grouped similar packages together as adjacent vertices. Then I could have traversed the graphs until a maximum traversal length of 16 was reached. This would also be a good method for future scaling.

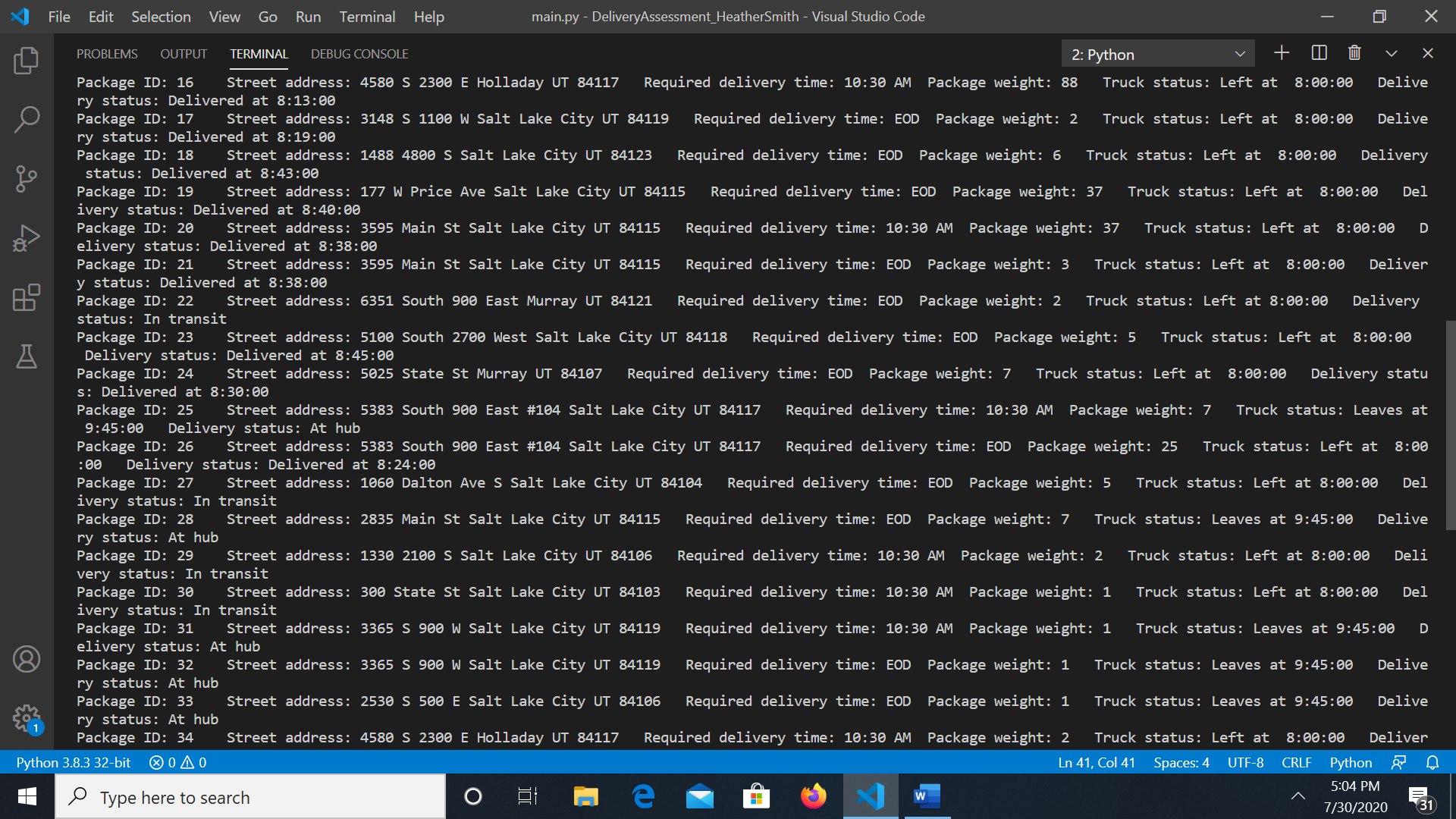
**Sources and In-Text Citations**

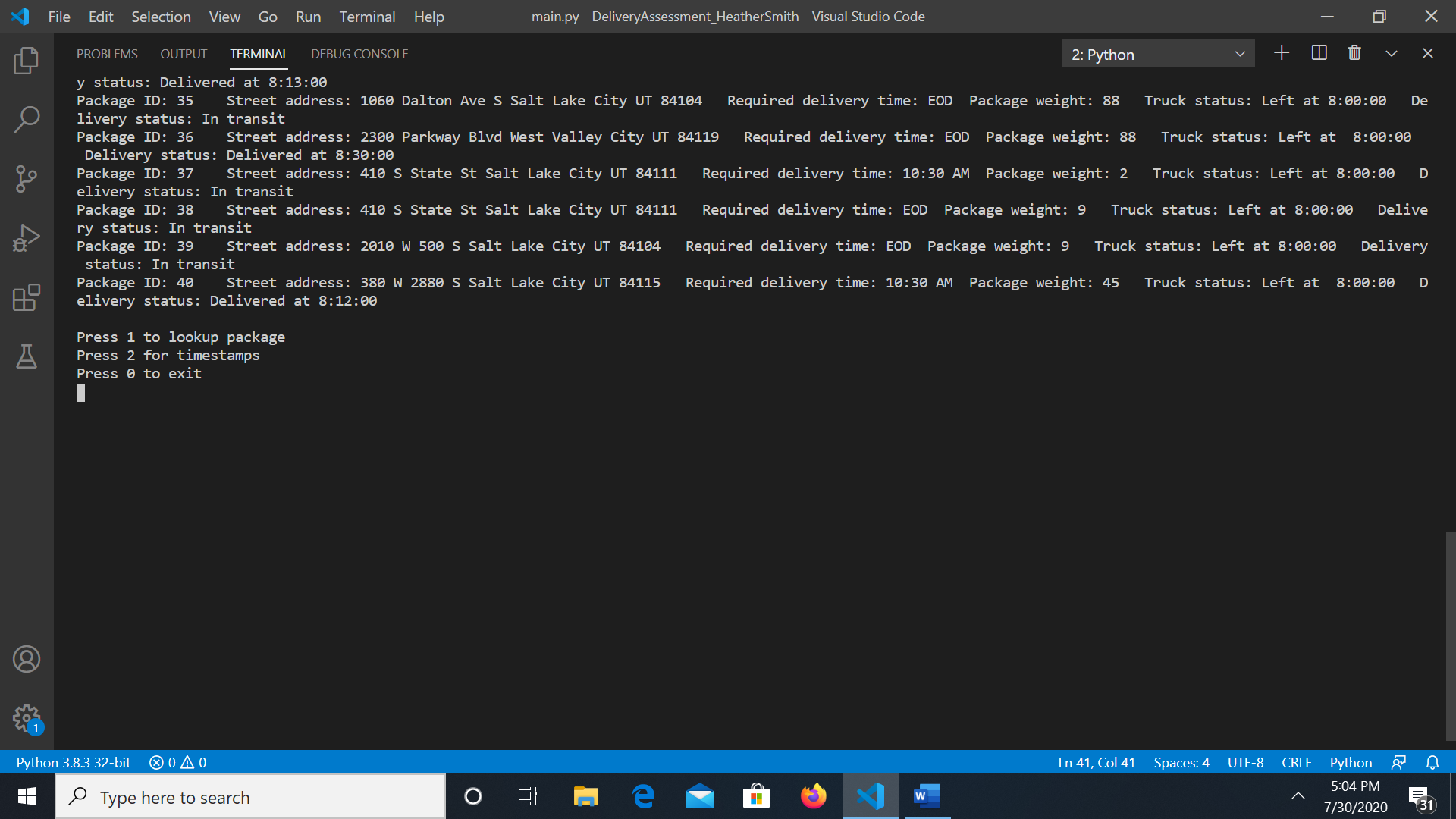
Learn.zybooks.com. (n.d.). *zyBooks*. [online] Available at: https://learn.zybooks.com/zybook/WGUC950AY20182019/chapter/3/section/5 [Accessed 10 Nov. 2019].

Screen Shots:

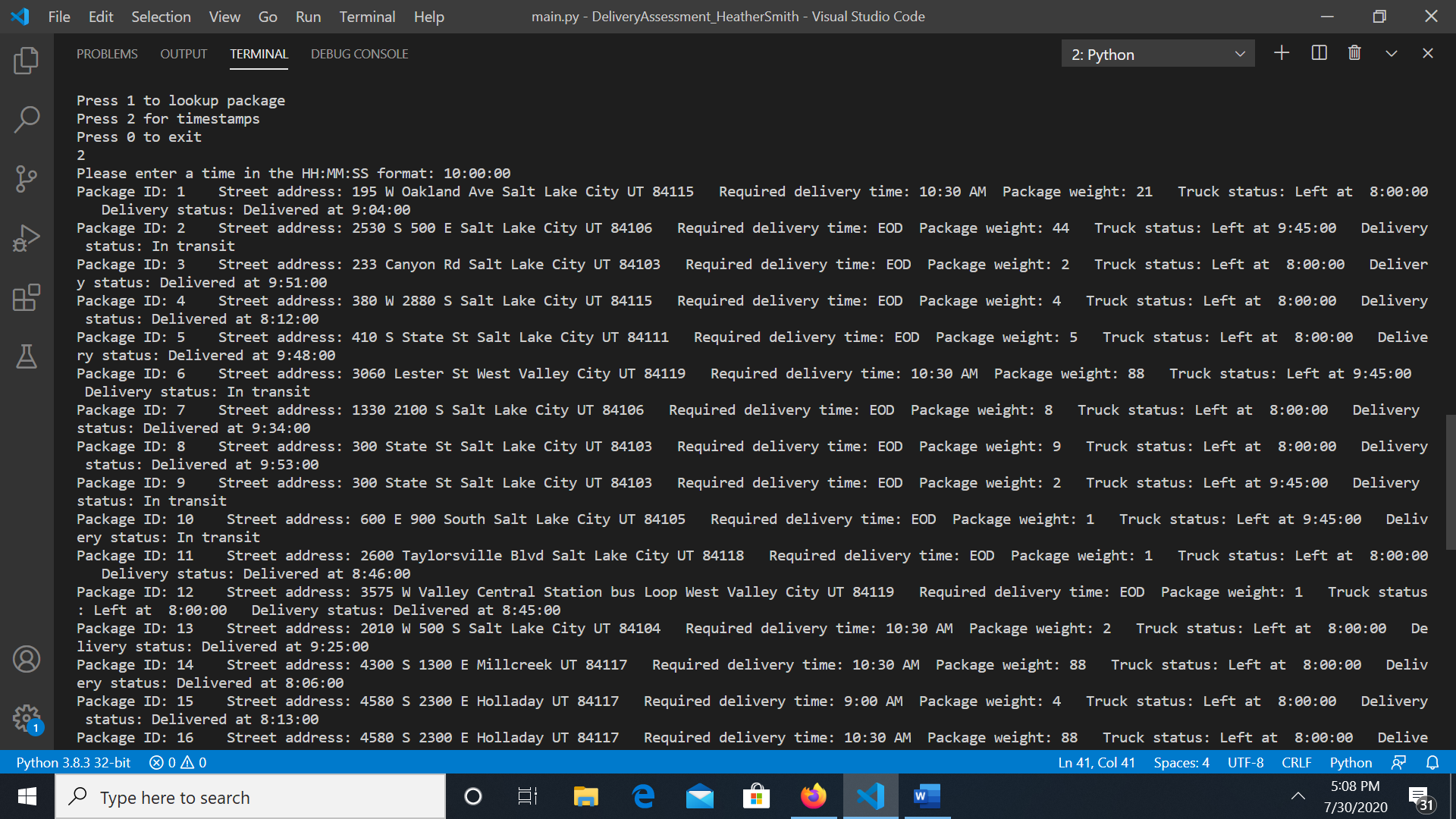
All packages between 8:35 am to 9:25 am

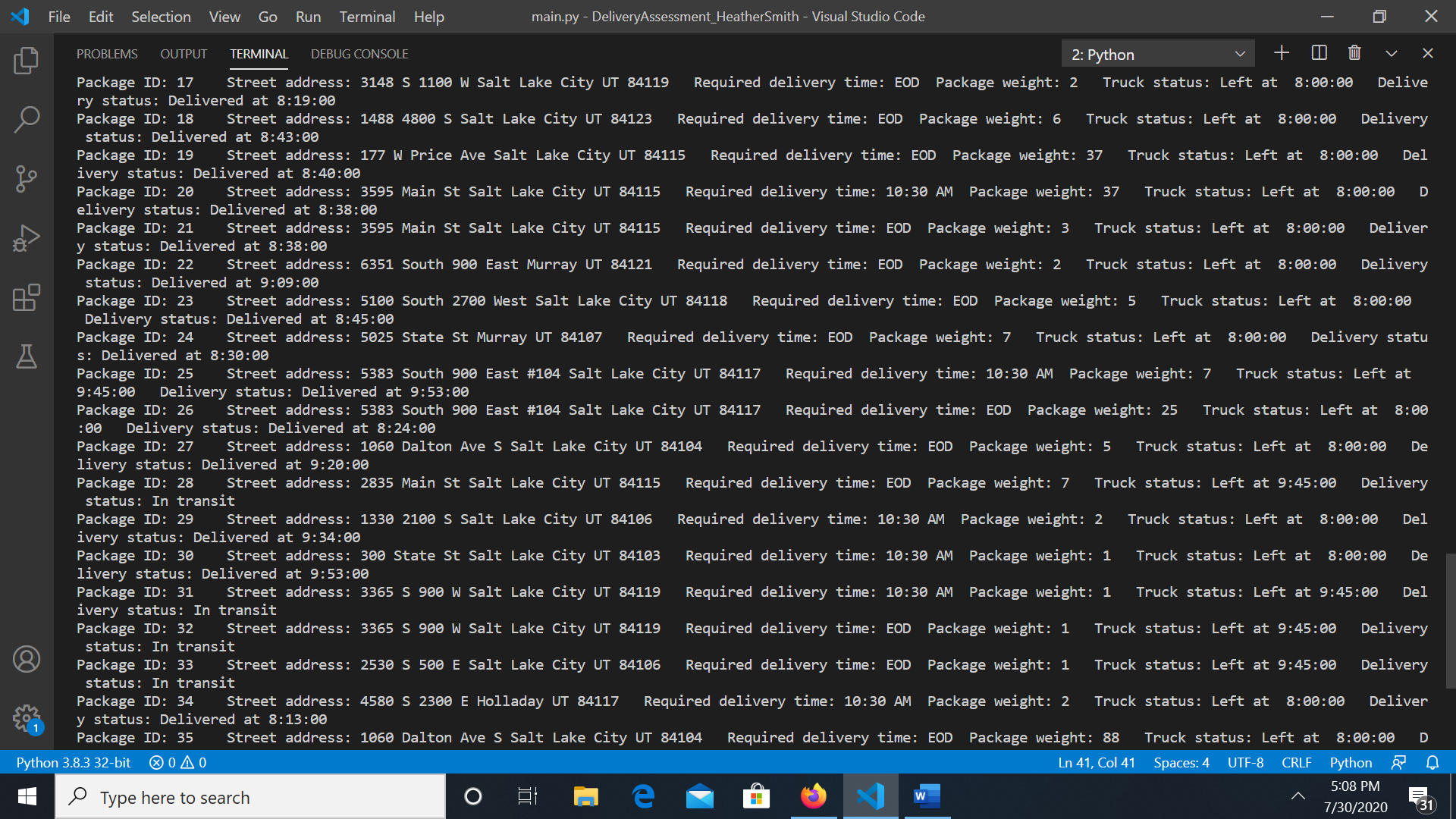


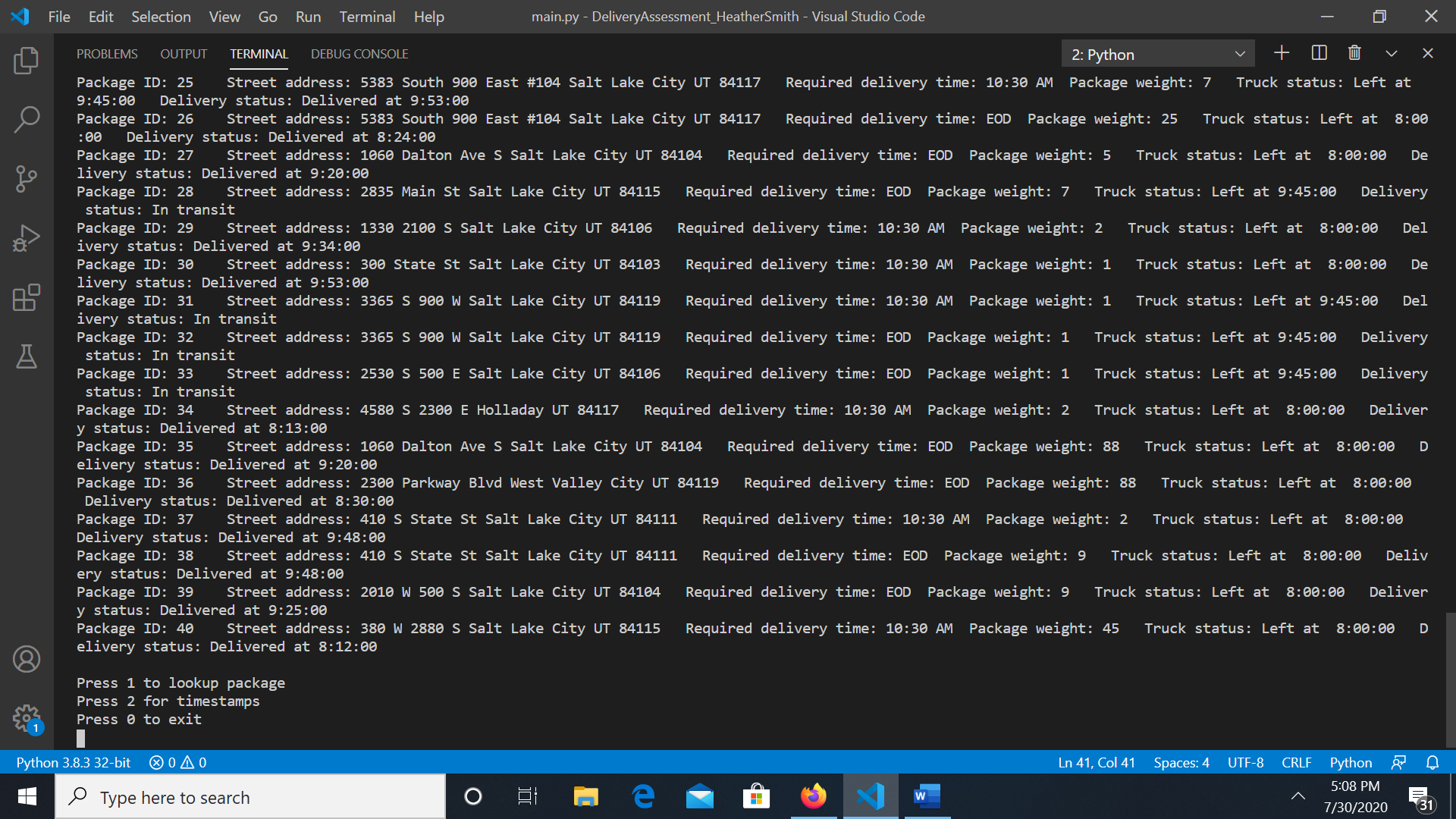




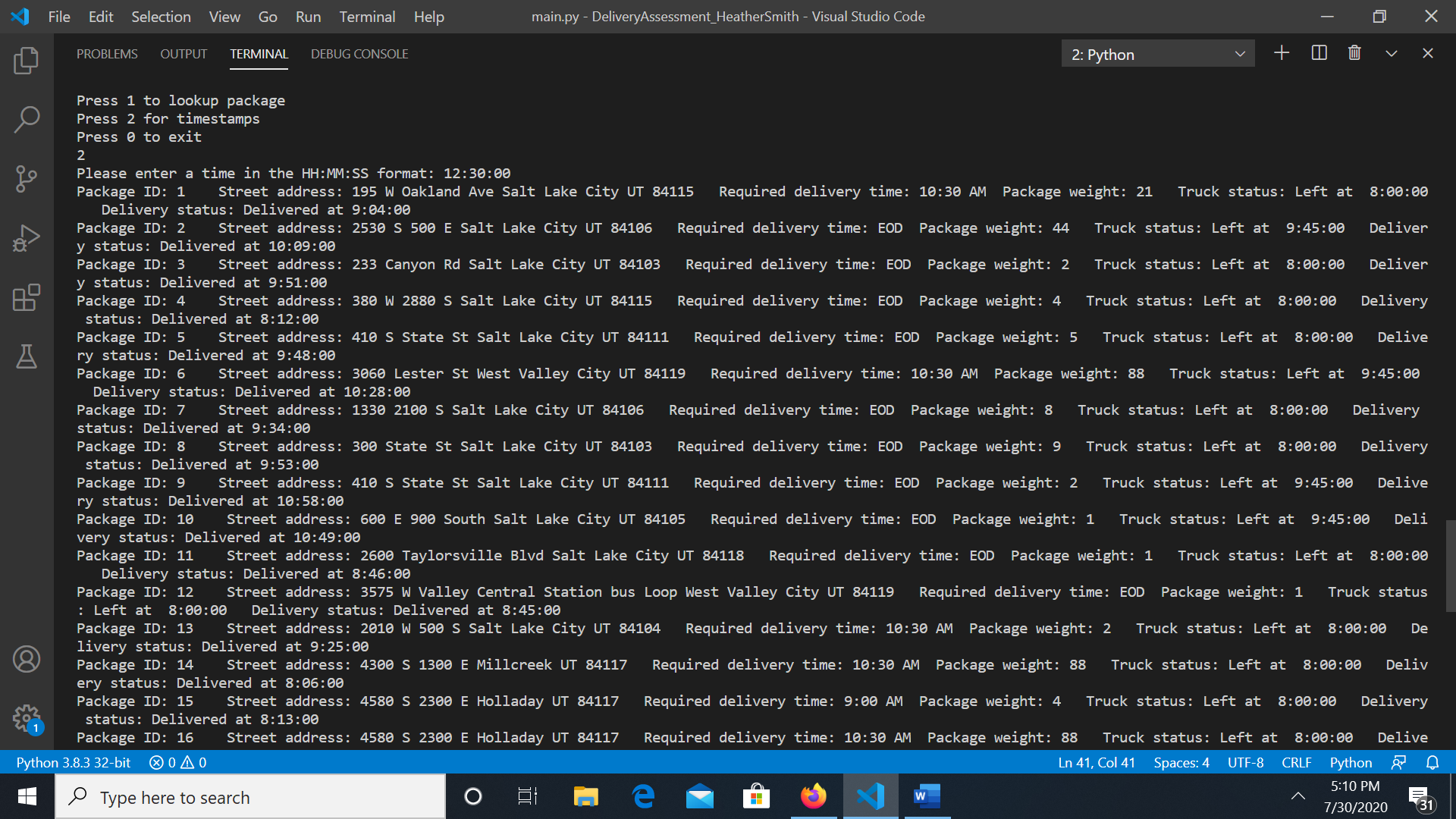
All packages between 9:35 am to 10:25 am

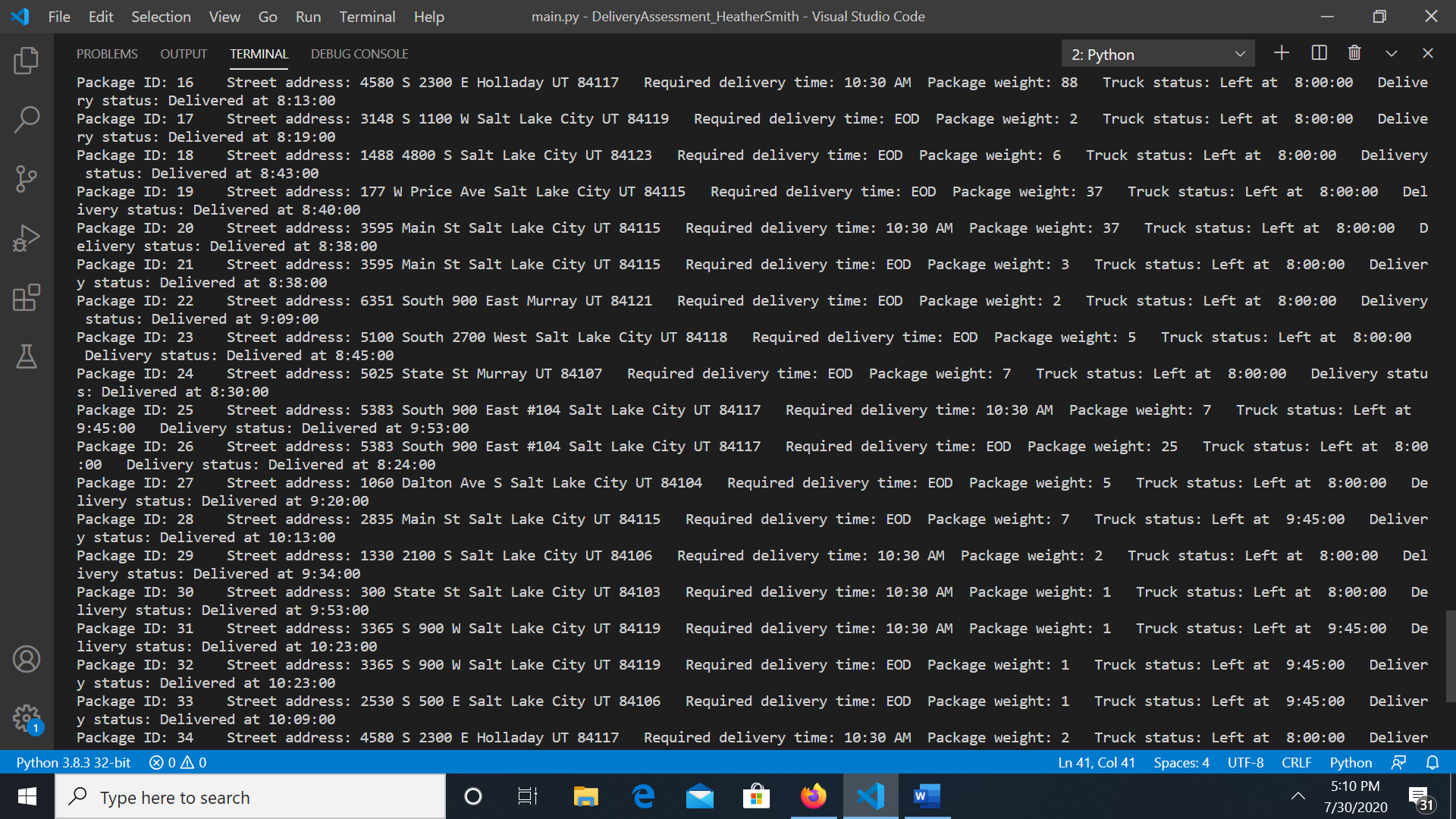


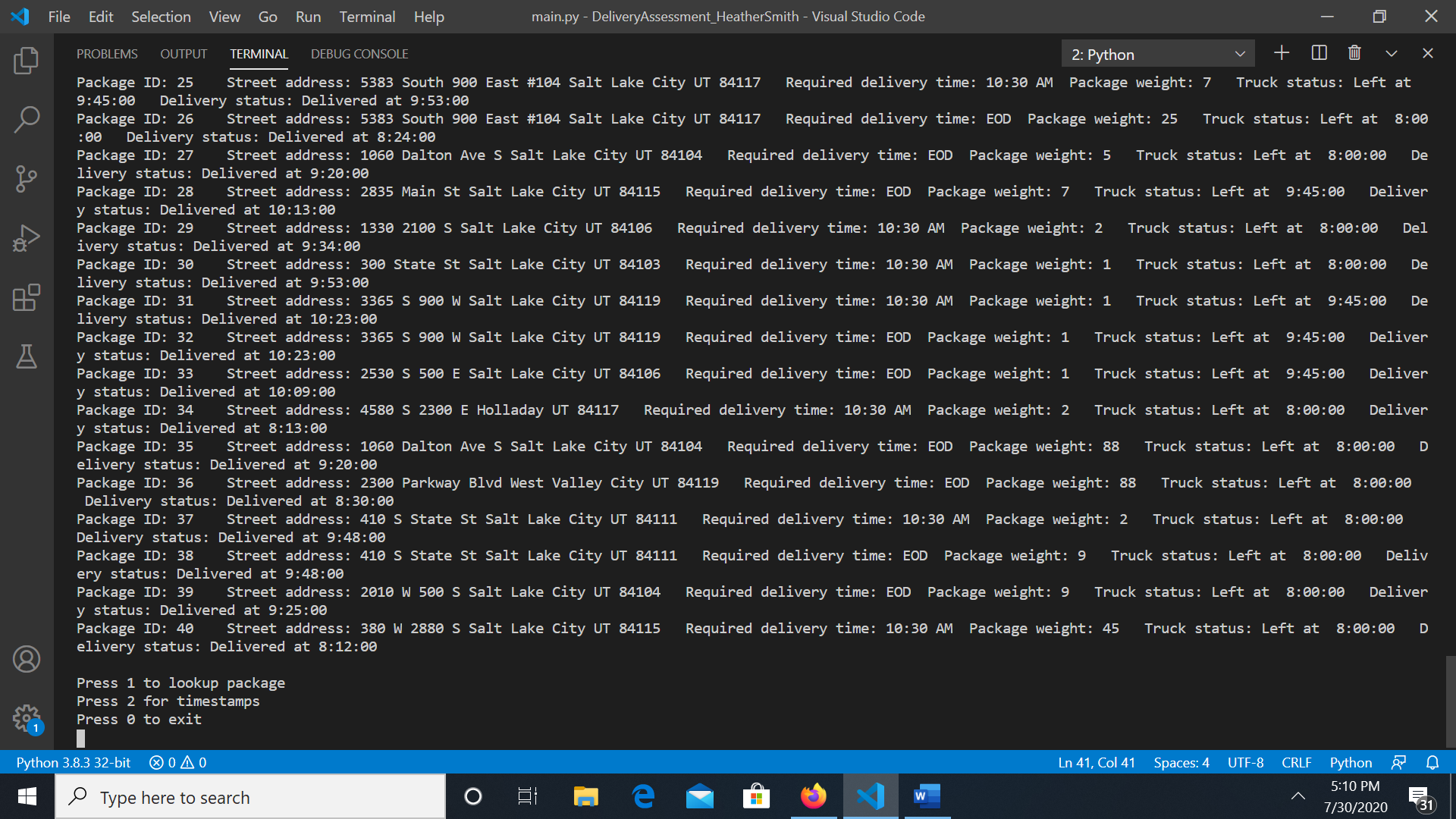




All packages between 12:03 pm to 1:12 pm

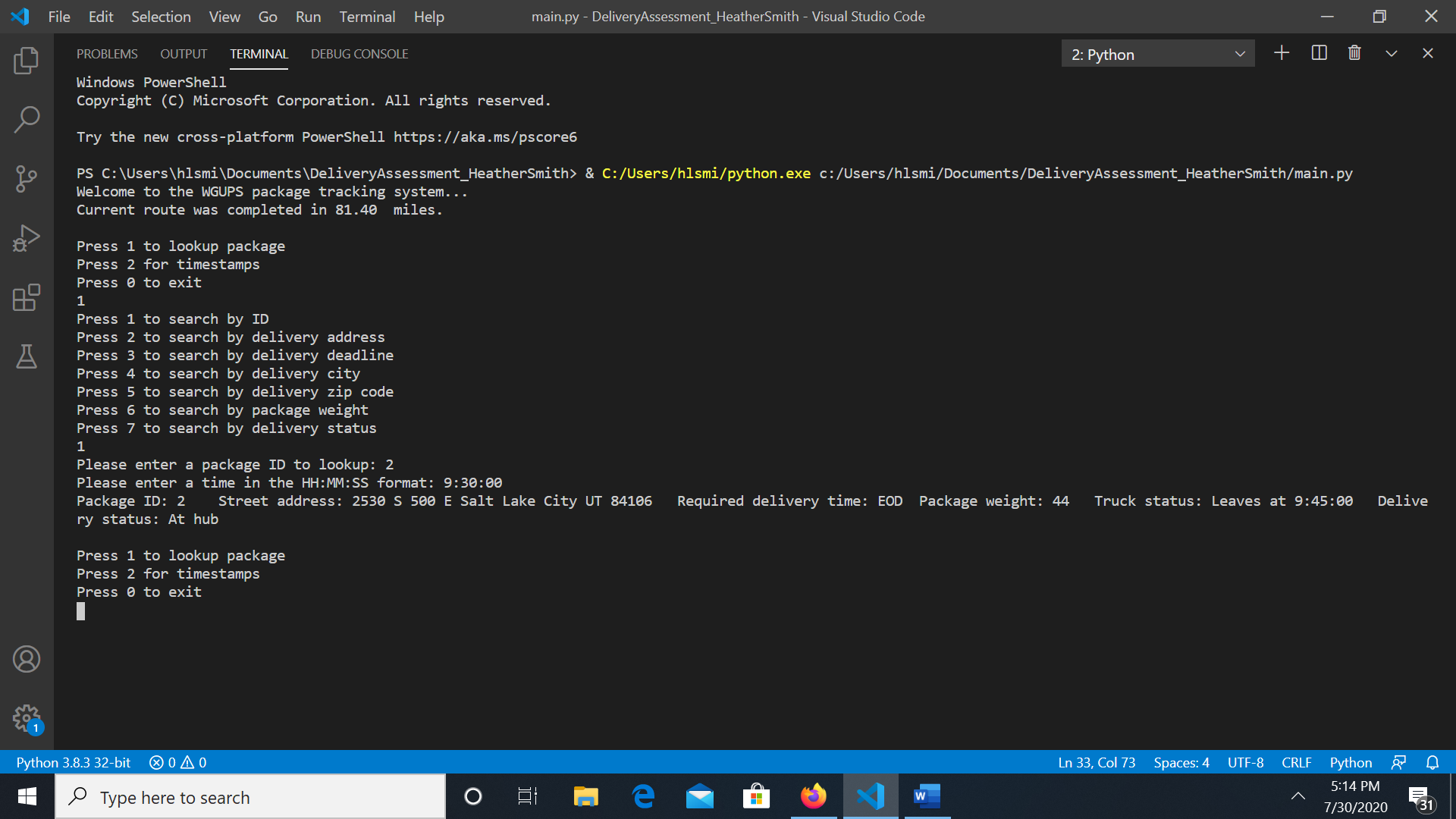




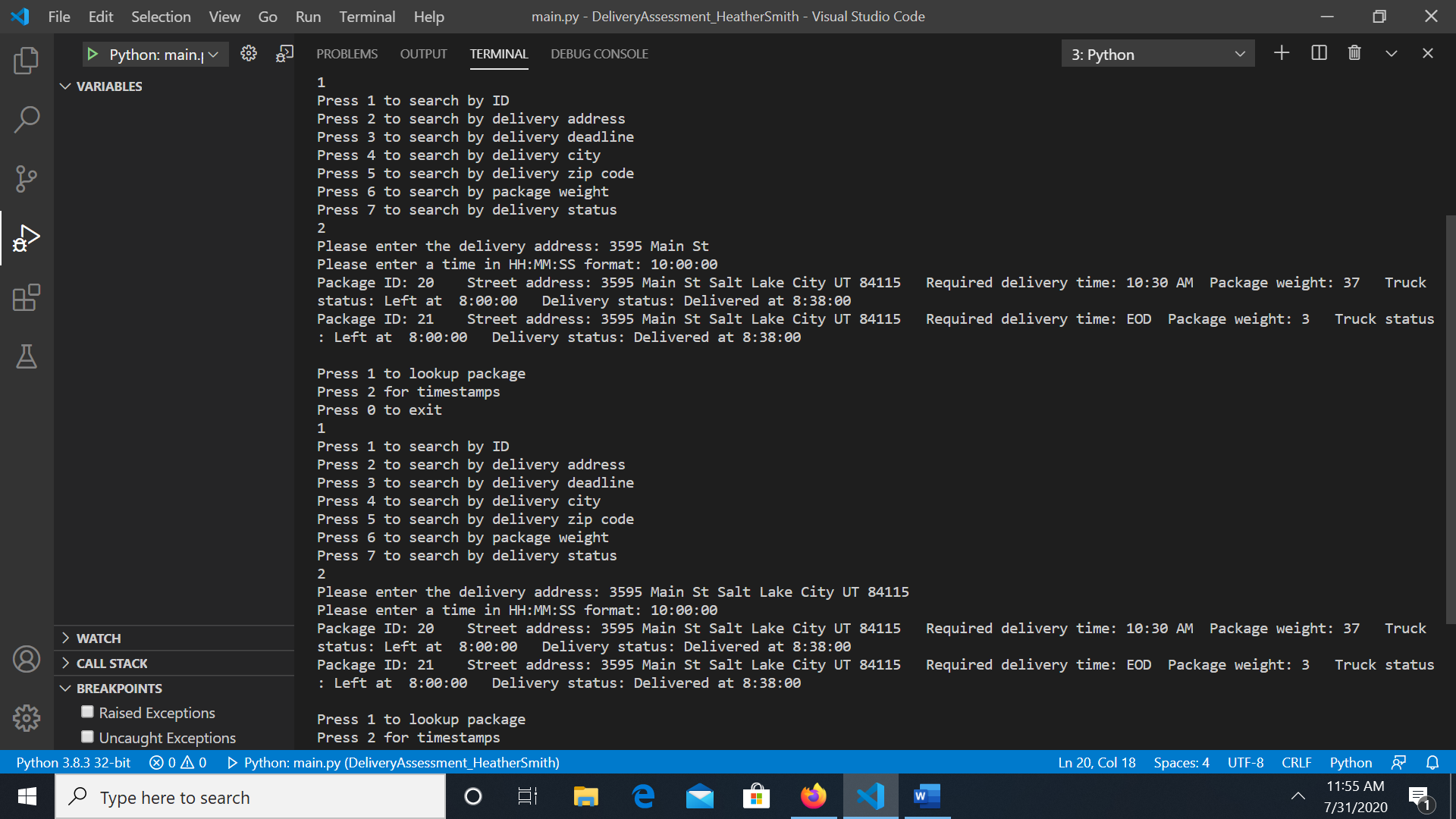


Complete Execution of code:

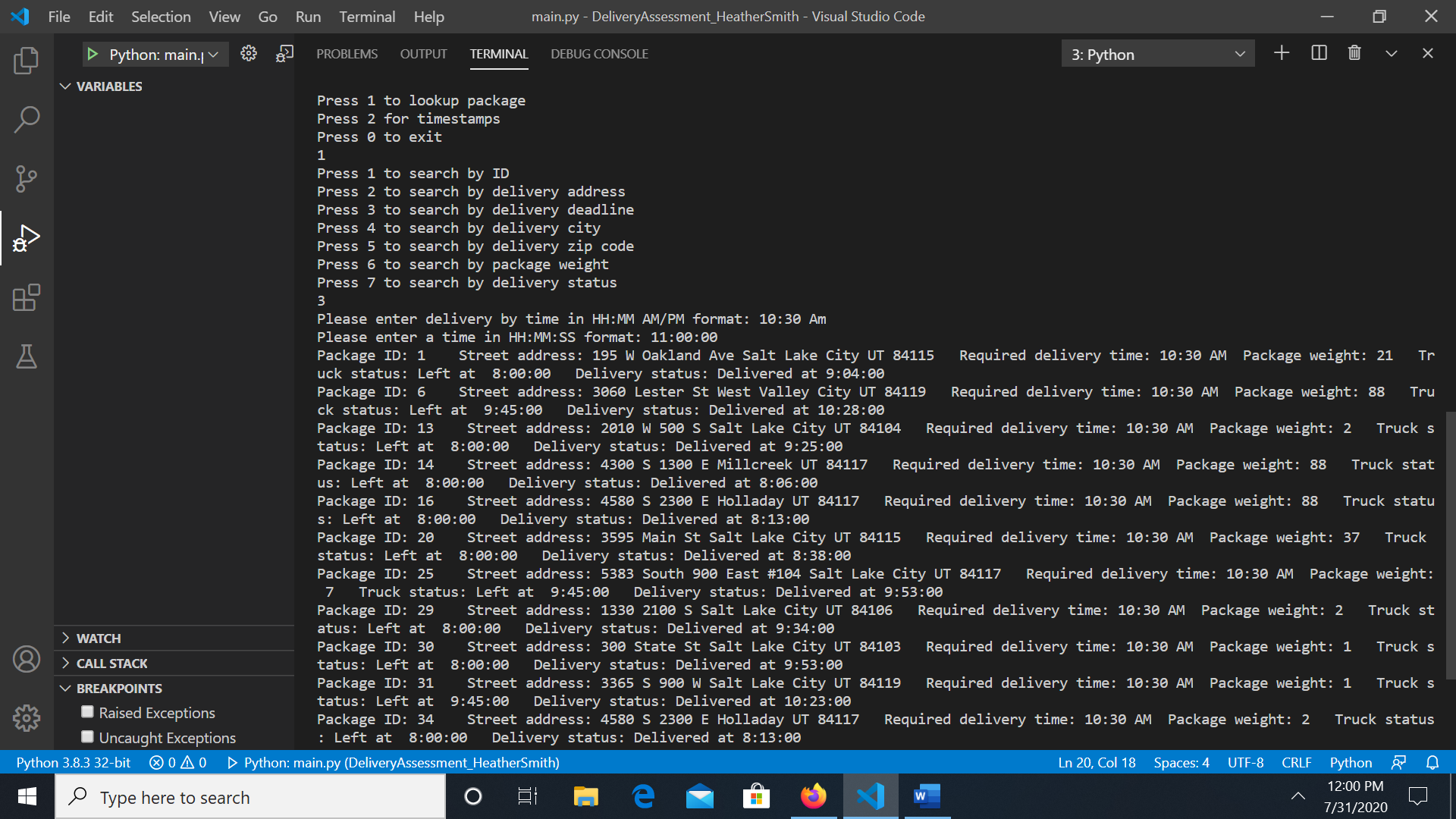
Search by package id:



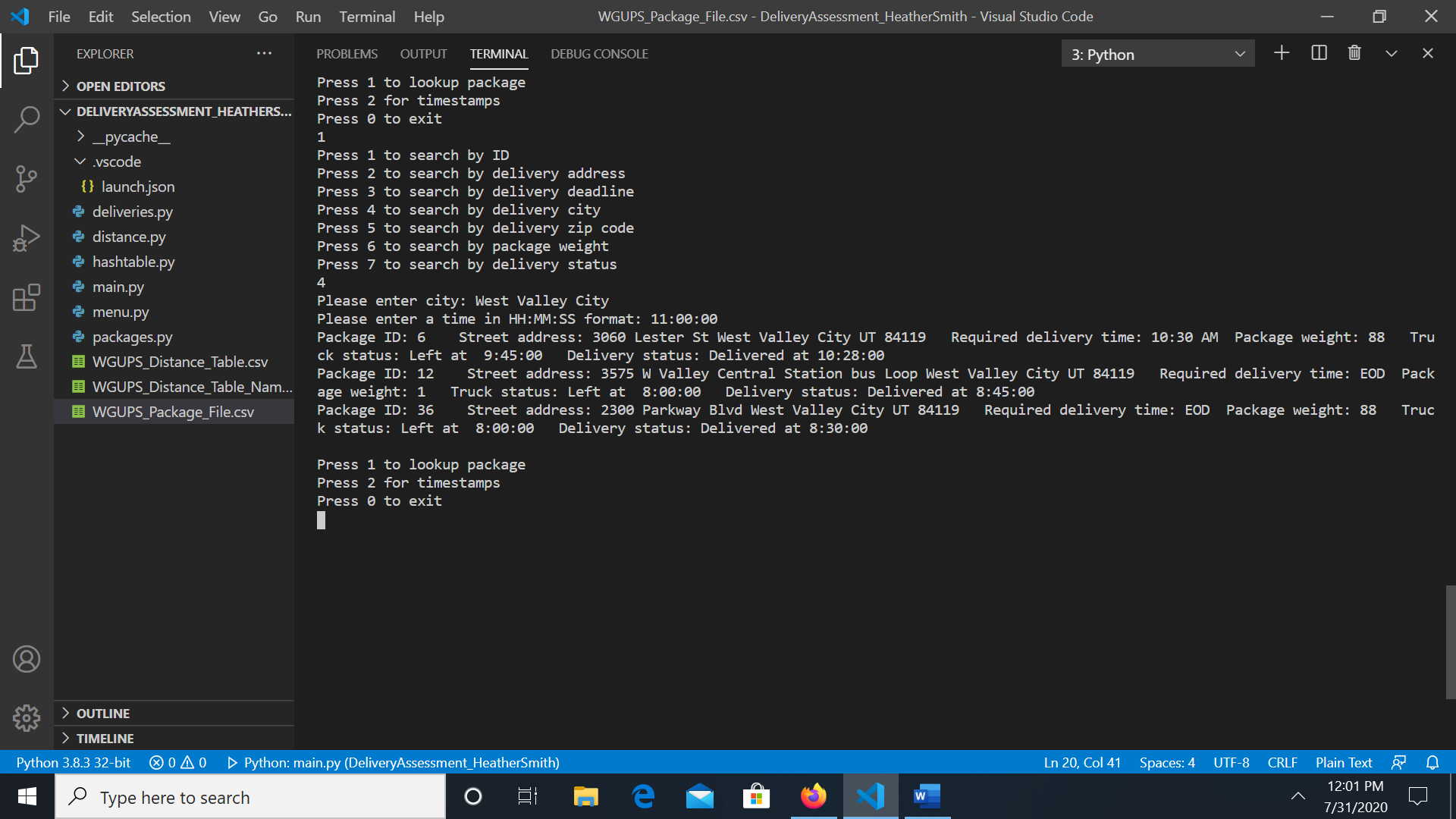
Search by address:



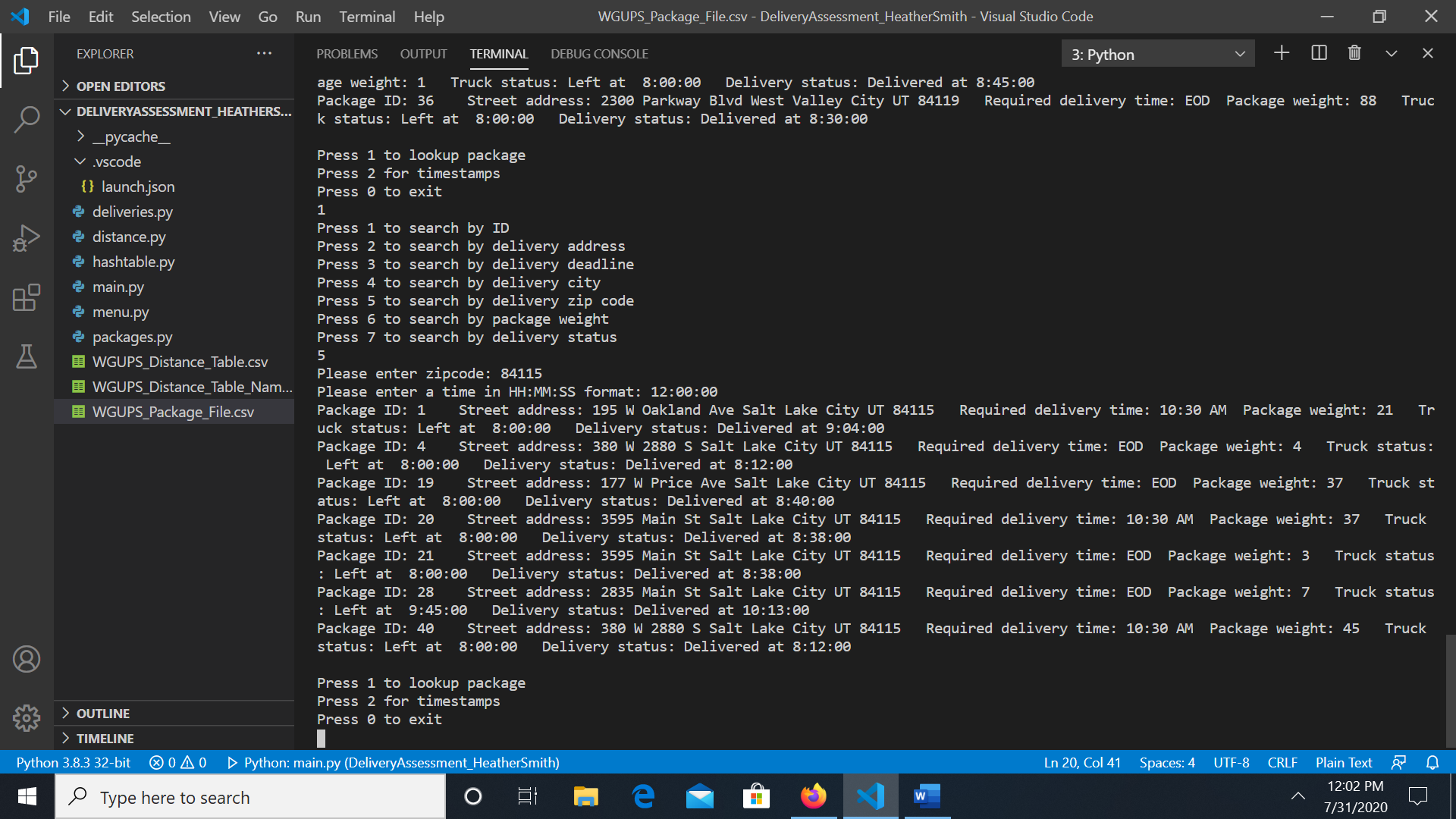
Search by delivery deadline:



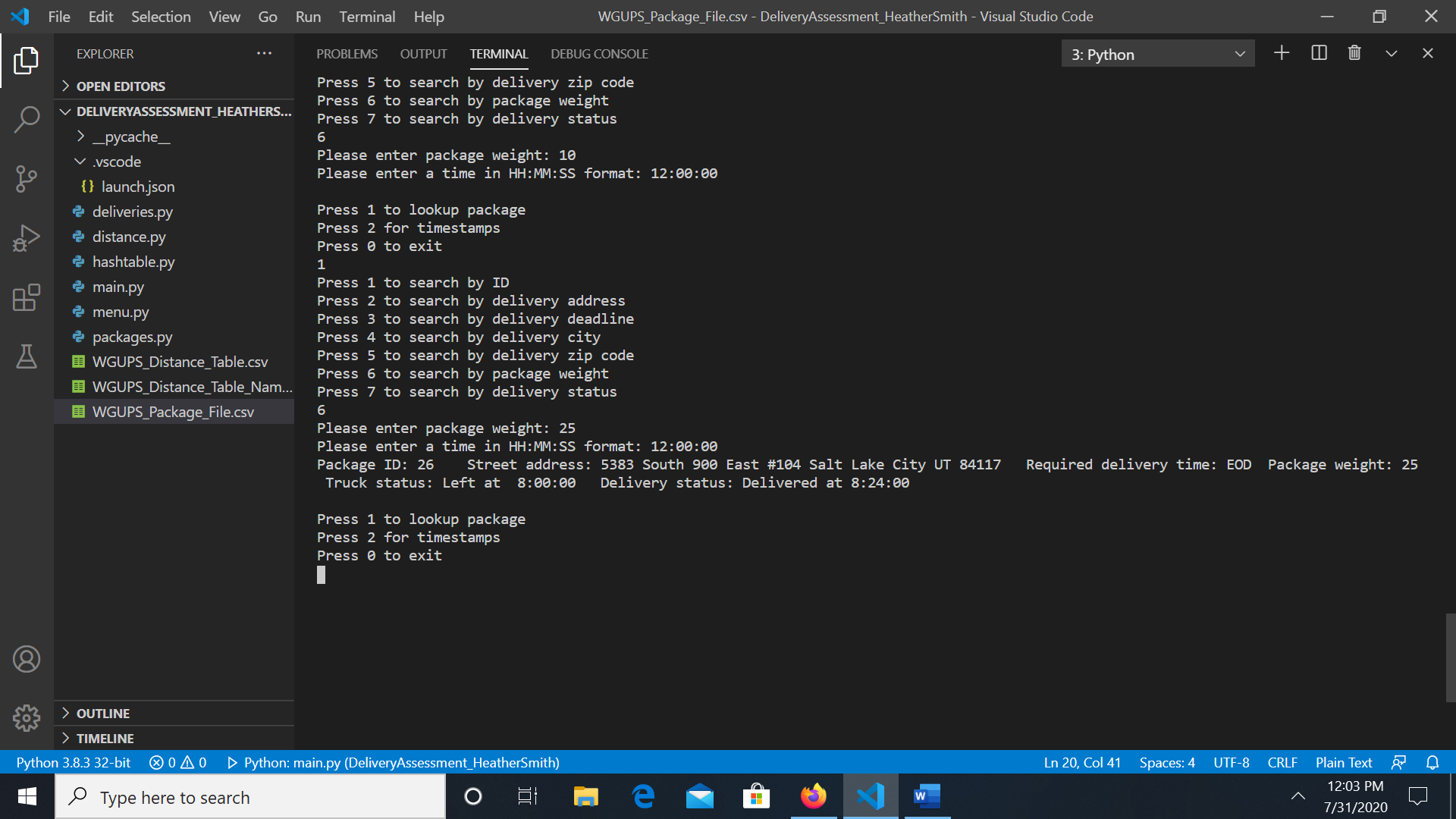
Search by city:



Search by zip code:



Search by weight:



Search by delivery status:

