**Core Algorithm Overview**

**Algorithm Identification:**

For my project, I have used python to develop a nearest neighbor algorithm. The algorithm simply selects the next node that has the shortest distance from the currently selected node. Since this greedy algorithm does not find the best possible path and only finds the best possible path at any given point, it is not a fully optimized solution. However, the algorithm is simple and easy to maintain with any future application changes.

**Pseudocode for primary algorithm:**

Compute route(location index)

Location index = 0

While not all locations have yet been visited

For every address in the distance dictionary

If the item is the first entry in the dictionary

List of distances = distance dictionary[address]

If length of the distance list > the location index

Distance = distance list[location index]

if distance is not zero or the float equivalent of zero

the distance is now the current smallest

Else the current smallest is now the next entry in the distance dictionary

If not the first entry in the dictionary

List of distances = distance Dictionary[address]

if the length of the value list-1 > location index

distance = distance list[location index]

if distance is not 0 or 0.0 and distance is less than the current smallest distance

if the address correlating to the distance has not been visited already

the distance is now the current smallest

location index = index of the distance within the distance list

other distance index = reverse search(location index)

if other distance < the current smallest

current smallest is now the other distance

location index is now the other distance index

compute route(location index)

reverse search(location index)

Distance list = distance dictionary[address]

For each distance in the distance list

If it is the first iteration, location has not been visited, and the distance is not zero

Distance is now the current smallest

Else if current smallest > distance

Current smallest is now distance

Index of smallest = index of the distance within the distance list

Return the index of smallest distance

**Integrated Development Environment:**

My application was developed on the PyCharm IDE, using the 2020.3.1 Professional Edition. PyCharm is a readily accessible IDE that includes various features such as code suggestions, debugging, and formatting options. PyCharm was installed and ran on a local machine, and the application is also designed to run on a local machine. This requires the application to be installed locally, however, this also creates an independence from network conditions and resources.

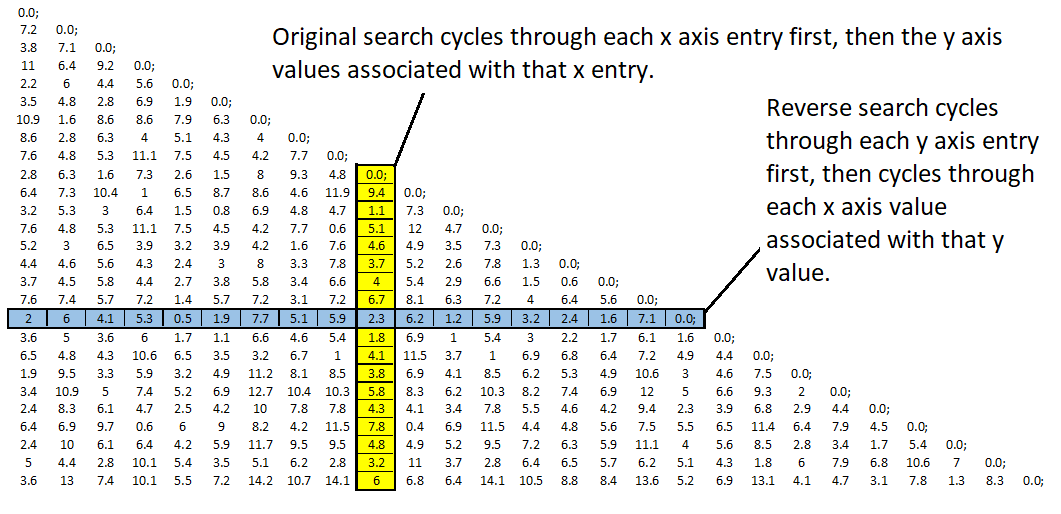
**Space Complexity, Time Complexity, and Big-O:**  
  
 The csv document for importing the package data is read in to count the rows and determine the size of the hash table. This portion of the application has a time complexity of O(N), where N is the number of rows in the csv file. The space complexity is also O(1) since this section functions as a simple counter where the value is only updated after every iteration.  
  
 Three lists are created: one for packages with special notes, one for packages that have a deadline, and one for all other packages. The csv document for distance data is read in a row at a time, adding that rows data to the corresponding list of packages. This block has a time complexity of O(N) and a space complexity of O(N).   
  
 The format method is used to remove spaces, commas, and quotations by cycling through each digit in a string. The time complexity of this portion is O(N) since it executes the loop for as many digits that exist in the string. If the string being analyzed contains no digits that require removal, the amount of memory required to store the data after the format is the same as before the format, hence, the space complexity is O(N).

Now, the distance dictionary is created, which will be used to look up distances by address and index. First, the address is used to find the list of values associated with that address, and then the index is used to locate the distance value for a destination. The csv file for distance is read into the application row by row, so O(N) time complexity and O(N) space complexity. The data in the first row, which is the header, is added to a separate list that will be used as a reference. The reference list can be searched by address to find which index corresponds to which address.  
   
 The location reference list is now simplified by removing any extraneous information and leaving only the street number of the address. The time complexity for this portion is O(N) since it relies on how many entries are in the list. Theoretically, the original string could already contain only the street number, so the space complexity would also be O(N).  
  
 The findNodeIndex function cycles through each element in the simplified location reference, which would in this case be the building number of an address, and compares that string to the passed value. If the building number is in the passed value, the index is returned. The time complexity of this function depends on how many items are in the list that require comparison. Assuming there is no match this would mean a time complexity of O(N). Space complexity is O(1), since the single matching entry will be returned.

An algorithm to calculate the nearest node from a given nodes passed in as a parameter is next. The first loop is a while loop that compares the number of locations that have already been visited to the total number of locations. The time complexity of this segment would be O(N), and the space complexity would be a constant O(1).

The next step of the algorithm cycles through an integer from 0 to the length of the dictionary. The integer will be used as a location index. Each dictionary entry contains a list of distances, as floats, associated with other addresses. Through each iteration of the integer, the corresponding list of distances is retrieved from the dictionary, and the integer is now used to obtain the distance at that index for each address. The value at that index is compared between each dictionary entry and the smallest is saved to a variable. Since this portion of the algorithm cycles through as many times as there are locations, the time complexity is O(N). Due to this portion being nested within the previous portion, this results in a time complexity of O(N^2). The space complexity will also remain constant since none of the data saved here grows relatively to the input or the size of the dictionary.

Inside of the most outer loop the reverse searching method is now called to search the distance dictionary, which essentially acts as a distance matrix, in reverse. This is required since not all the distance values are contained for a given address on one axis. Like the original algorithm, the time complexity is O(N), and since it is also nested within the first loop, this results in a total time complexity of O(N^2). The space complexity is constant since the same amount of data is used regardless of how many times the loops are performed.



The space complexity of the dictionary increases proportionally to the number of addresses. An increase in addresses would not affect the time complexity of a distance list look up if the address is already known. Since using the address key will pull the distance list without having to iterate through each location. However, for the primary algorithm the time complexity would be affected by an increase in addresses since the algorithm cycles through each address in order to compare each distance value. The time complexity for the algorithm stated previously increases proportionally to the number of addresses that are in the operational radius of the delivery station.

An increase in the number of cities would have the same affect if that increase in cities also increased the number of addresses. If the number of cities increased but the number of addresses stayed the same there would be no effect on any complexities. Likewise, the number of packages has no effect on the space-complexity or time-complexity of a dictionary look-up.

Following the primary algorithm is a method for sorting and combining the packages lists. The packages that have delivery deadlines will be prioritized and appended first. The outer loop consists of a for loop that iterates through all the locations, resulting in a time complexity of O(N). Within this outer loop are three other loops which iterate through each package, again this for loop has a time complexity of O(N). The combine time complexity of the other loop and inner loop is O(N). The space complexity of this block of code is O(N) since the input lists, with N item, will be appended onto the final package list.

The next method is the load method, designed to load truck objects with packages. This method begins with an if condition to check if there are remaining packages to deliver. If packages are undelivered, the method continues with more if conditions for special circumstantial packages, and finally, loads all other packages to the max capacity of each truck. Since this method cycles through every package, the time complexity is O(N). The space complexity is O(1), still the truck capacity is a constant 16 packages.

After the load method is a method of which is used to find the distance between two locations using the corresponding location indexes of each. Since this method iterates through each entry in the distance dictionary to find the matching address, the time complexity is O(N). Space complexity of this method is a constant O(1), since only the distance is found and returned.

The method used to the deliver the packages is next in the application. This method begins with a outer while loop of whose condition is that the truck inventory is not empty. Depending on how many packages are in the trucks inventory this loop will iterate proportionally. Since the maximum capacity of a truck is 16, that means the maximum amount of iterations for this loop is 16. Hence, the time complexity is a constant O(1). Again, since the maximum number of packages that can be delivered is 16, when this method appends the list of delivered packages the maximum number of additions is 16. The space complexity of this method is a constant O(1).

The final method in the application is the snapshot method which intakes a parameter from the user, a time, and outputs the status of all packages at that time. In order to display the current status of each package, the delivered packages list is iterated through and the delivery time saved to each package hash object is compared with the input time. The time complexity for this method is O(N) due to the list iteration, and the space complexity is O(1) since the few variables involved are updated each iteration.

My application is concluded with a while loop, the condition of which is to iterate as long as the length of the delivered packages list is less than the length of the sorted packages list. During each iteration truck one and truck two are loaded and the packages then delivered. The time complexity of this loop is O( N) and the space complexity is O(1). Since the time complexity of the load is O(N) and the time complexity of the deliver method is O(1), the final time complexity of this block is O(N^2).

There exist a few separate python documents related to my python application: a truck document for the truck object, a hash creation document for the hash table, a hash object document for the hash object, and a clock object for tracking time. The truck document has a time complexity of O(1) due to load method having the only loop. The loop in the load method can only execute up to a maximum of 16 times due to truck capacity. The space complexity of the load method is also O(1) due to this same capacity limitation.

The hash creation object document contains all O(1) time complexity and space complexity methods except for the initialization method. The initialization method loops through the passed integer to indicate the size of the hash table, and on each iteration the hash table is expanded. The time complexity of this portion of the hash creation class is O(N) and the space complexity is also O(N). The hash object class and the truck class contain no methods that have a time complexity or space complexity greater than O(1).

**Scalability and adaptability:**

In regards to the scalability of the application, the amount of packages or addresses served can easily be expanded upon using the csv files. For packages with special notes attached to them, the code will have to be updated in order to perform appropriately for these specific circumstances. The code will also have to be updated if the number of trucks or drivers available changes. However, since the program uses a simple greedy algorithm, it will be easy to adapt for any future unforeseen needs or changes.

**Software efficiency and maintainability:**

Although the greedy algorithm used to compute the delivery routes will not result in the most optimized solution, it is simple and makes any new software requirements easy to roll into the application. The entire application has an efficiency of O(N^2). For increased efficiency, however, the algorithm would need to be replaced with one that has a better time complexity. For use in small to medium coverage areas this application will suffice, although in the most densely populated of areas runtime and data storage may become an issue. As far as maintainability goes, there is little to maintain except the condition of the local host machine running the application and the csv files used to import the package and distance data. To change the package or distance data, a user must simply open the document and update it with the associated information while maintaining its basic structure.

**Self-adjusting data structure:**

For the storage of the package data imported from the csv file I utilized my own hash table of hash objects. The hash table allows a direct look-up of specific package data without having to always iterate through each object in order to find it, providing a time complexity of O(1). The downside of using this type of data structure is risk that collisions will occur when adding hash objects to the hash table. Since this would only occur if there were two packages assigned with the same package id, or if there was an error in the csv file containing the package ids, the risk is minimal.

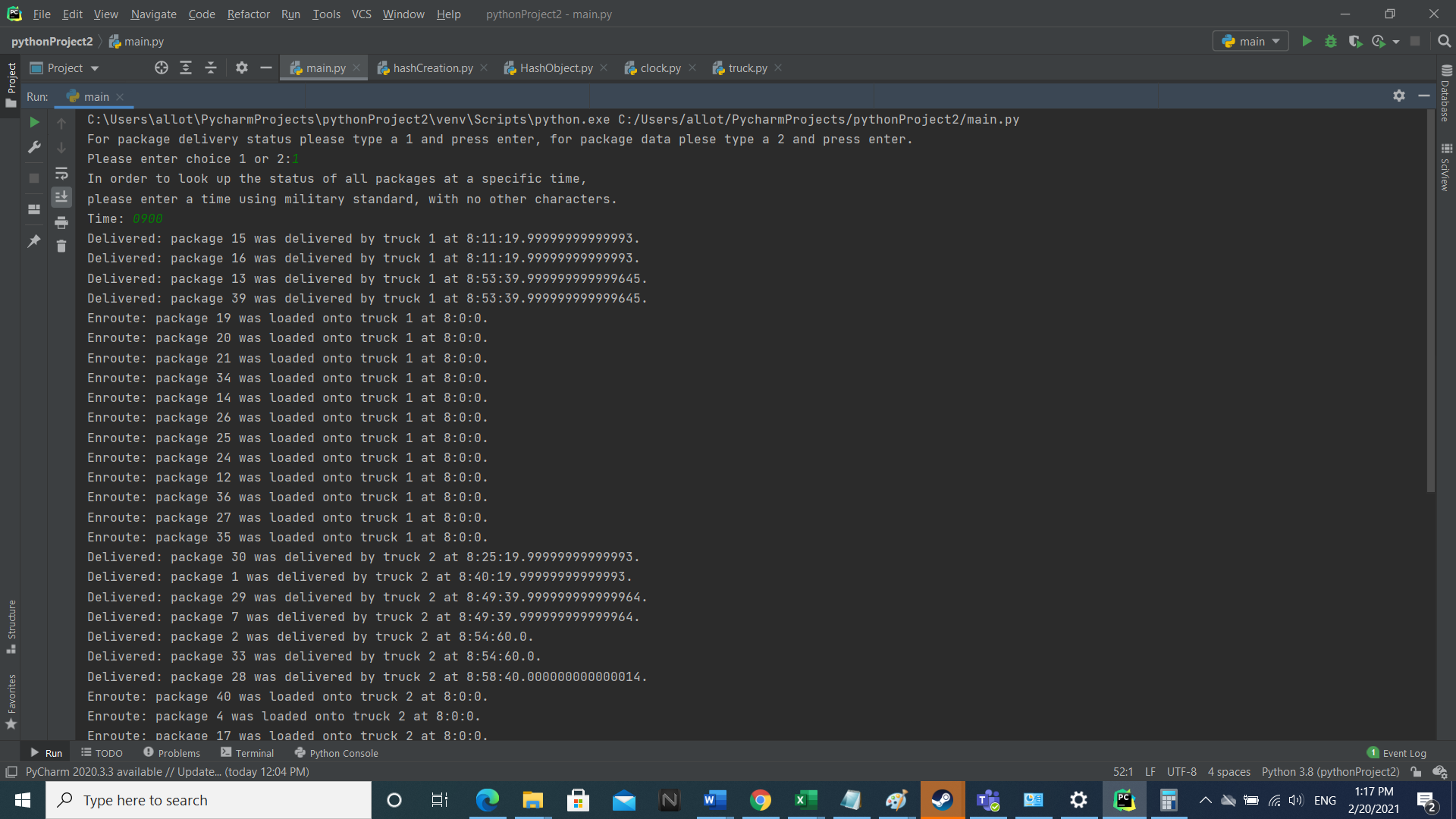
An object to represent the package is created in order to be inserted into the hash table. This hash object contains the package data which will be stored in the table. The hash object contains: the package id (integer), the delivery street address (string), the city, state, and zip-code (all strings). Also, the delivery deadline is stored (string), the weight is stored (integer), and delivery notes are stored (string). Finally, a time is stored for the delivery time (string), the time the package was loaded onto the truck (string), and the truck id of the truck it was loaded onto (integer).

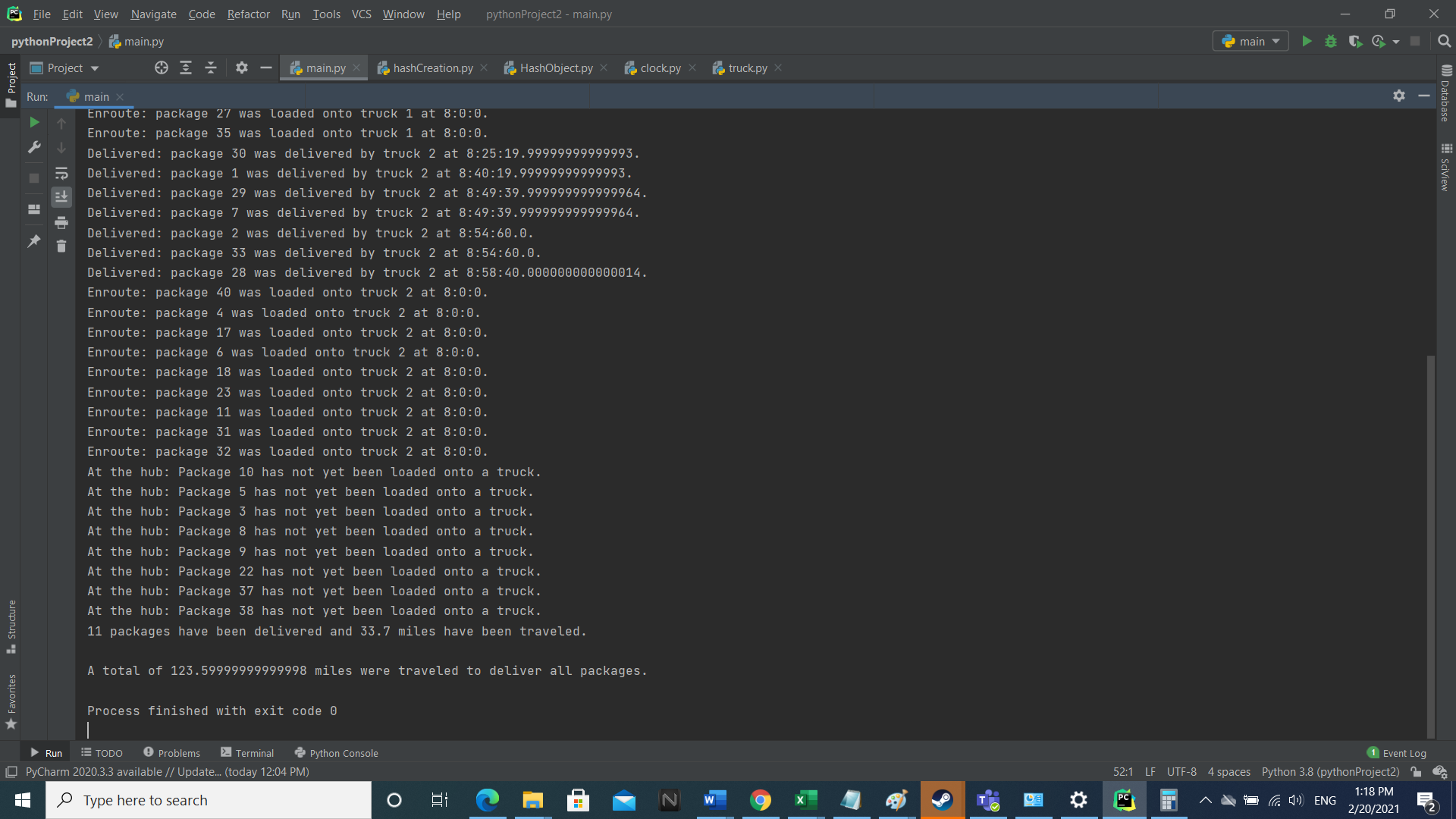
An increase in the number of packages does not affect the time complexity of the look-up function, however, the space complexity O(N) is directly proportional to the number of packages in the hash table. The more packages in the table the larger the N will be. An increase in the number of trucks being used for delivery does not have either the space complexity or the time complexity of the look up function. The truck id is simply stored as an integer within the hash object and has no other function in the hash table. The number of cities has no affect on the hash table look-up function time complexity or space complexity since the city associated with each package is simply a string stored within the hash table object.

**Explanation of data structure:**

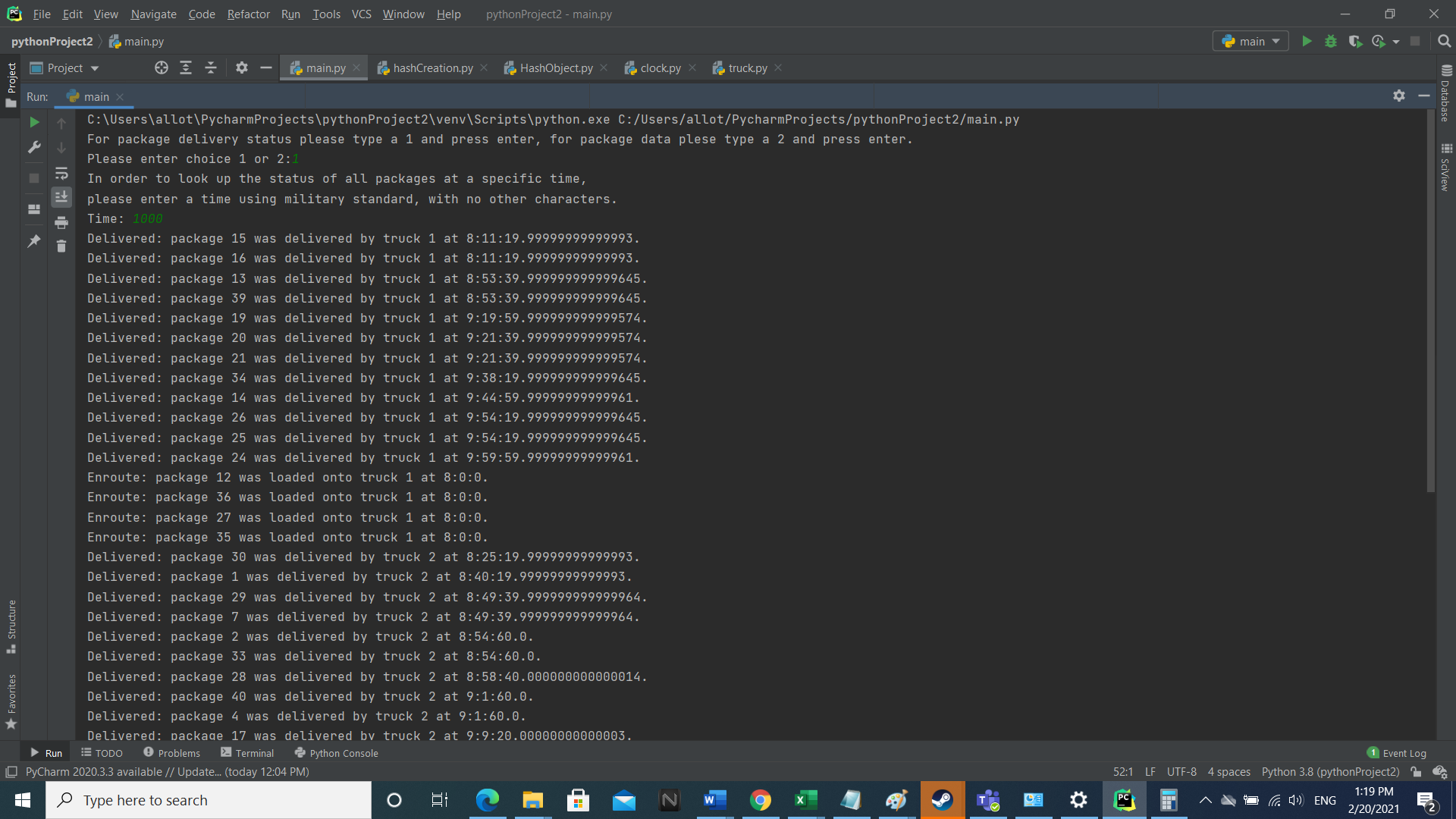
The hash table creation class is responsible for initiating the hash table itself, it uses a passed integer to determine the initial size of the table. Additionally, the class also contains the following methods: insert, setDelivery, setTruckId, setOnTruck, setDeliveryMiles and the corresponding getter methods. Each of the aforementioned methods takes a passed parameter and updates the hash object with the information. The hash object allows the storage of data such as: the time the package is delivered, the time the package is loaded onto the truck, total miles travelled by the truck that delivers the package, delivery deadlines, physical attributes and destination information.

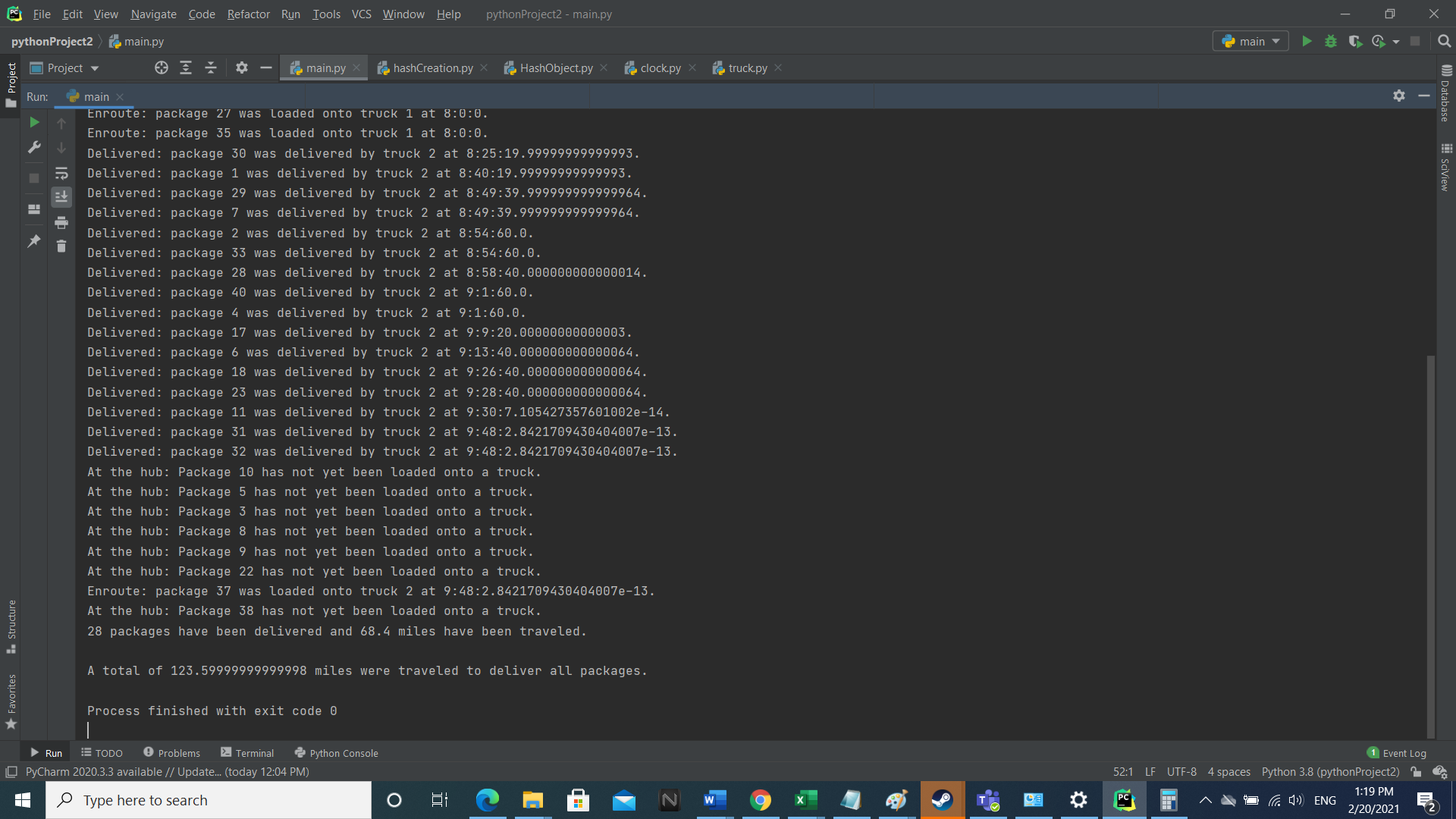
**Screenshot for 0900:**



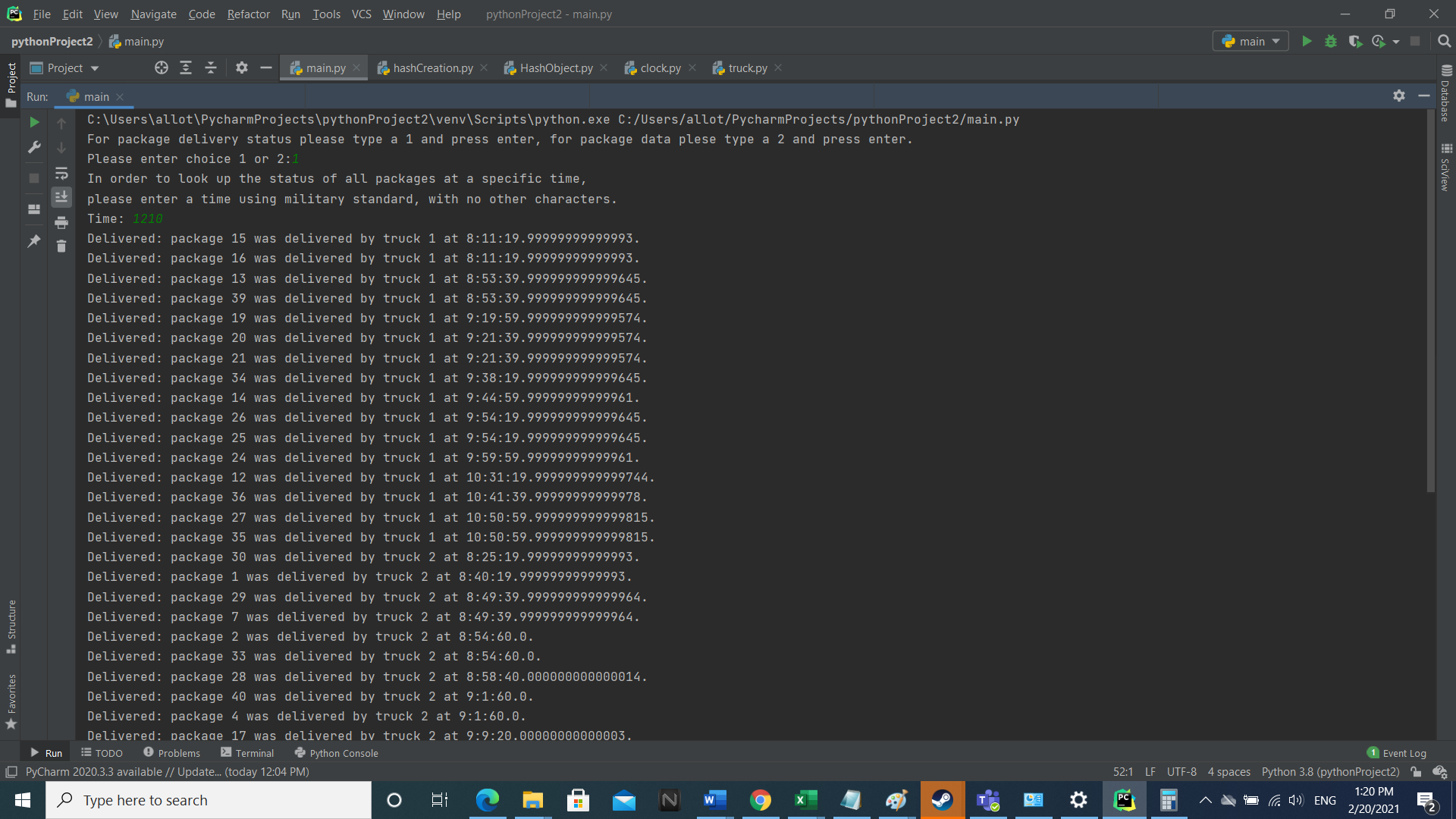


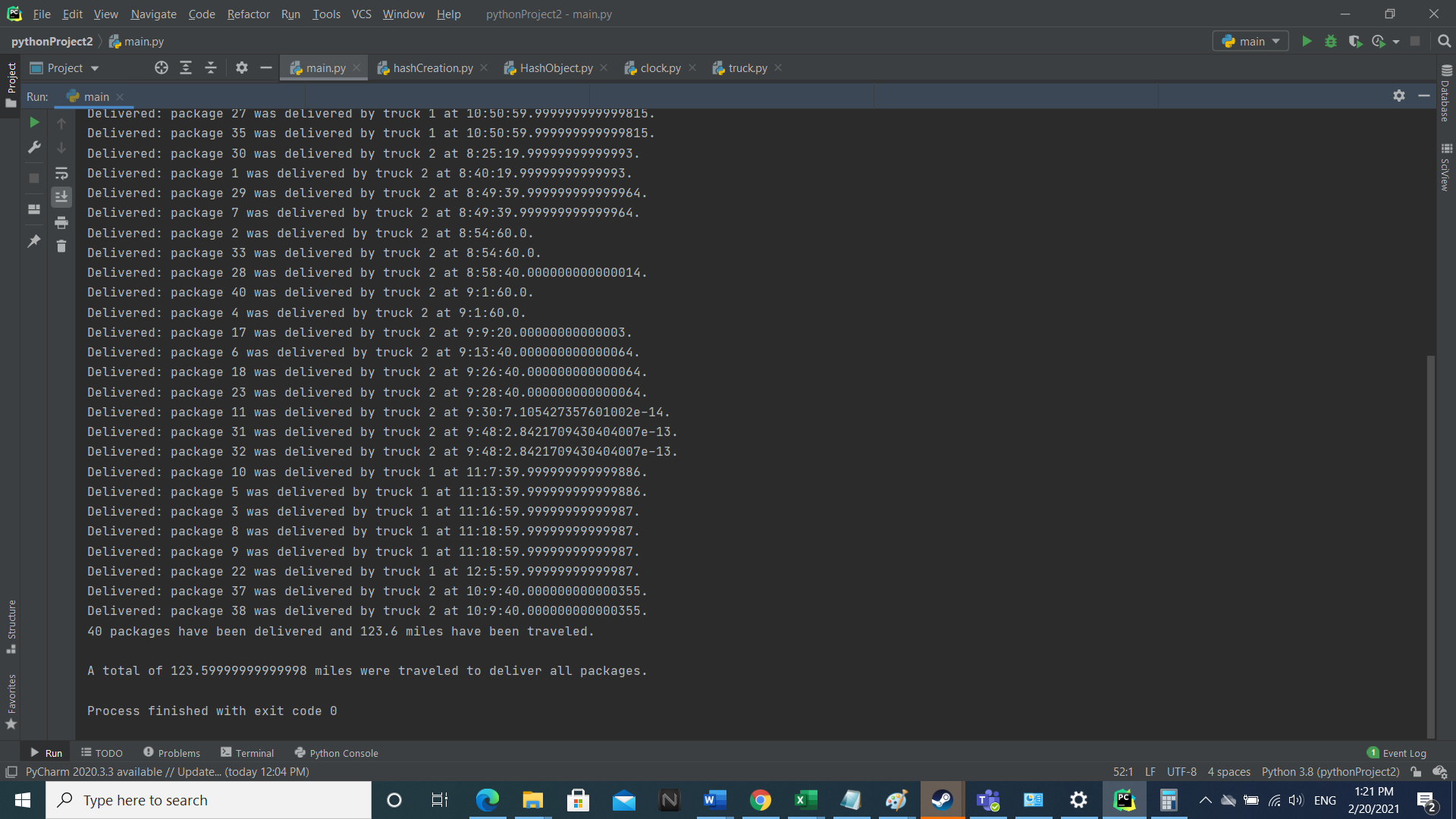
**Screenshots for 1000:**



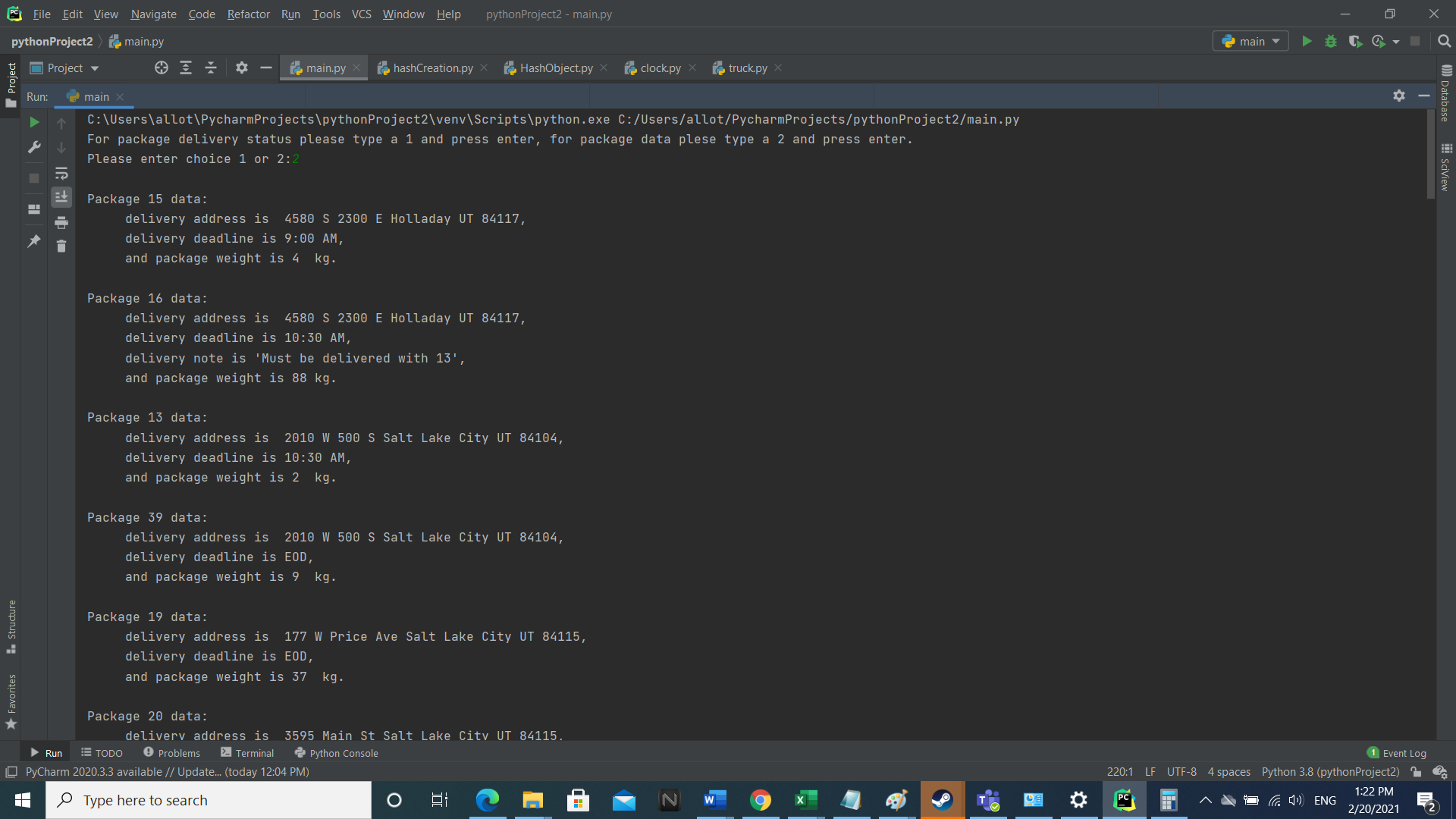


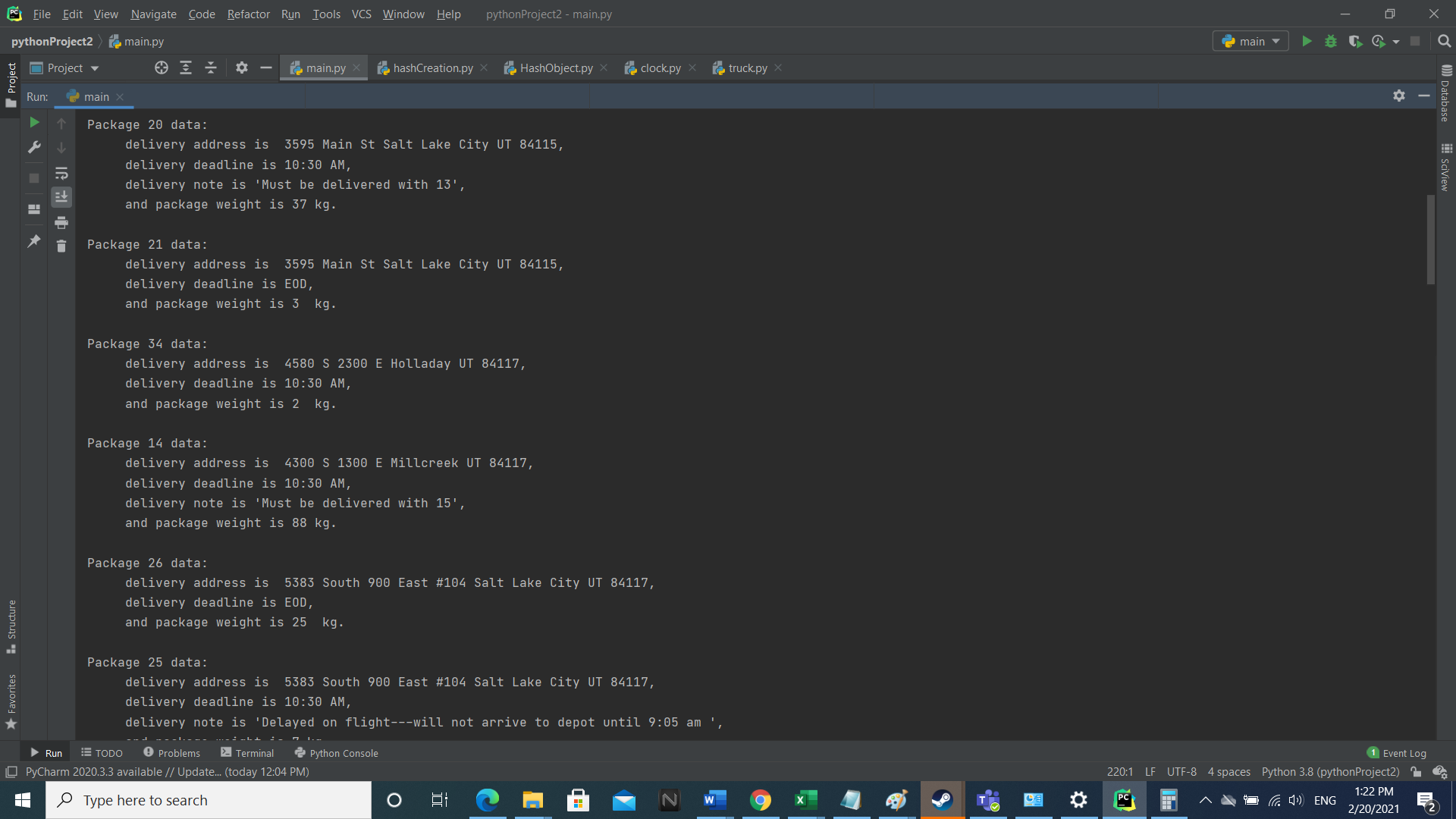
**Screenshots for 1210:**

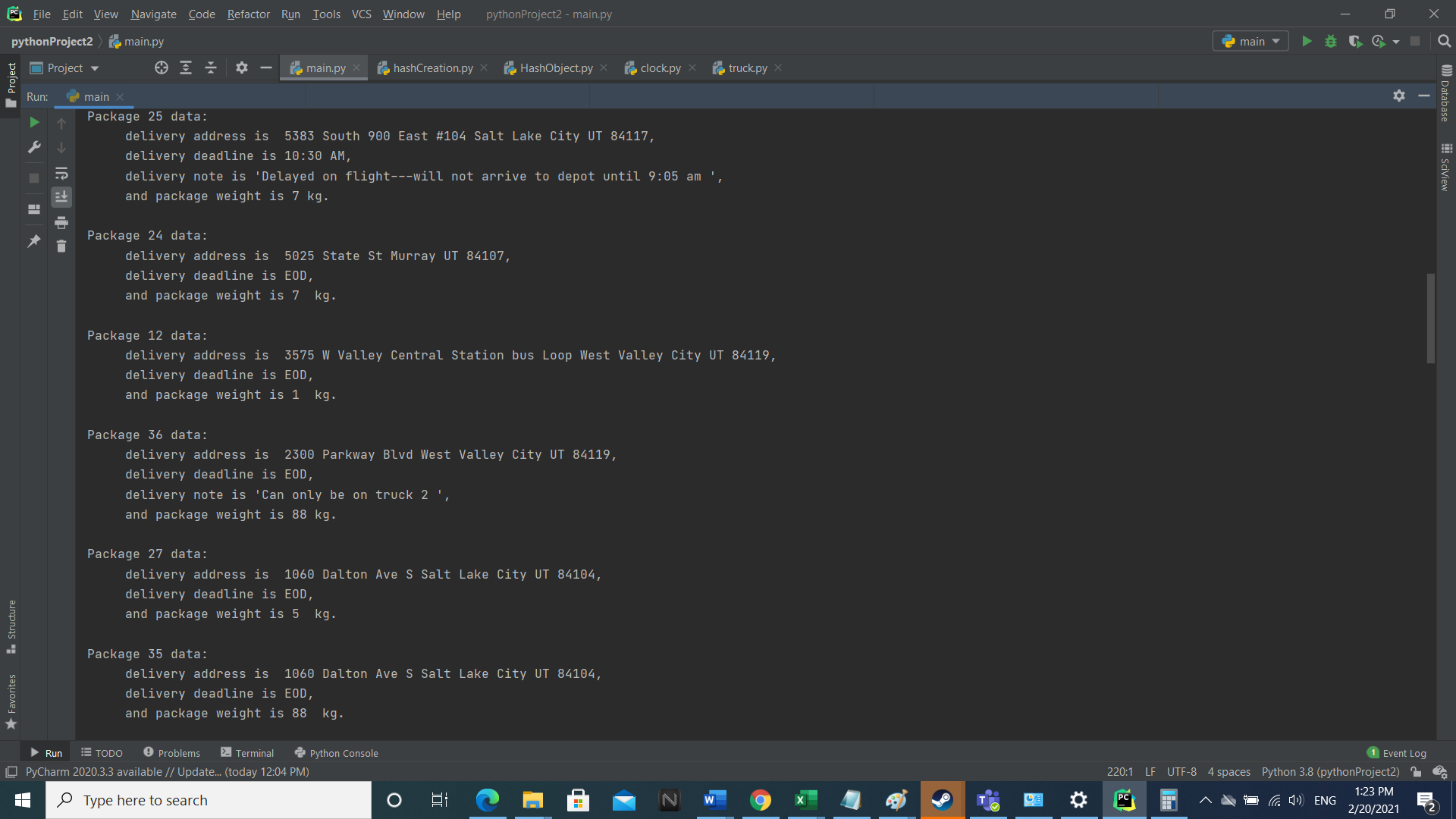


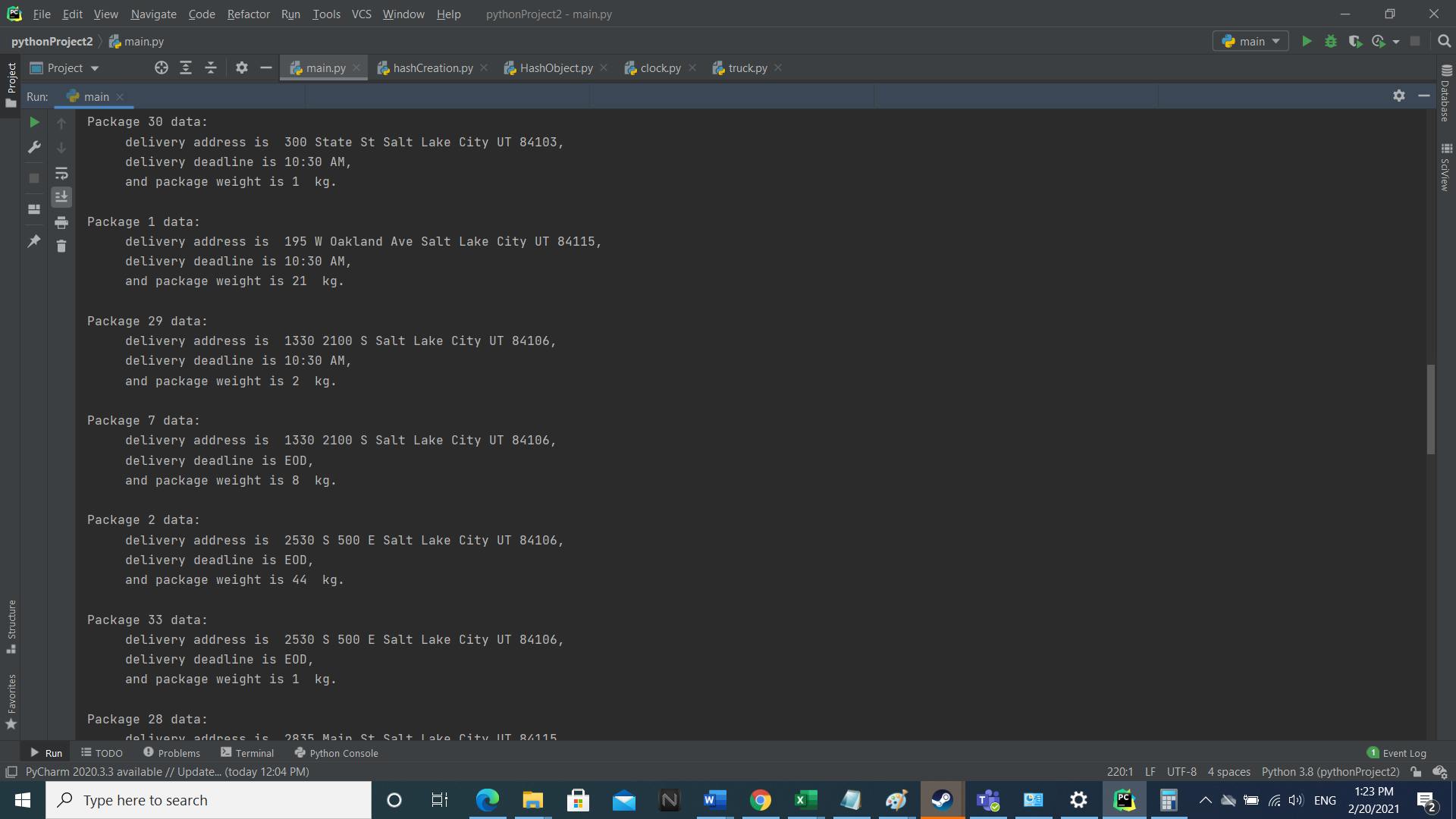


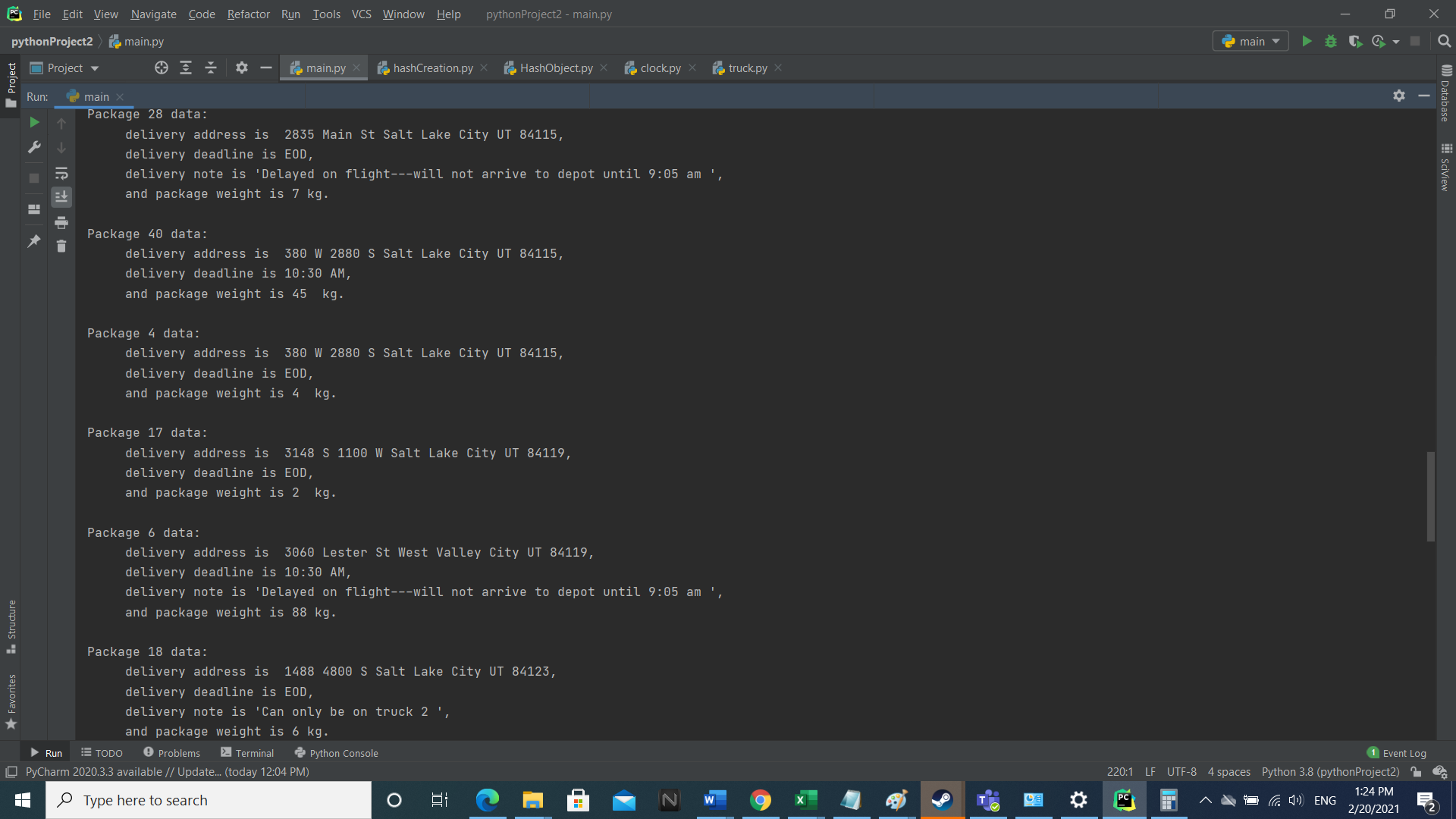
**Screenshots for package data look-up:**

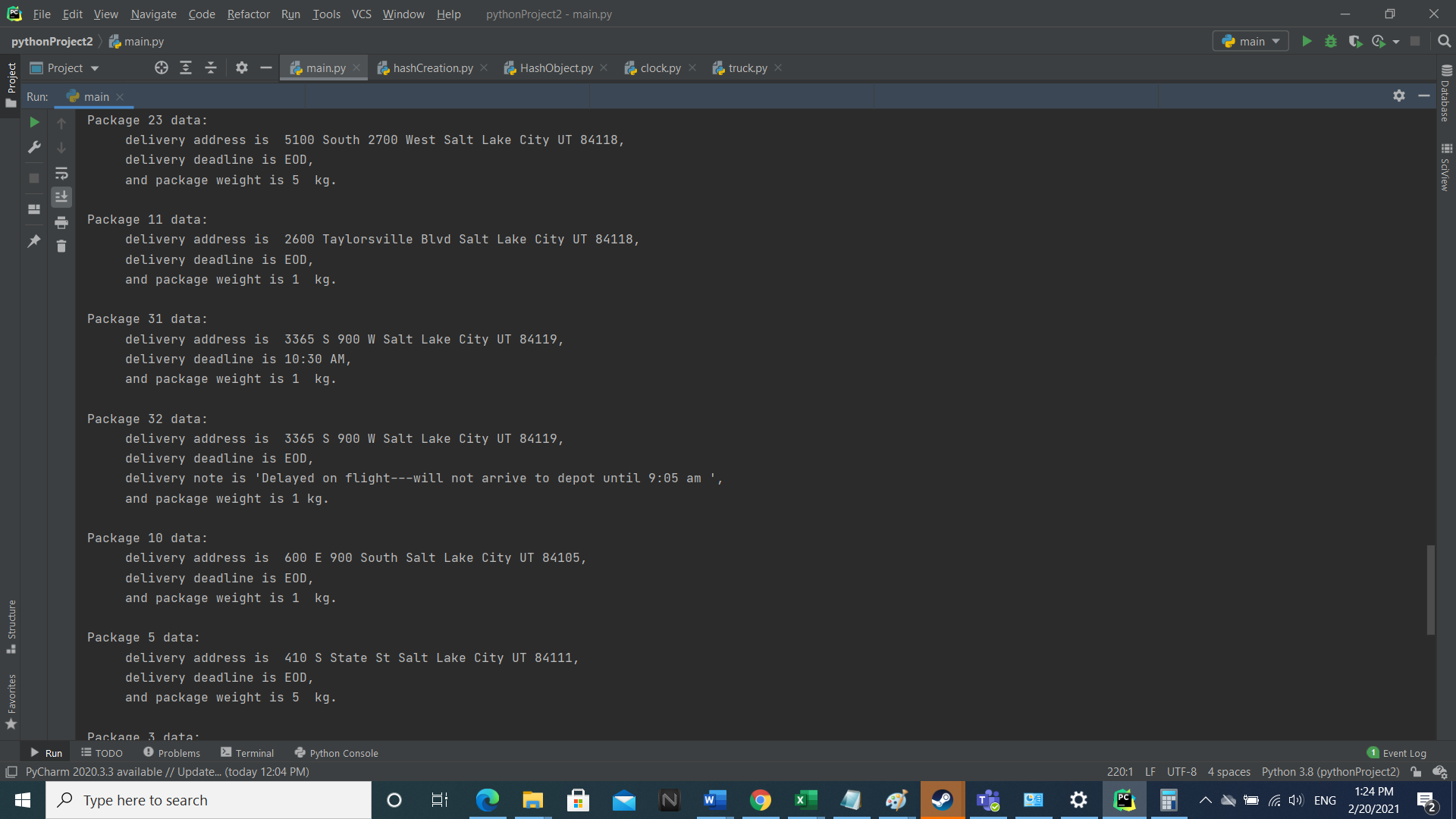


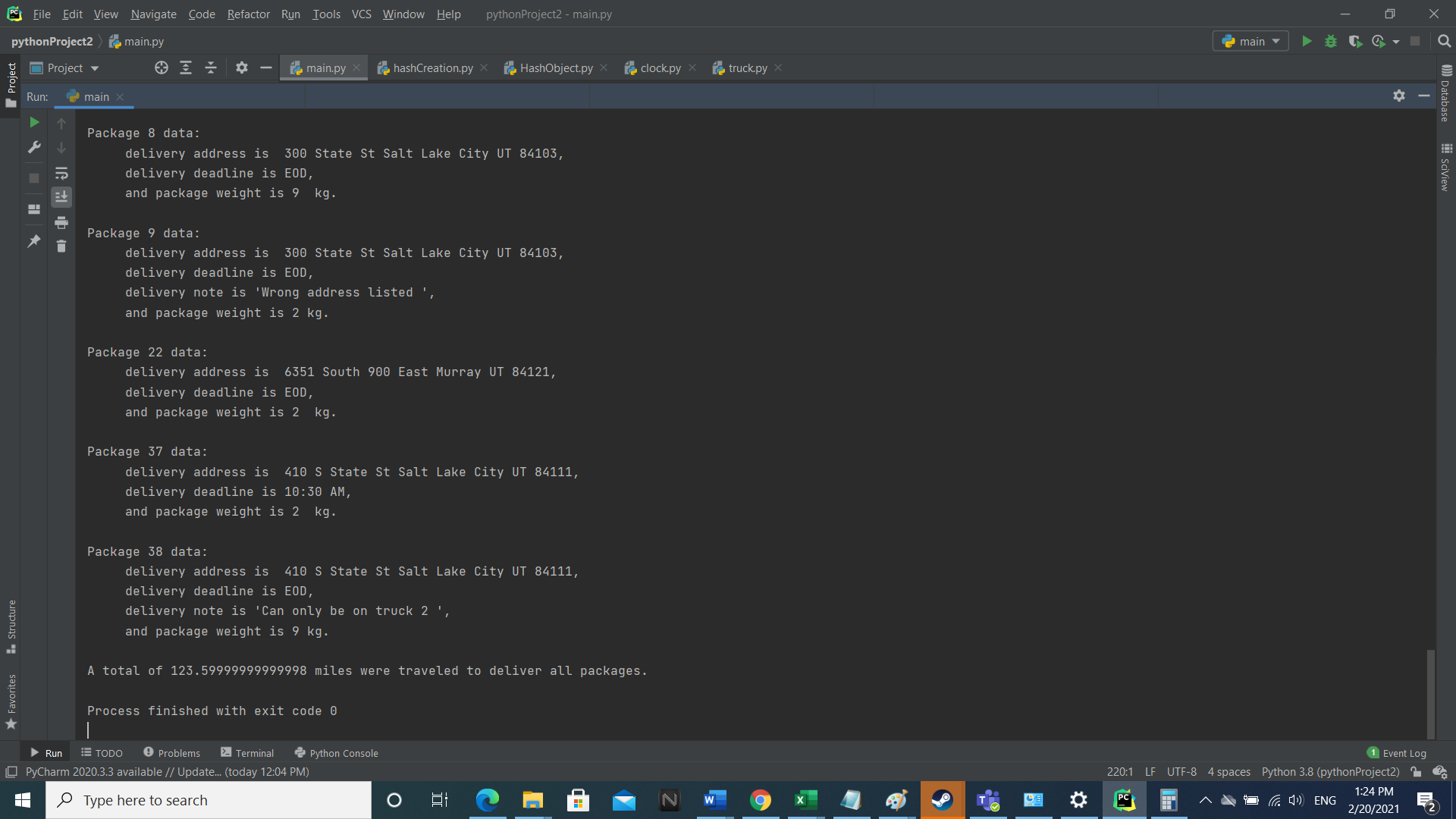
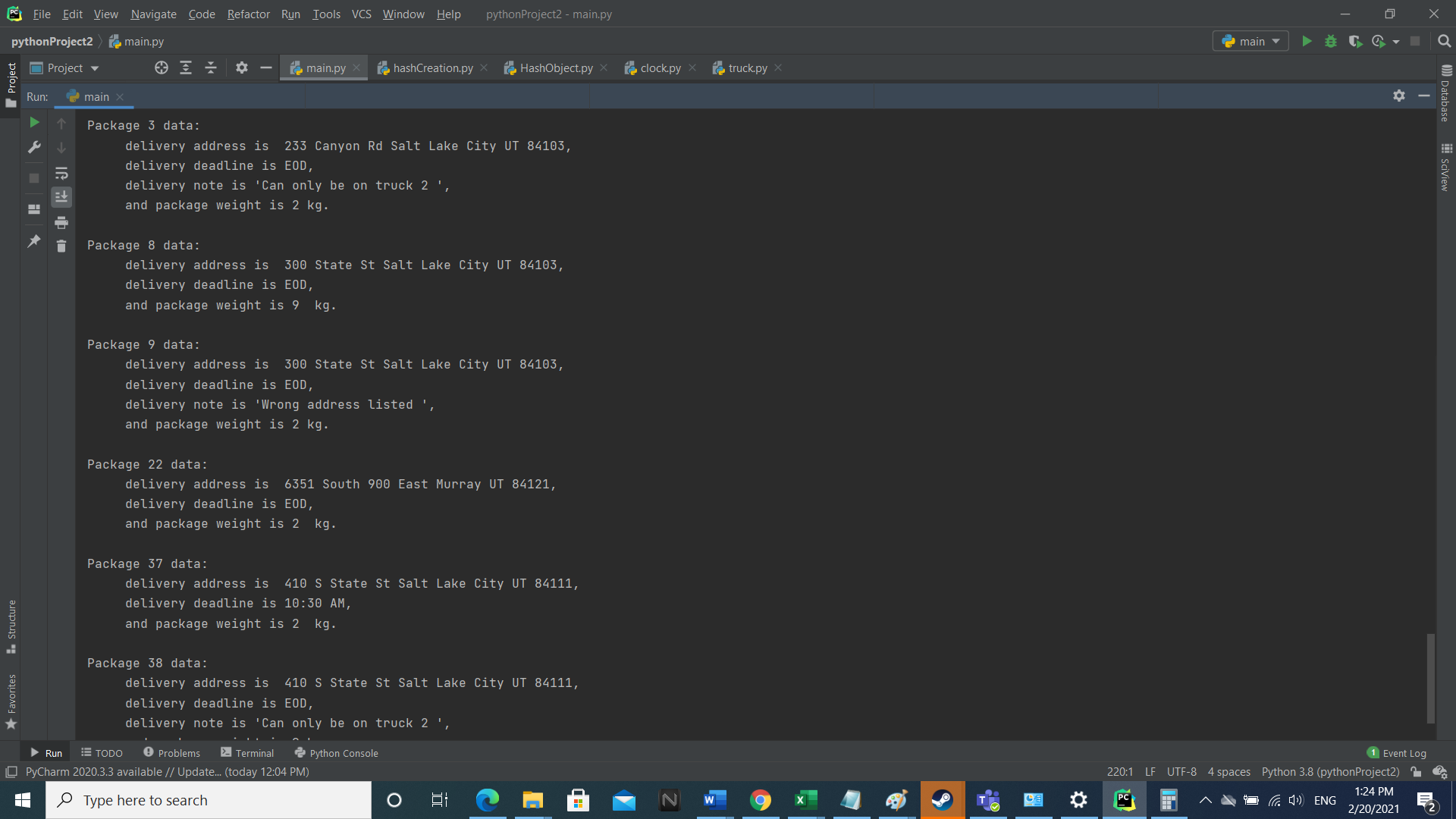












**Other algorithm options:**

Two alternative algorithms that could have been used to execute the requirements of this application include Dijkstra’s shortest path and a depths-first-search. Both Diijkstra’s algorithm and a depths-first search would result in a more efficient delivery path. However, these two algorithms would be more difficult to develop and maintain due to their complexity. Also, greater complexity of application would result in higher processing requirements for the local machine running the application, increasing initial deployment costs.

**Alteration and optimization of application:**

If I were to continue to optimize this application, I would replace the greedy algorithm with a more optimized solution, such as Dijkstra’s shortest path. Also, I would move the number of trucks into a csv file for ease of alteration at different deployment locations. For special packages, I would also alter the csv file in order to ensure the program can handle the special requirements without any of the code needing to be changed. As an example, I would put a separate column in for packages that need to be delivered together, and a separate column to indicate if there is a truck requirement for which the package must be delivered with.

**Verification information:**

All delivered packages by package id, in delivery order from first to last:

[13, 39, 15, 16, 19, 20, 21, 34, 14, 26, 25, 24, 12, 36, 27, 35, 1, 29, 7, 2, 33, 28, 40, 4, 17, 6, 18, 23, 11, 31, 32, 10, 5, 3, 30, 8, 9, 22, 37, 38]

* The **first delivery by truck one** includes the drop-off of the following packages:

[15, 16, 13, 39, 19, 20, 21, 34, 14, 26, 25, 24, 12, 36, 27, 35]

The corresponding distances are (starting with the distance from the hub to the first address and ending with the distance from the last destination to the hub):

[10.9] [0, 12.7, 0, 7.9, 0.5, 0, 5.0, 2.0, 2.8, 0, 1.7, 9.4, 3.1, 2.8, 0] [ 7.2]

The total distance for the arrival, the delivery of all packages, and the return to the hub for truck one is:

3.4+ 47.9 + 7.2 = 58.5

* The **first delivery by truck** two includes the drop-off of the following packages:

[30, 1, 29, 7, 2, 33, 28, 40, 4, 17, 6, 18, 23, 11, 31, 32]

The corresponding distances are:

[7.6] [4.5, 2.8, 0, 1.6, 0, 1.1, 1.0, 0, 2.2, 1.3, 3.9, 0.6, 0.4, 5.4, 0][ 3.7]

Total distance travelled by truck two on first trip: 7.6 + 24.799999999999997 + 3.7 = 36.099999999999997

* The **last delivery by truck one** includes the drop-off of the following packages:

[10, 5, 3, 8, 9, 22]

Starting with the distance from the hub to the first address, the corresponding distances are:

[5] [1.8, 1.0, 0.6, 0, 14.1]

Total distance travelled by truck one on second trip: 5 + 17.5 = 22.5

* The **last delivery by truck two** includes the drop-off of the following packages:

[37, 38]

Starting with the distance from the hub to the first address, the corresponding distances are:

[6.4][0.0, 0.0]

Total distance travelled for the final trip: 6.4

Summing up the distances from the four individual trips obverse that:

58.5 + 36.099999999999997 + 22.5 + 6.5 = 123.499999999999997

The distance above, in miles, matches the output of the application.

In regards to packages with special requirements or time constraints, please reference the status snapshots output by the program after a time has been entered (or see section titled screenshots).