**CS 2302 Data Structures**

**Fall 2019**

**Lab Report #3**

Due: October 8, 2019

Professor: Olac Fuentes

TA: Anindita Nath

**Introduction**

In this lab, various operations on linked lists were performed. We implemented all of these functions in two different class. One class was that of a sorted list, and the other was an unsorted list. Operations such as insertion, deletion and merging were done on the different linked lists, along with many other operations. These operations were done in order to compare Big-O running times of a sorted list versus a unsorted list. The main purpose of this lab was to compare these running times and to observe the efficiency of different algorithms on different types of linked lists.

**Proposed Solution Design and Implementation**

**Sorted List Implementation-** For the sorted list functions, I first generated a random list, then sorted it. I then implemented every function. For most functions I simply iterated through the entire linked list and made the necessary changes to the list. I used the same methods as used in class for the print, append, and append to list operations. I then created an Insert method by inserting data into a list while having the list remain sorted. The delete method was implemented similarly, except it removed an element of the linked list. Since the list is assumed to be sorted, the operations to find the maximum and minimum simply returned the head and tail of the list. For the operation to check if the list has duplicates, I checked if the data from the current node is equal to the data of the next node and returned true of false. For the select method to find the kth smallest element, I wrote a function that iterates to the kth element and returns the data of that element. The last function was a function that cleared the entire list, which was done by using the delete function on every element in the linked list.

**Normal List Implementation-** For the functions implemented with the normal list, a lot of the time the list did not need to be sorted. The same solutions from the sorted list were used for these functions with the exception of a few of the functions. The insert, delete, clear, index of, and merge were implemented the same as the sorted list functions. However, when it came to finding the minimum and maximum functions, I compared every element of the normal list to see which one was the minimum, and I did the same operation to find the maximum. For the function that returned the kth element, I first sorted the list, then iterated up to k and returned the kth element. For the duplicates function, I also first sorted the list then found duplicates in the same way as in the sorted list implementation.

After creating every function for every operation. I set up a menu in the console that prompts the user with a menu. The menu contains different operations for all the different operations on the linked list that can be performed. Whatever the user chooses will update the linked list. The user will be shown both a sorted list and normal list and the time that both lists took to operate on for the specific function. This allows comparing of Big-O running times to be seen and compared clearly.

**Experimental Results**

The big-O running times vary depending on the type of list. For sorted lists, a lot of the time the Big-O is constant since the list is already sorted and there is no need to search through the entire list. The normal list on the other hand, has to be searched or sometimes even sorted before having any operations performed on it. This causes a larger Big-O running time.

|  |  |  |
| --- | --- | --- |
| **Function** | **SortedList** | **List** |
| **Print()** | O(n) | O(n) |
| **Insert(i)** | O(n) | O(n) |
| **Delete(i)** | O(n) | O(n) |
| **Merge(M)** | O(1) | O(n) |
| **IndexOf(i)** | O(1) | O(n) |
| **Clear(i)** | O(n) | O(n) |
| **Min()** | O(1) | O(n) |
| **Max()** | O(1) | O(n) |
| **HasDuplicates()** | O(n) | O(n) |
| **Select(k)** | O(1) | O(n) |

To test each function, I will run a few different lists and compare all the running times for each function. The same list will be used for each run, however one list will be sorted and the other unsorted. The big-O notation is relatively faster for the sorted list, since no sorting has to be done. However, for some of the functions, the Big-O is the same for both types of linked lists.

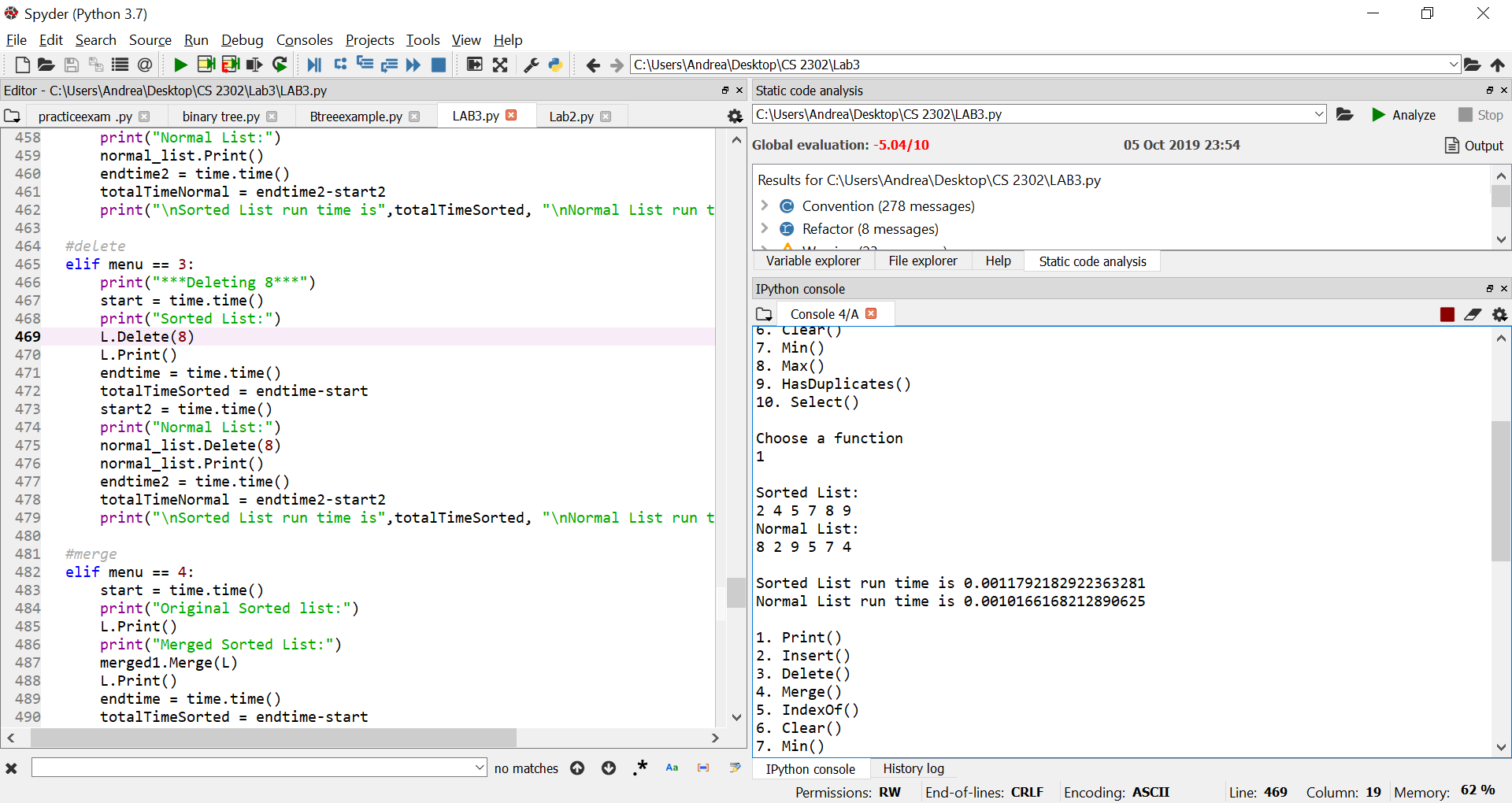
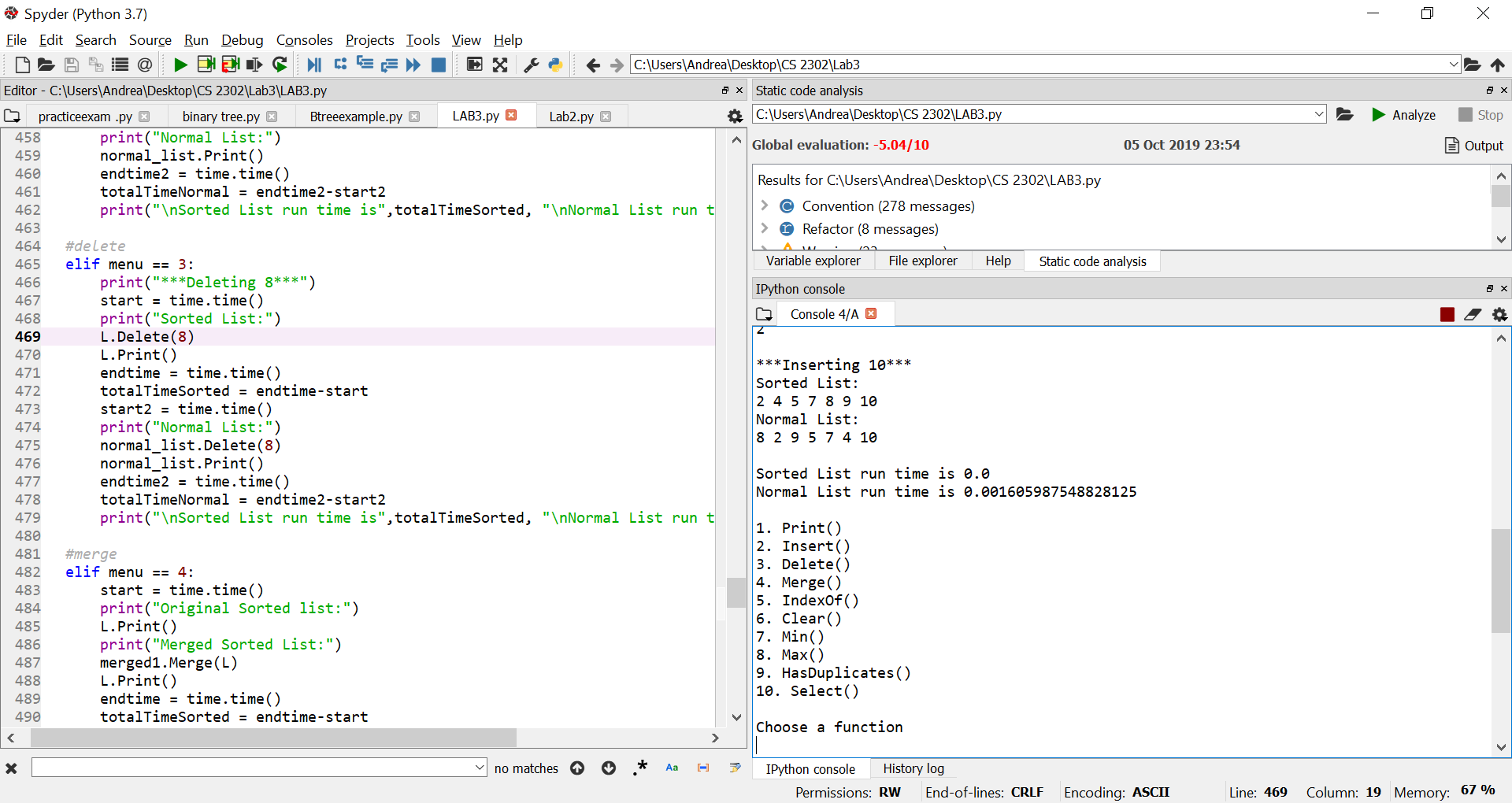
**Test #1:**

**Sorted List: [2,4,5,7,8,9]**

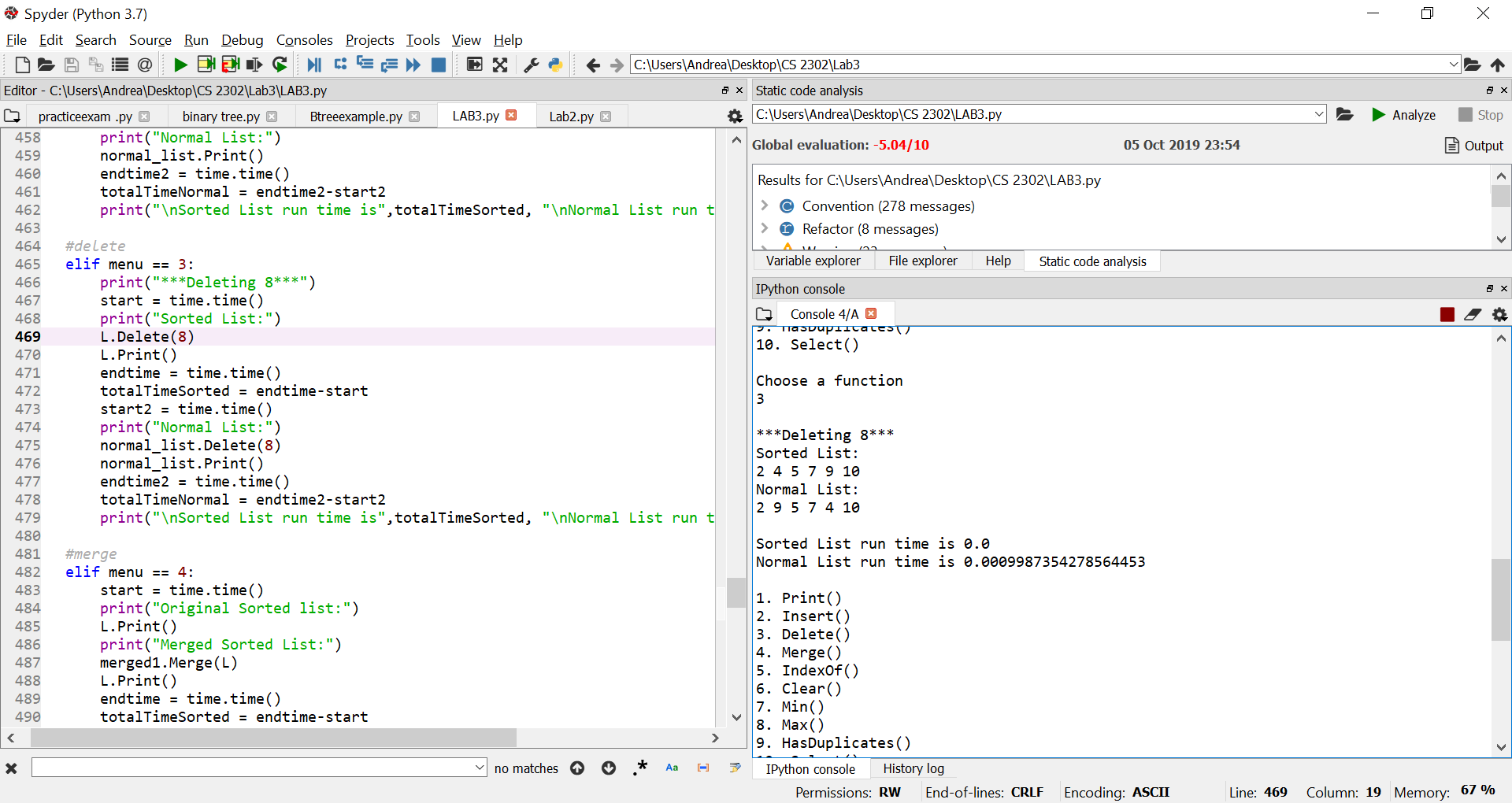
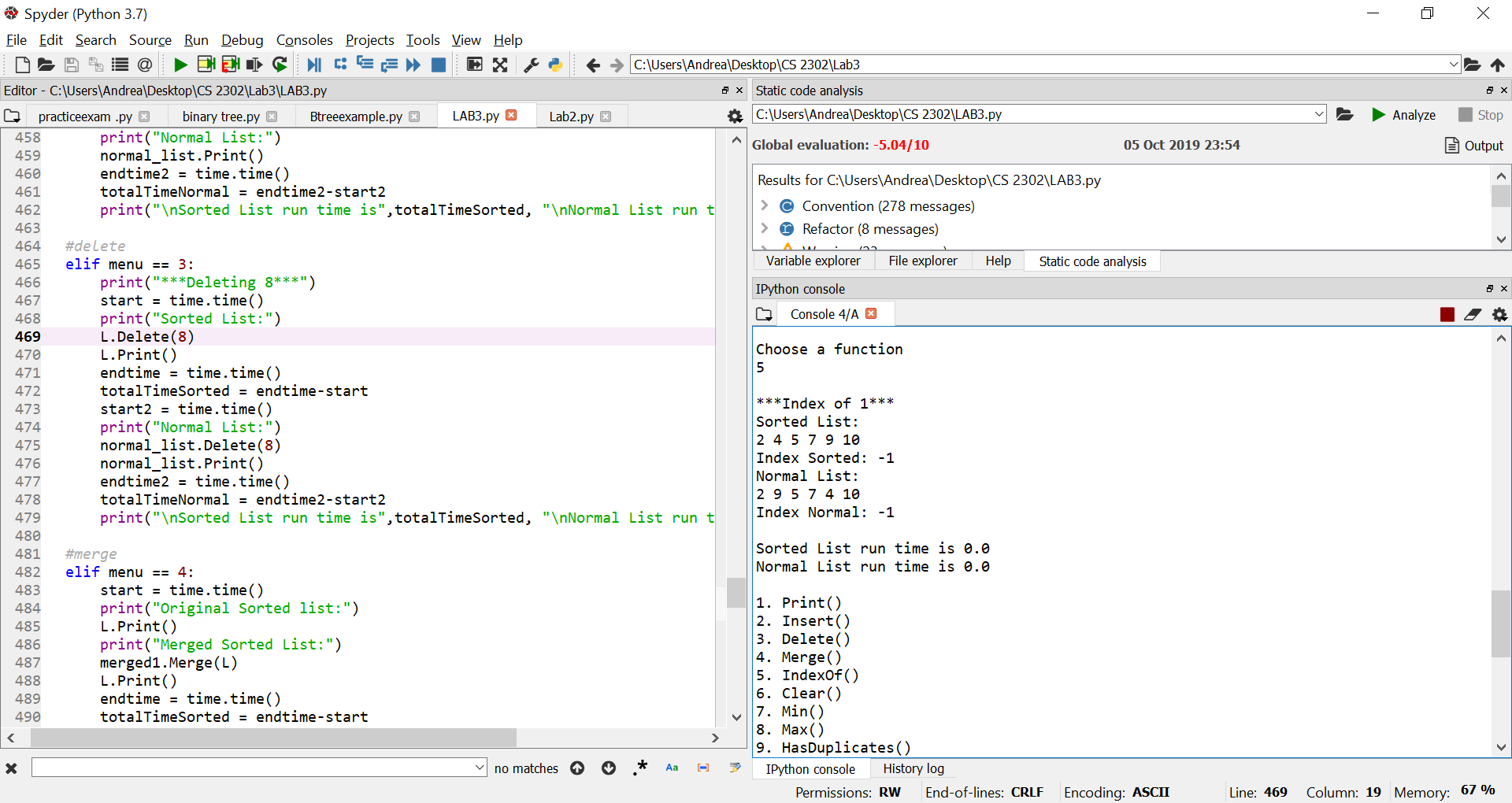
**List: [8,2,9,5,7,4]**

The first test was done with a list of length six. The sorted list appeared to have the quickest run time in every function, with the exception of a few functions. For printing, both lists took about the same time, which is true since they have the same Big-O running time of O(n). Each element is visited in the list. The normal list took longer in the insert and delete methods, since the list is not previously sorted, and these also have a running time of O(n). Finding the max and min took longer in the normal list since it had to go through the entire list and compare each element. The sorted list on the other hand was faster since it only returns the head or tail since the list is already sorted. The running time for max and min is O(1) in the sorted list class, while in the normal list class it is O(n) due to every element being visited. The sorted list had the faster running time for the most part.

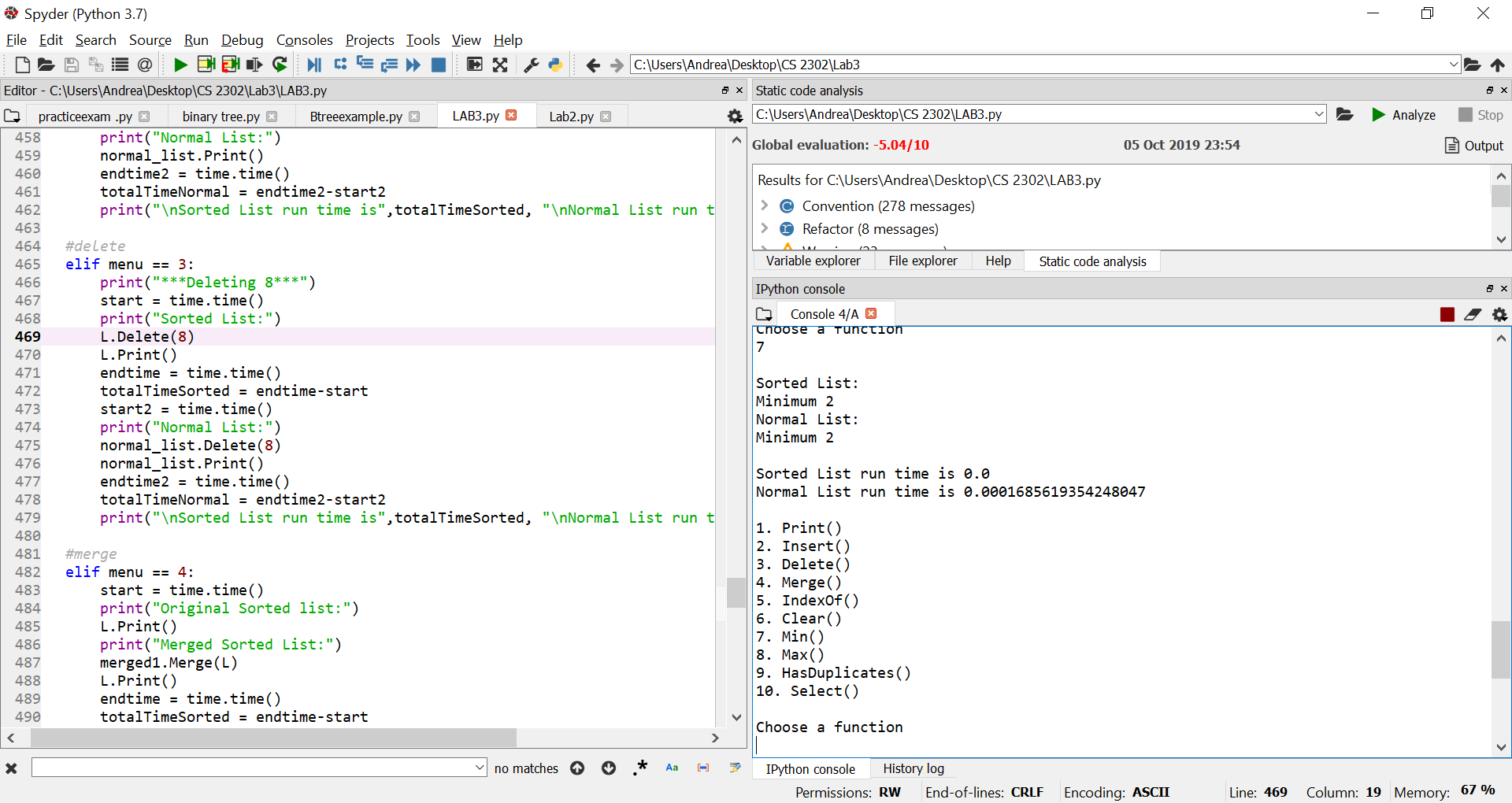
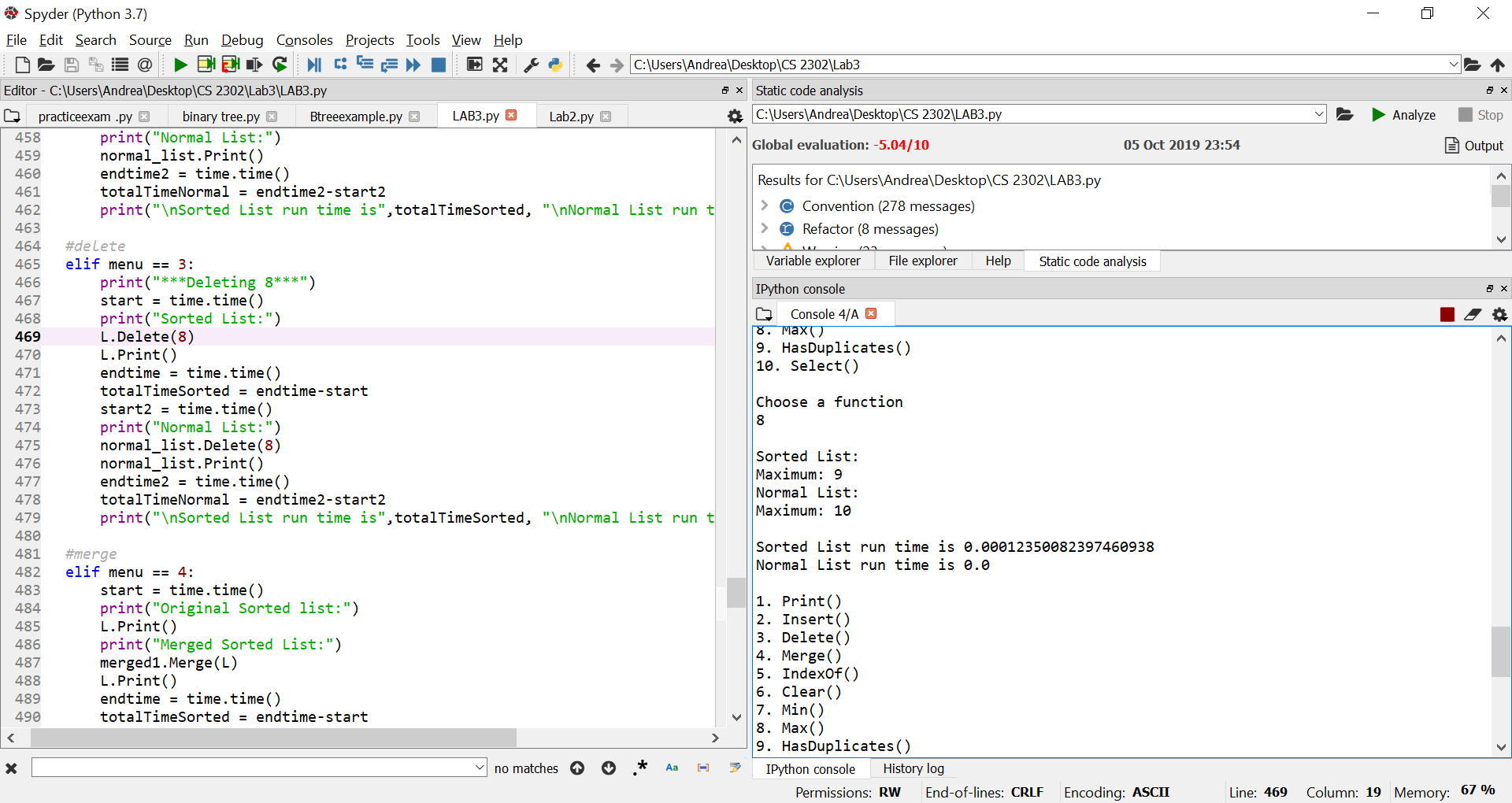
Print() Insert(i)

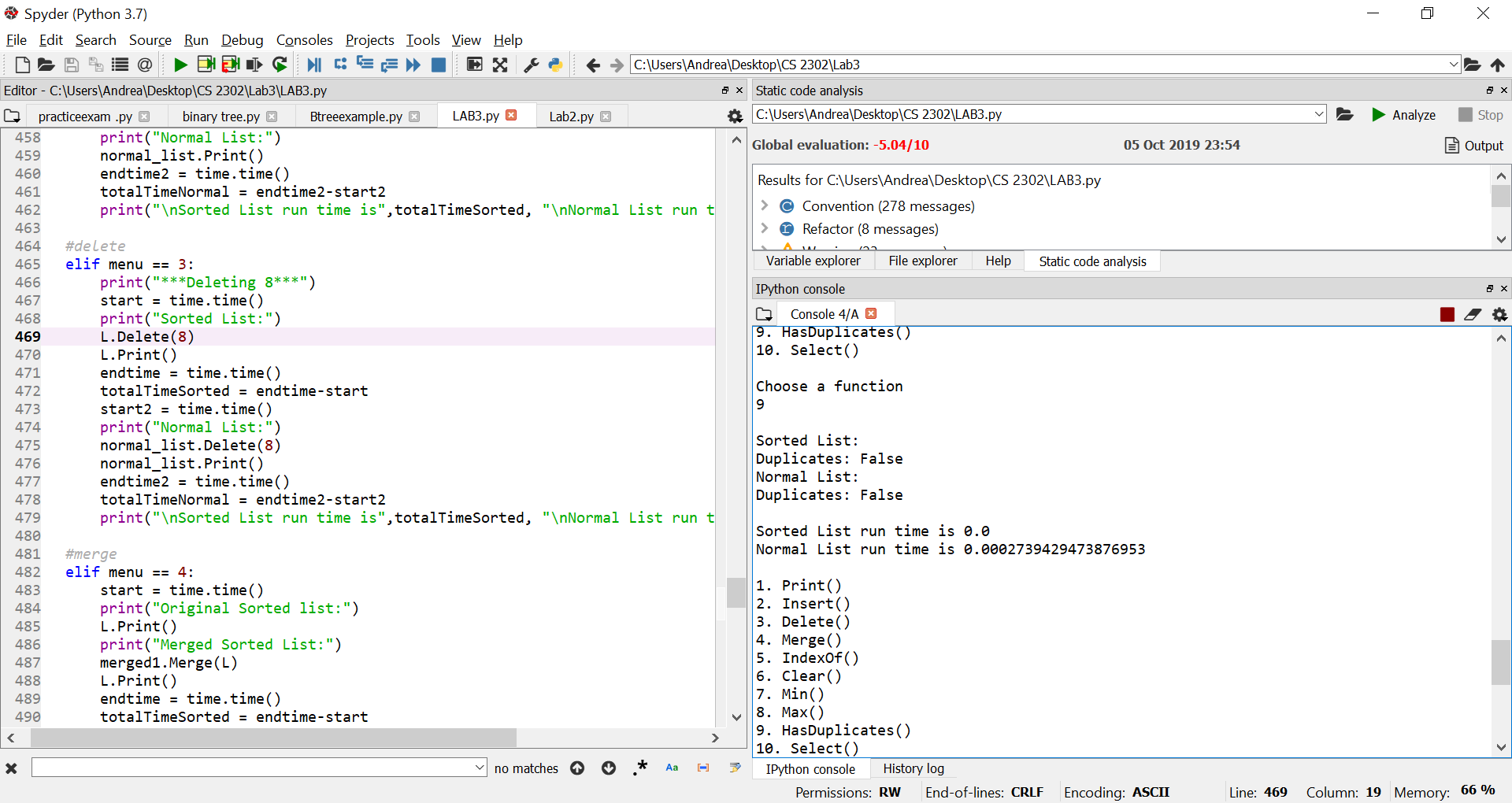
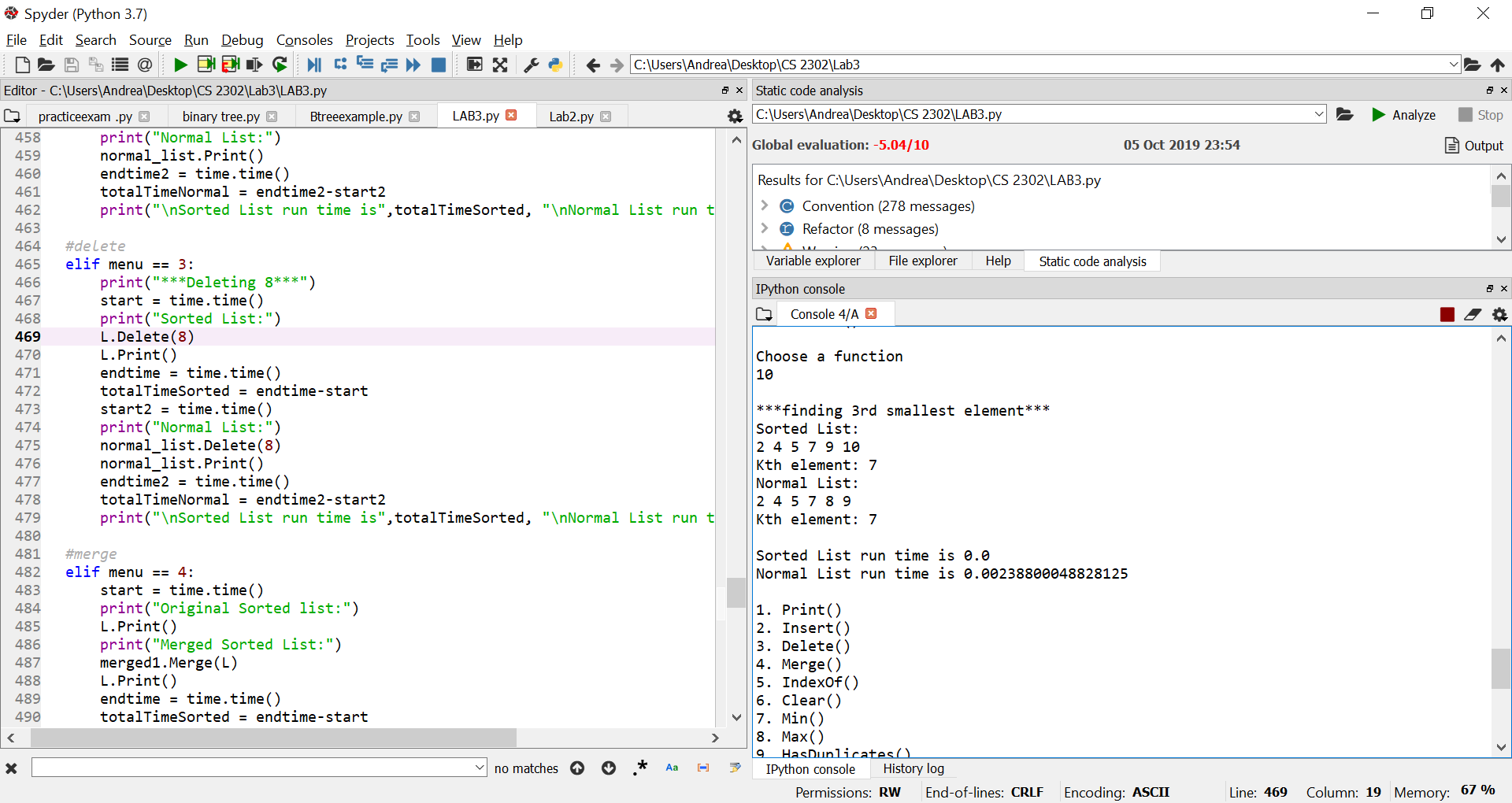
Delete(i) IndexOf(i)

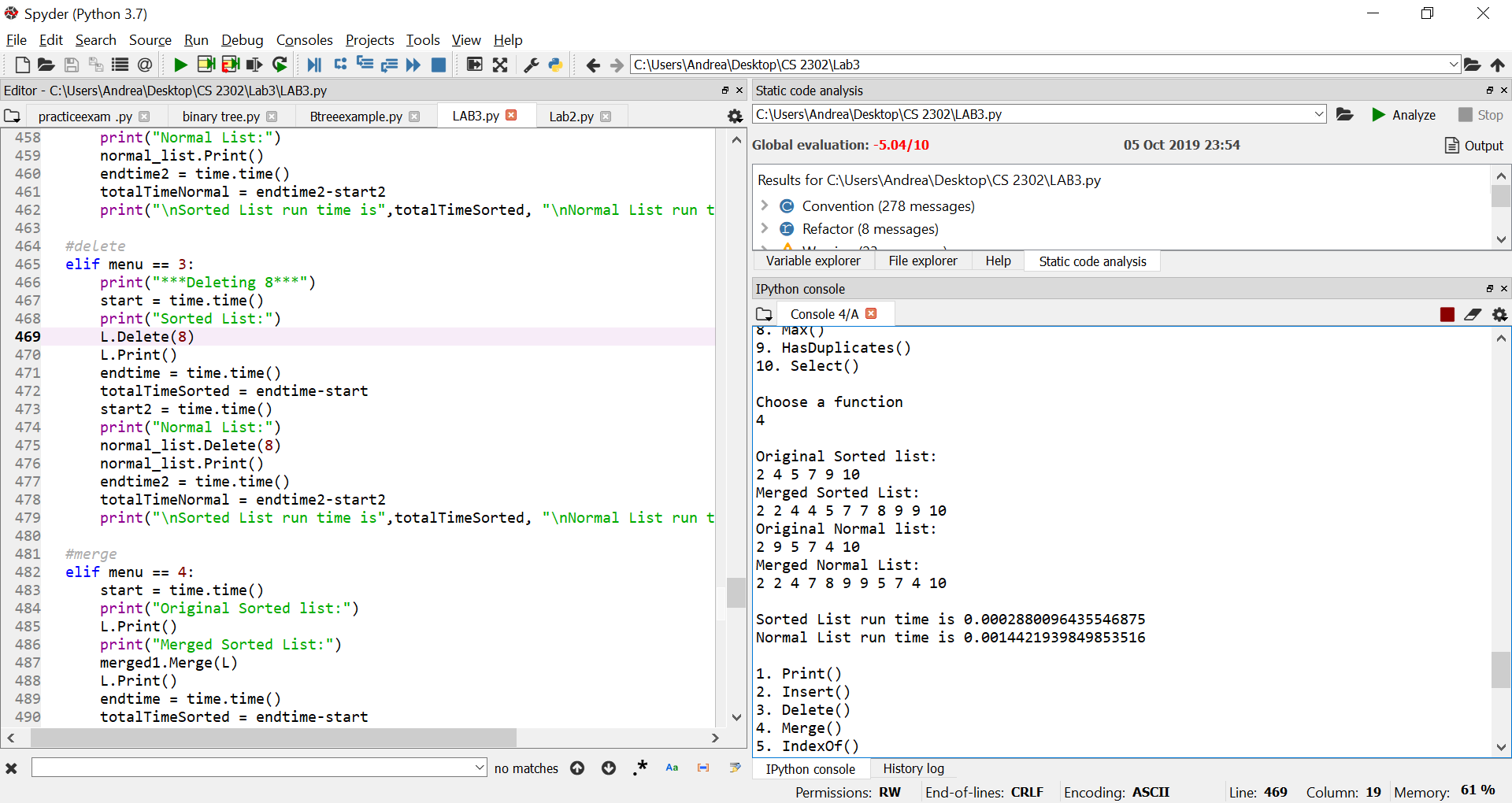
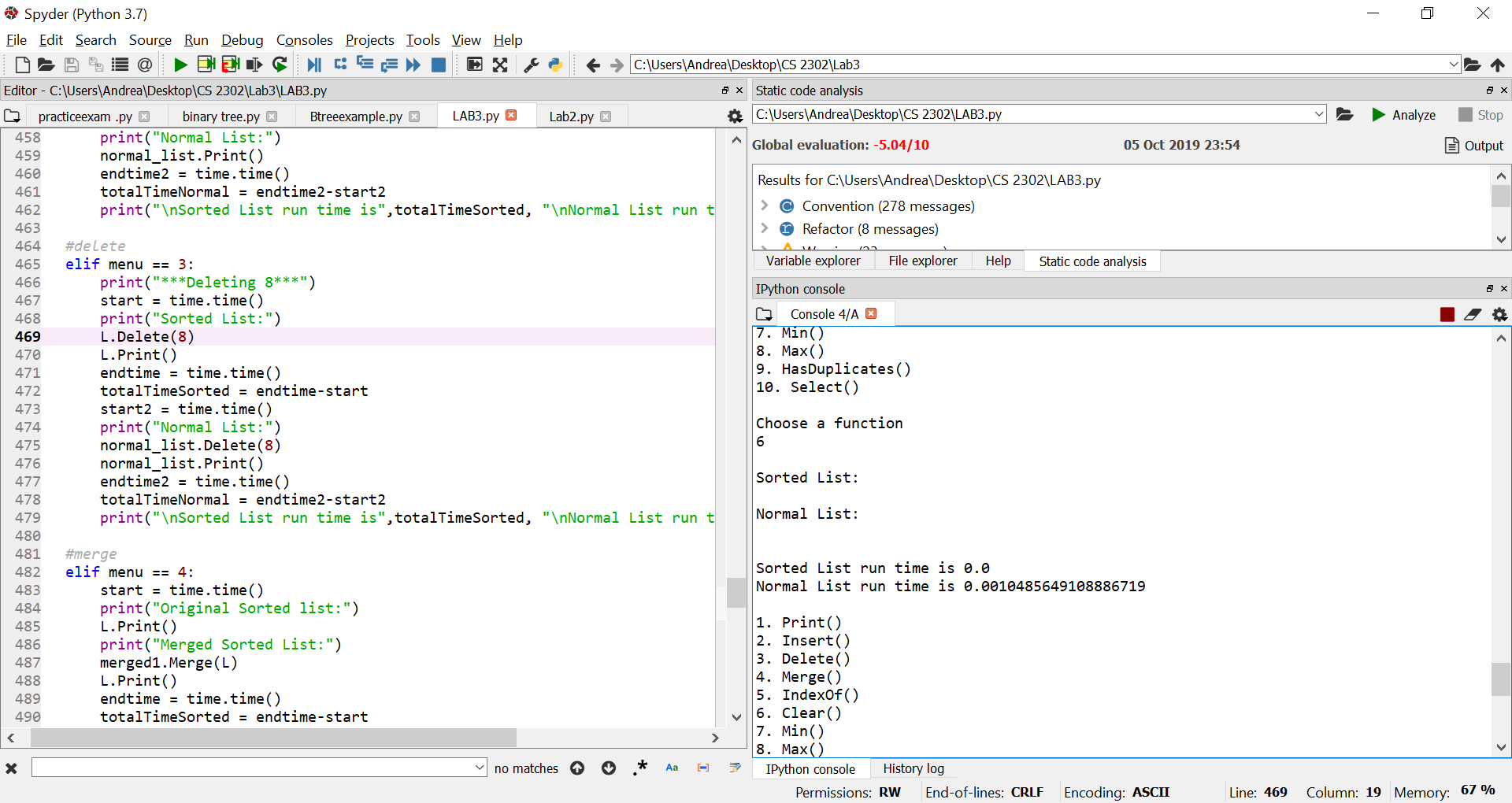
Min() Max()

HasDuplicates() Select(k)

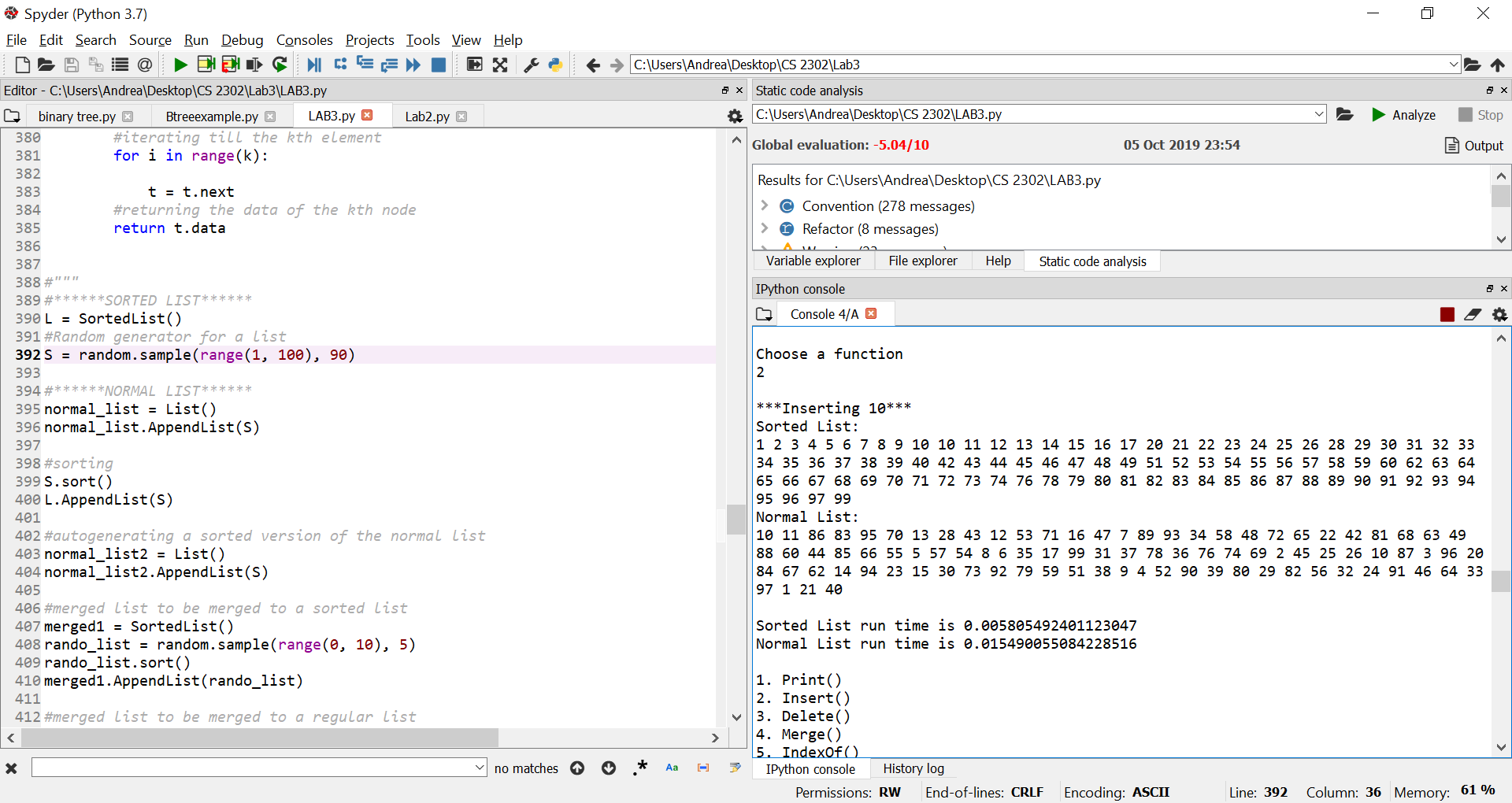
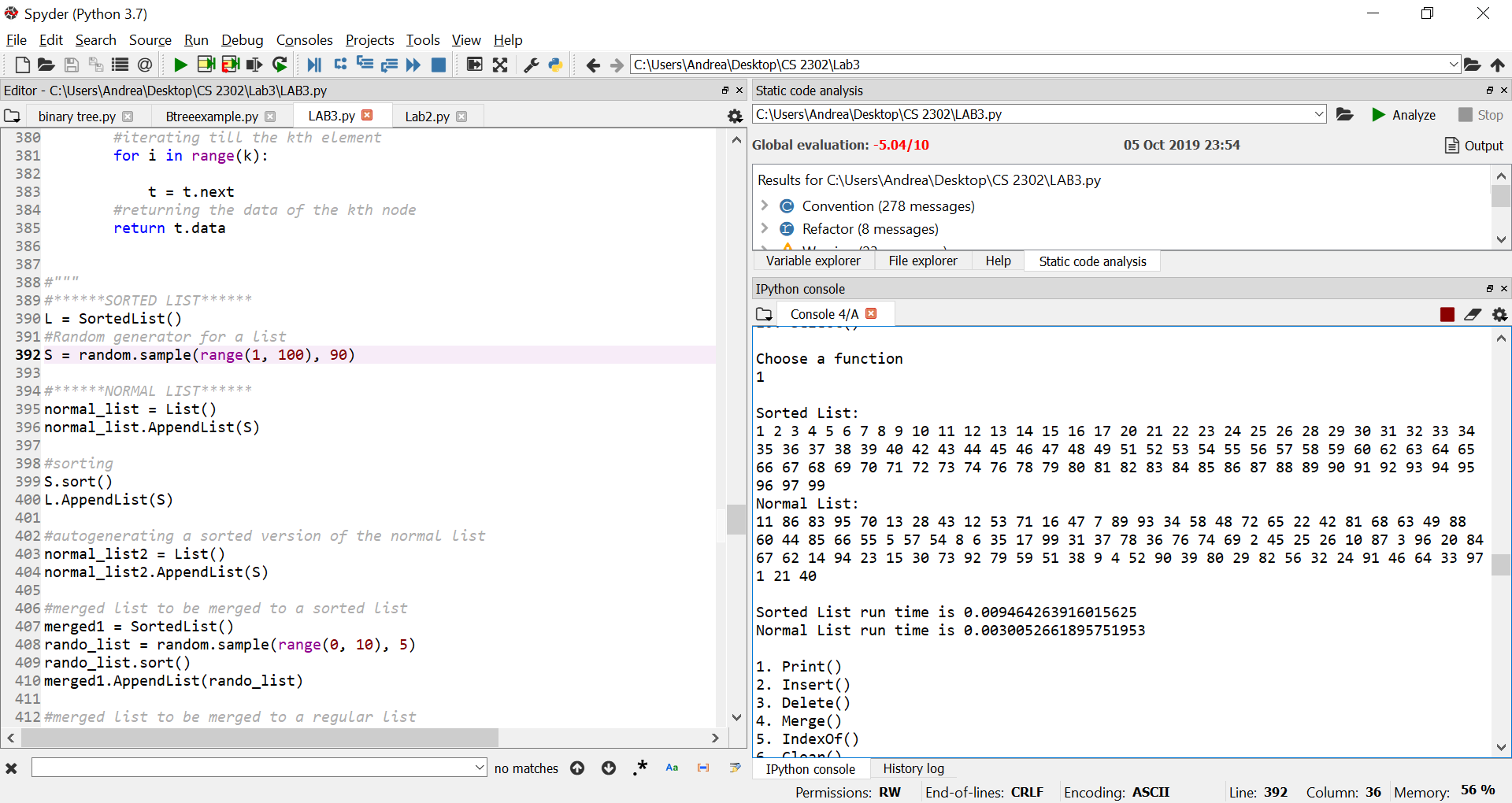
Merge(M) Clear()

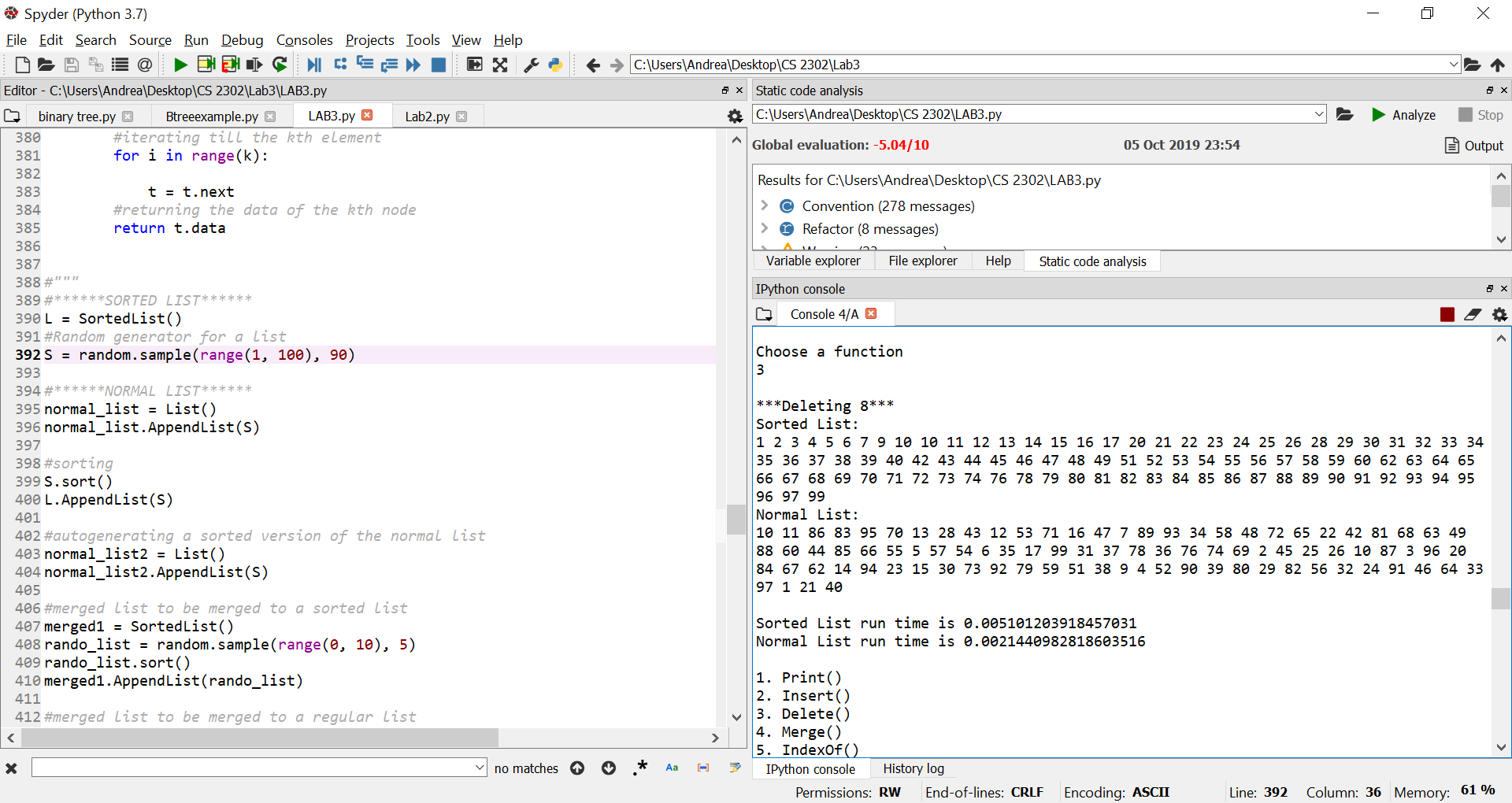
**Test case #2:**

The second test case consisted of a longer list of numbers. The length of the list is 90 and it contains numbers from 0-100. For this particular case, both types of lists seemed to have a relatively similar runtime. They both took longer since the list is of larger size, compared to the first test. The normal list did however take a little longer than the sorted list in some cases due to it being unsorted. In functions such as Clear() and Merge(), the normal list took longer to operate on since the functions went through every single element with a running time of O(n). Finding the maximum and minimum values took zero seconds for the sorted list and the minimum operation took longer for the normal list which makes sense since there is a O(n) running time. Similarly as in test #1, the max and min for the sorted list have a constant running time of O(1) causing the faster running time in seconds.

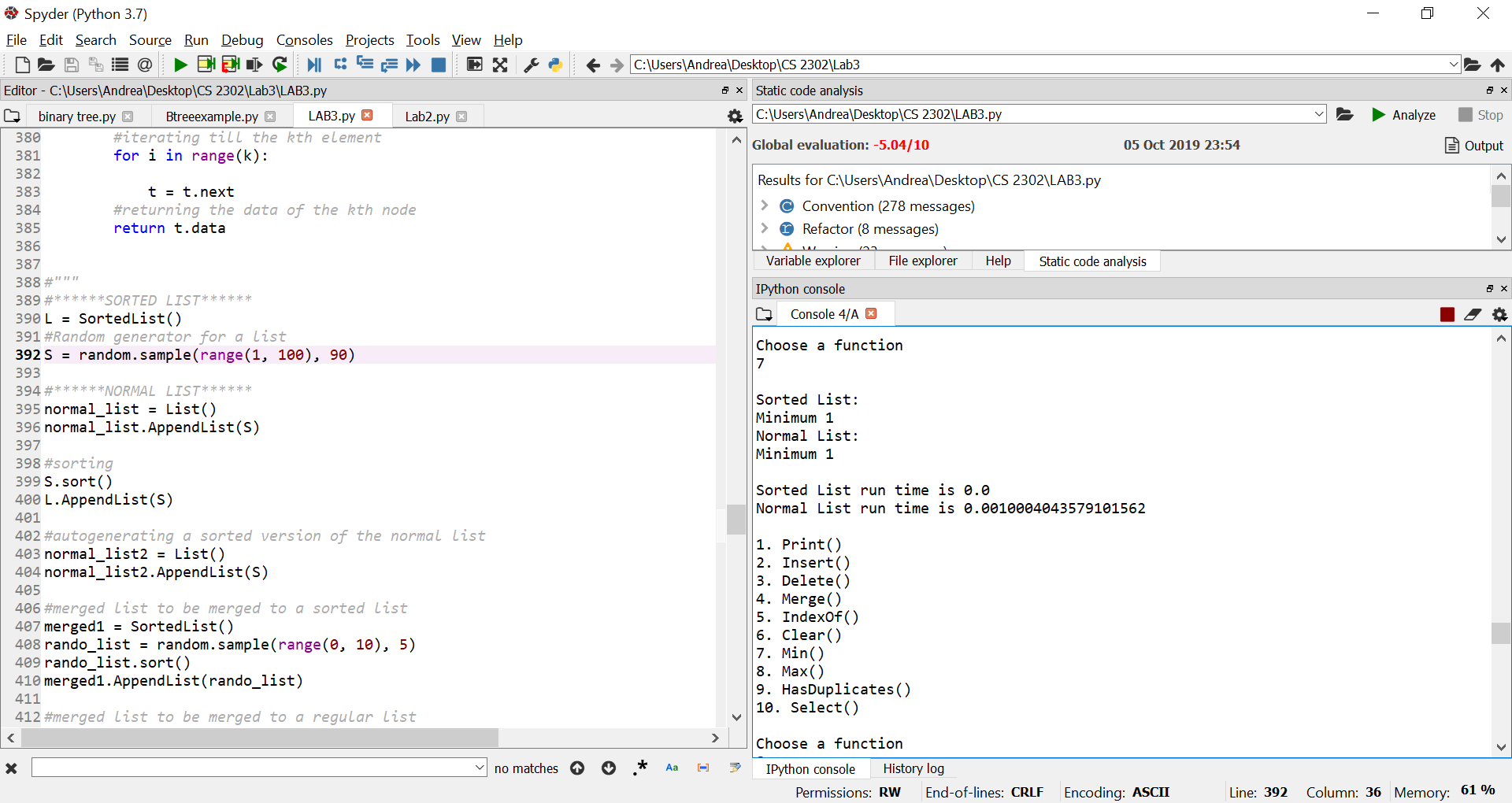
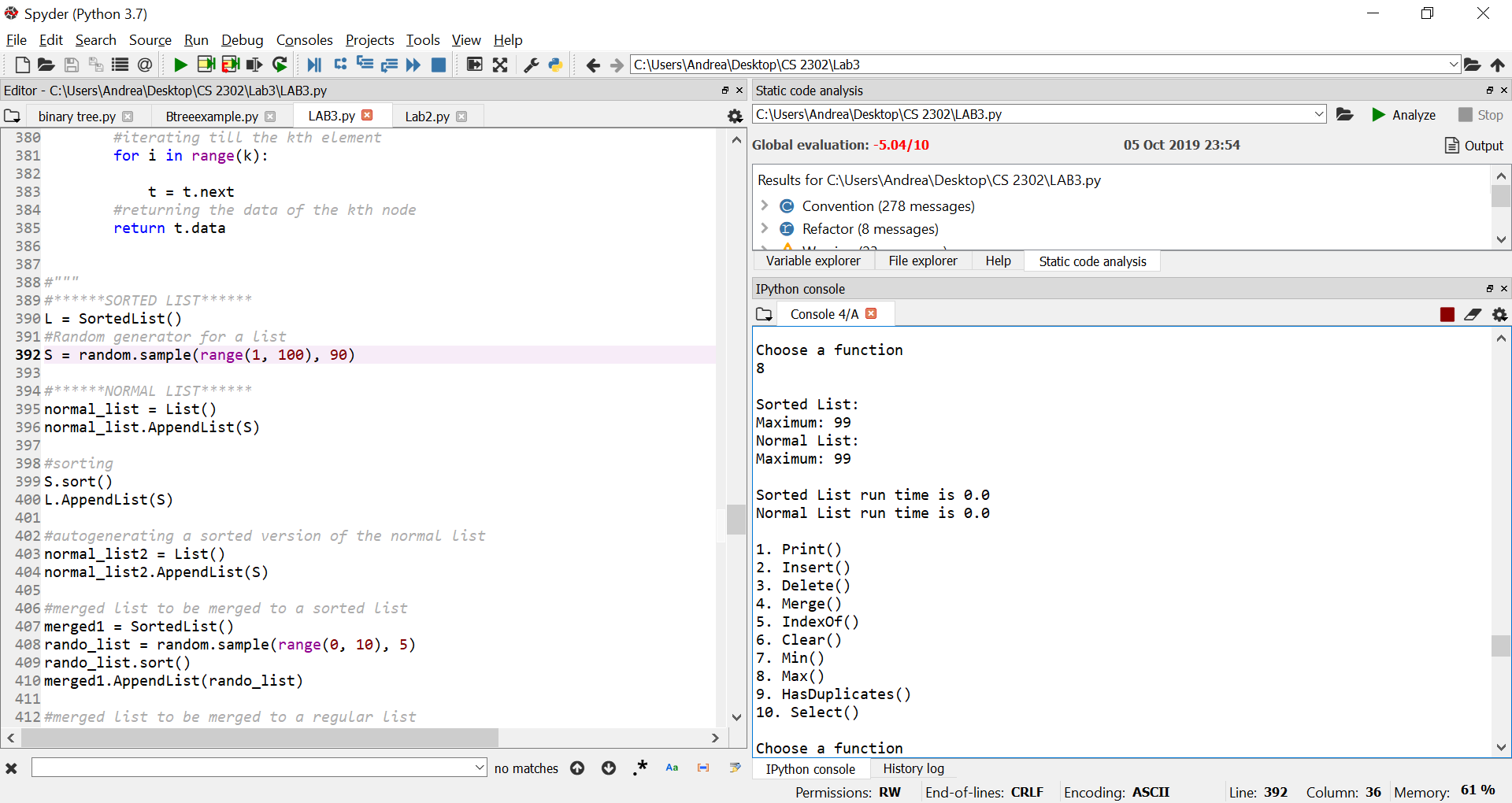
Print() Insert(i)

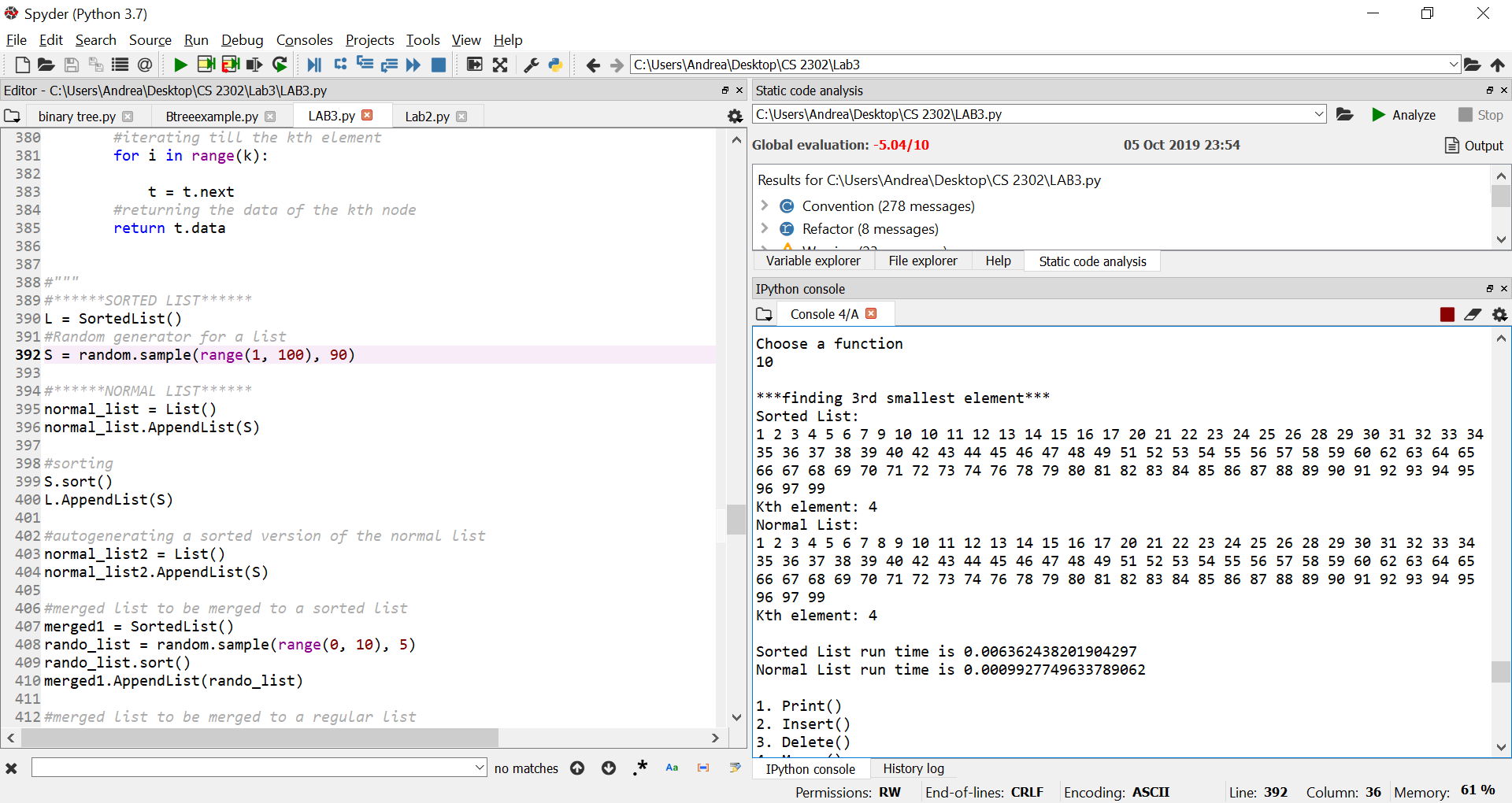
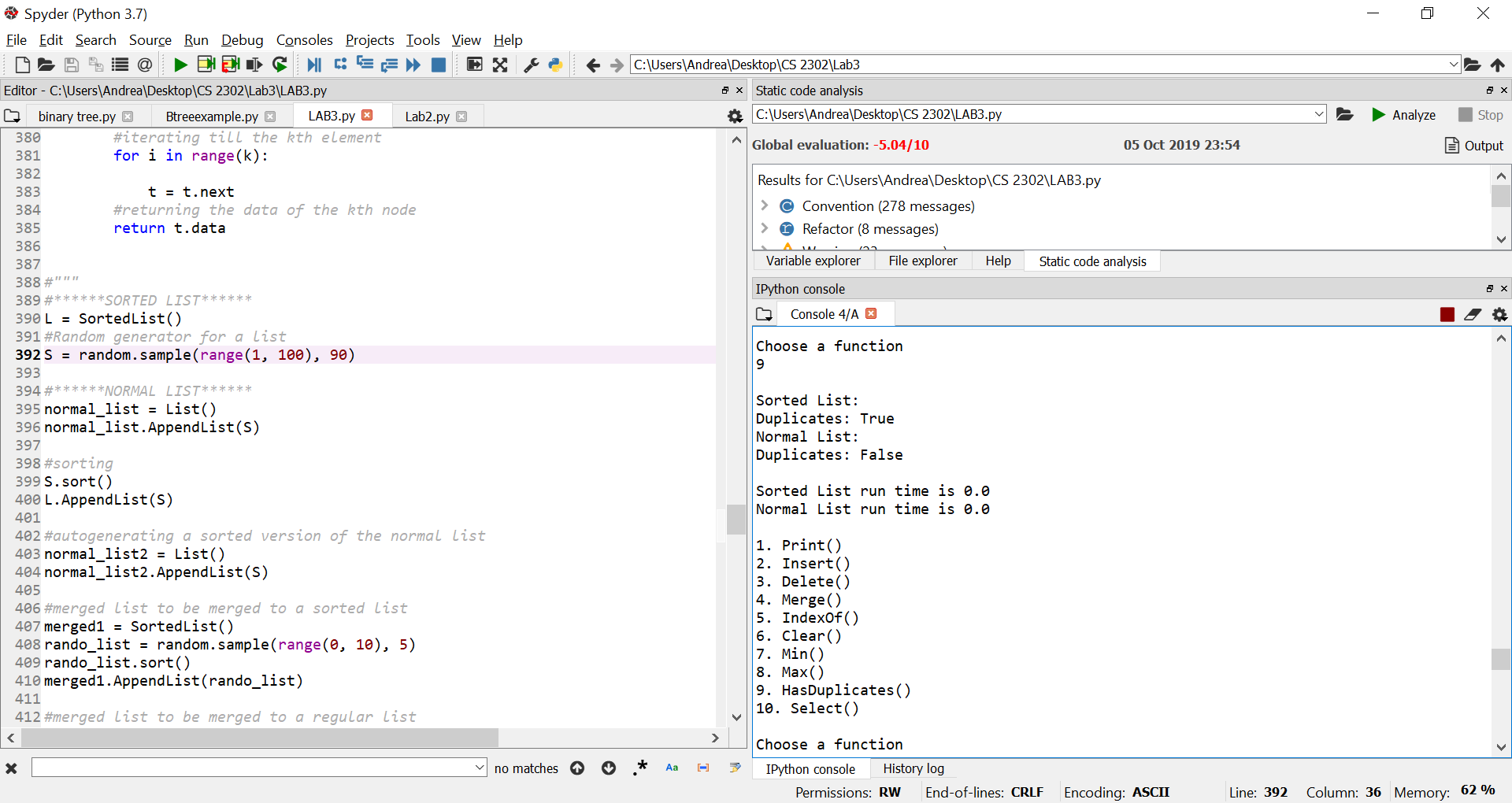
Delete(i) IndexOf(i)

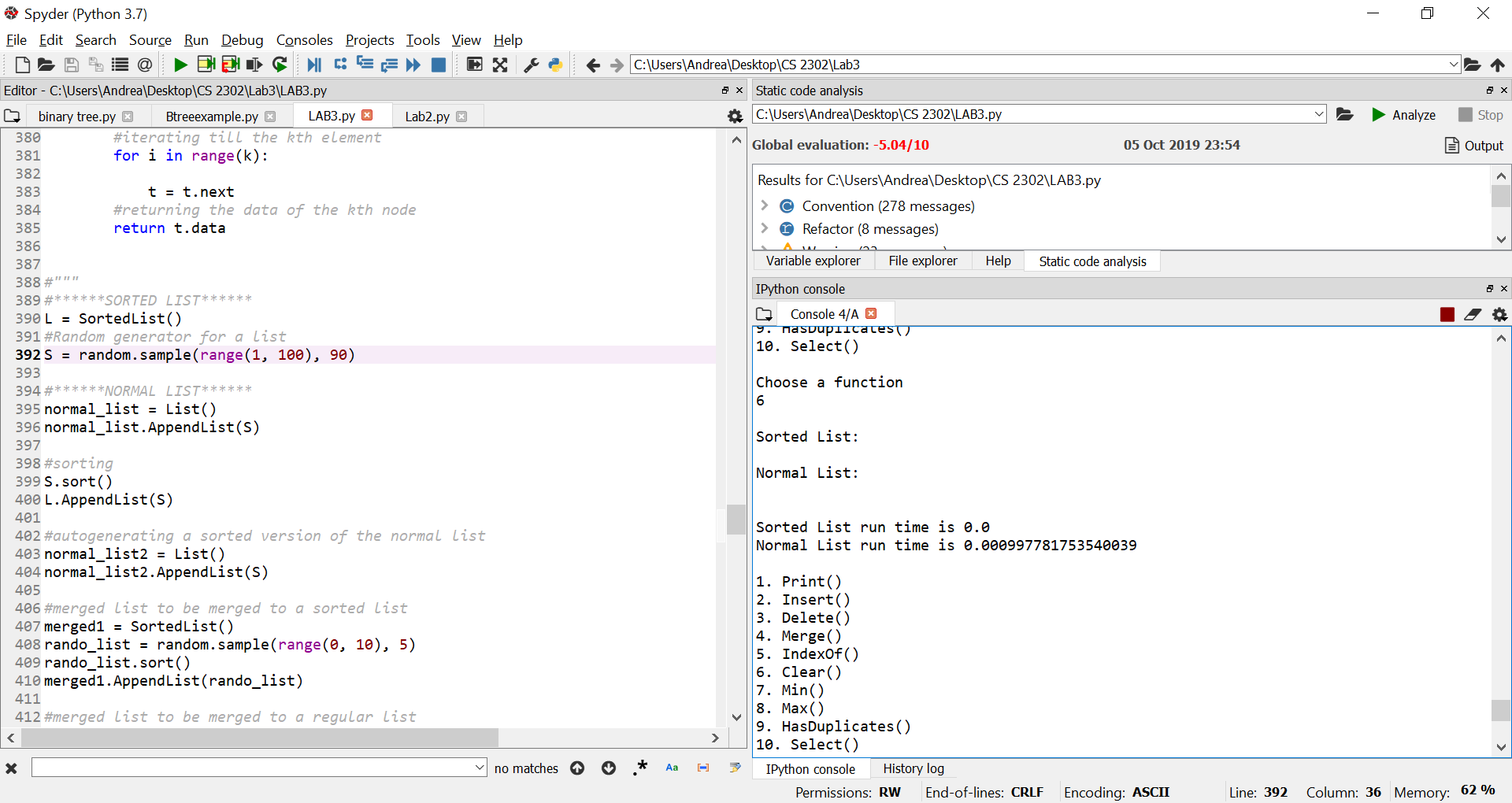
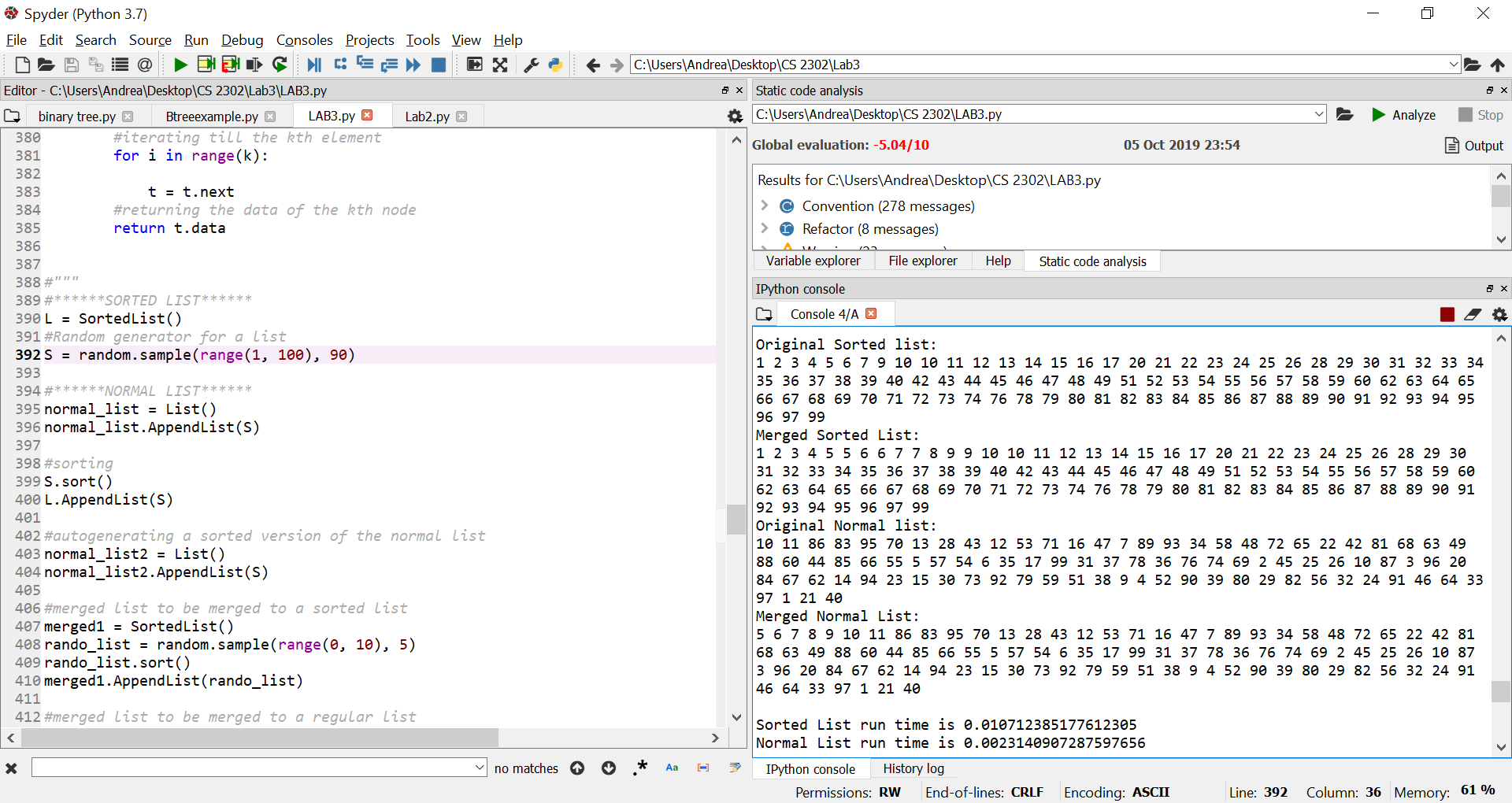
Min() Max()

HasDuplicates() Select(k)



Merge(M) Clear()



**Running times:**

**Test #1:**

|  |  |  |
| --- | --- | --- |
| **Linked List Function** | **Sorted List** | **List** |
| **Print()** | 0.001179 | 0.001016 |
| **Insert(self,i)** | 0.0 | 0.001605 |
| **Delete(self,i)** | 0.0 | 0.000998 |
| **Merge(self,M)** | 0.000288 | 0.001442 |
| **IndexOf(self,i)** | 0.0 | 0.0 |
| **Clear(self)** | 0.0 | 0.001048 |
| **Min(self)** | 0.0 | 0.000168 |
| **Max(self)** | 0.000123 | 0.0 |
| **HasDuplicates(self)** | 0.0 | 0.000273 |
| **Select(self,k)** | 0.0 | 0.002388 |

**Test #2:**

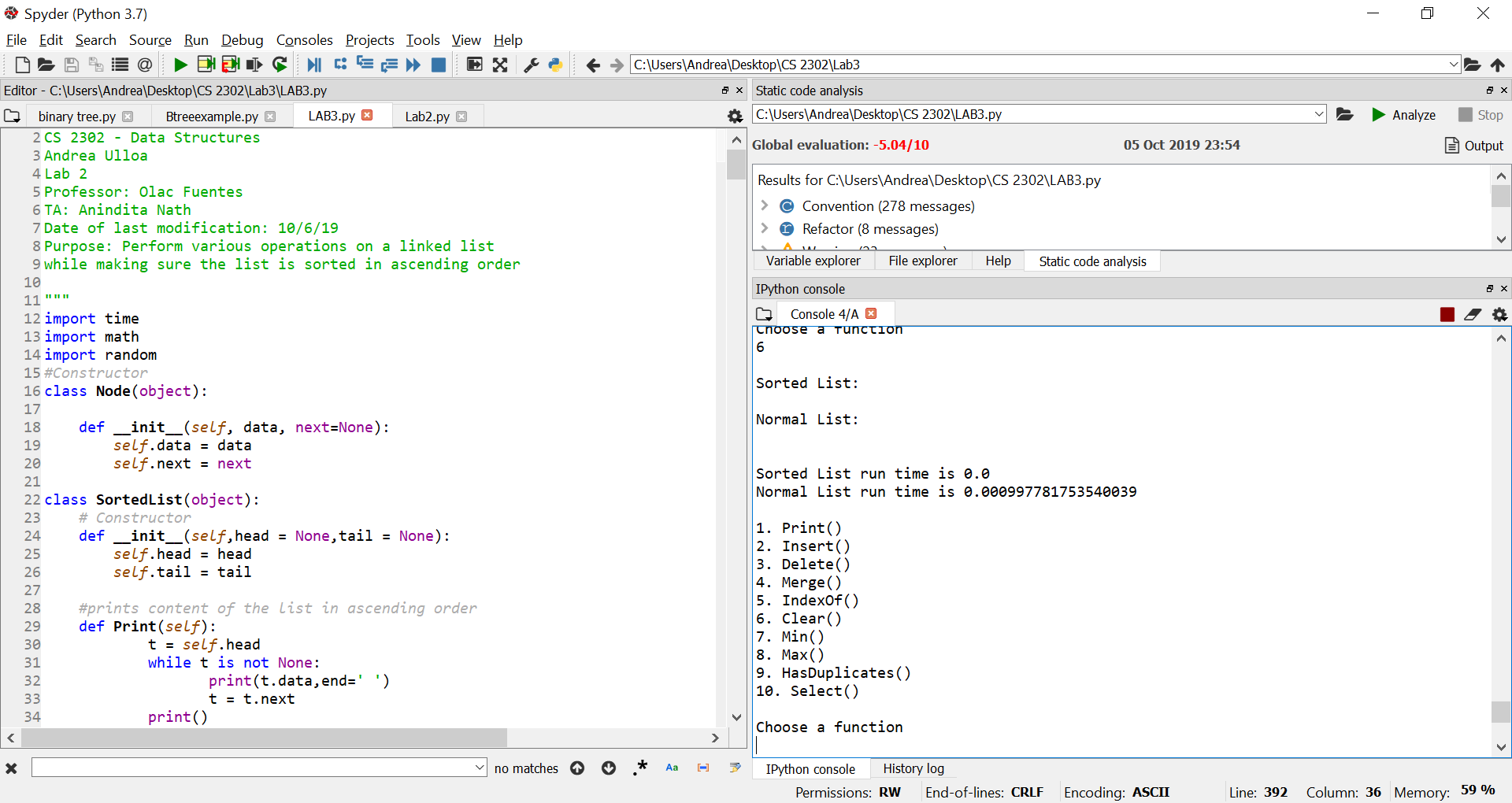
|  |  |  |
| --- | --- | --- |
| **Linked List Function** | **Sorted List** | **List** |
| **Print()** | 0.094642 | 0.0030052 |
| **Insert(self,i)** | 0.005805 | 0.0154900 |
| **Delete(self,i)** | 0.051012 | 0.0021440 |
| **Merge(self,M)** | 0.010712 | 0.0023140 |
| **IndexOf(self,i)** | 0.0020835 | 0.0020835 |
| **Clear(self)** | 0.0 | 0.0009977 |
| **Min(self)** | 0.0 | 0.0010004 |
| **Max(self)** | 0.0 | 0.0 |
| **HasDuplicates(self)** | 0.0 | 0.0 |
| **Select(self,k)** | 0.0063624 | 0.0009927 |

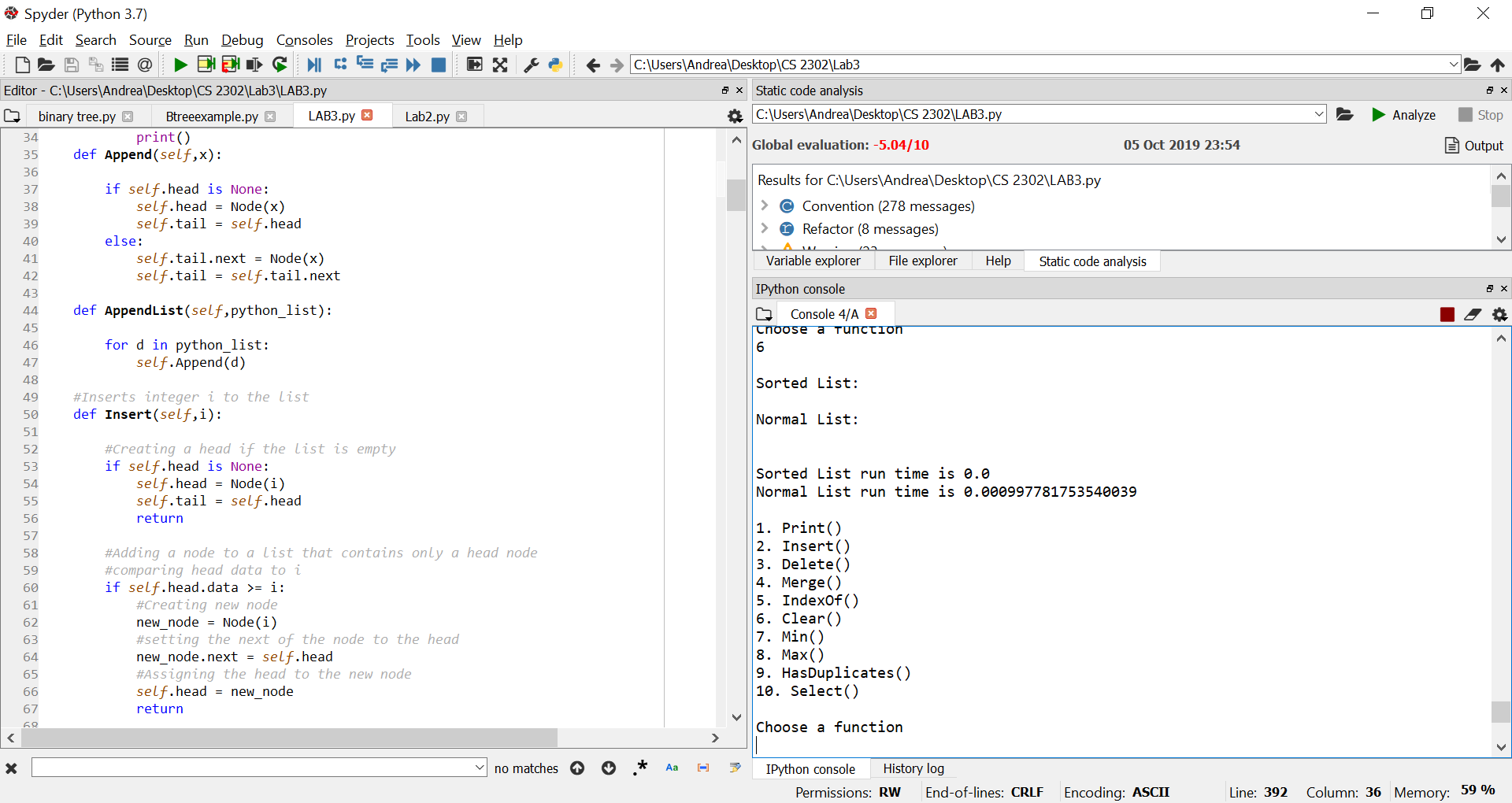
Overall, the sorted list and its functions performed quicker than the normal list. Despite a few instances such as deletions, the sorted list maintained a constant running time throughout both tests of the program. This is true because the sorted list is sorted before entering each operation which makes it easier to find certain elements since there is not always a need to iterate through the entire list.

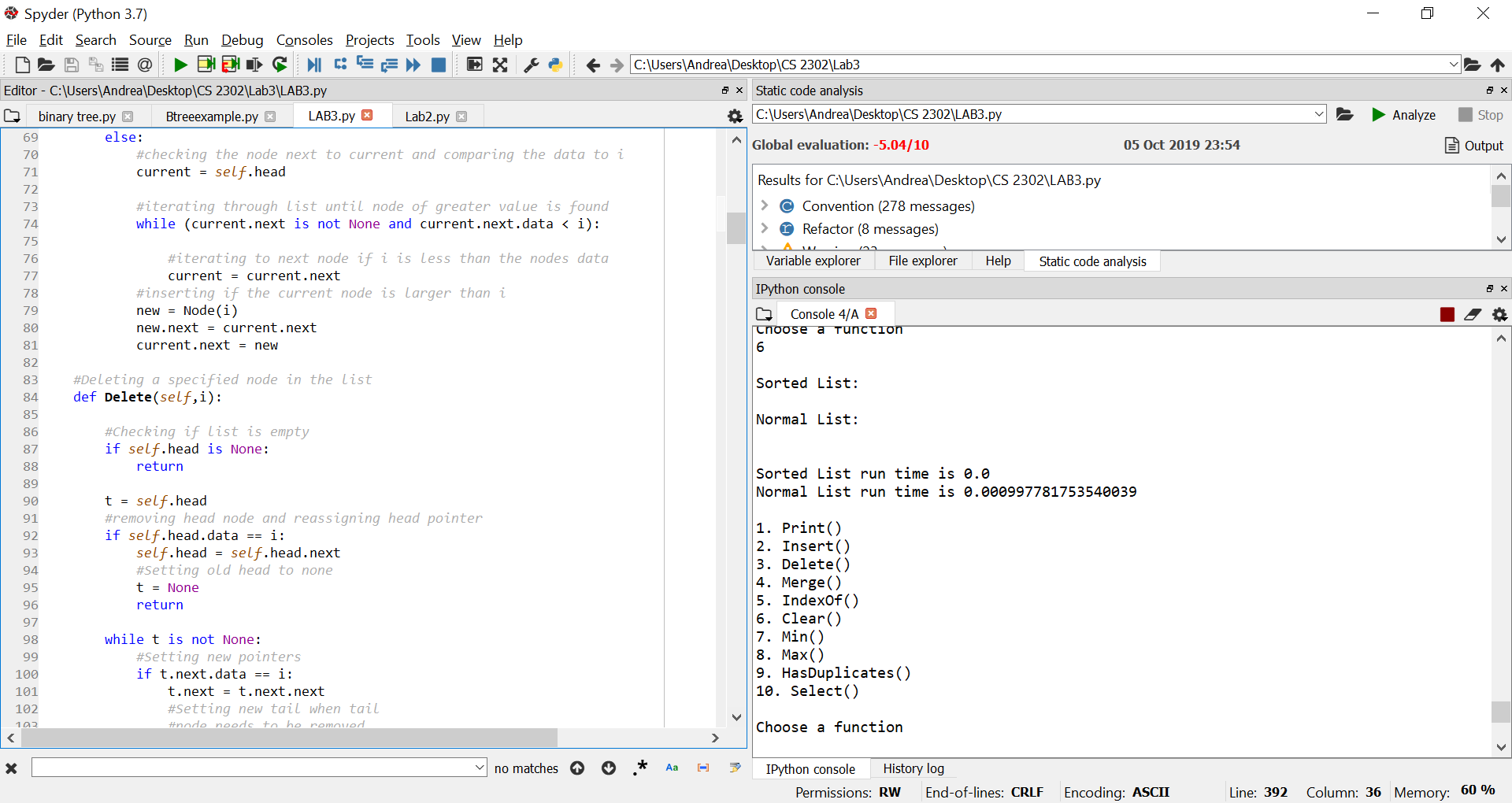
**Conclusion**

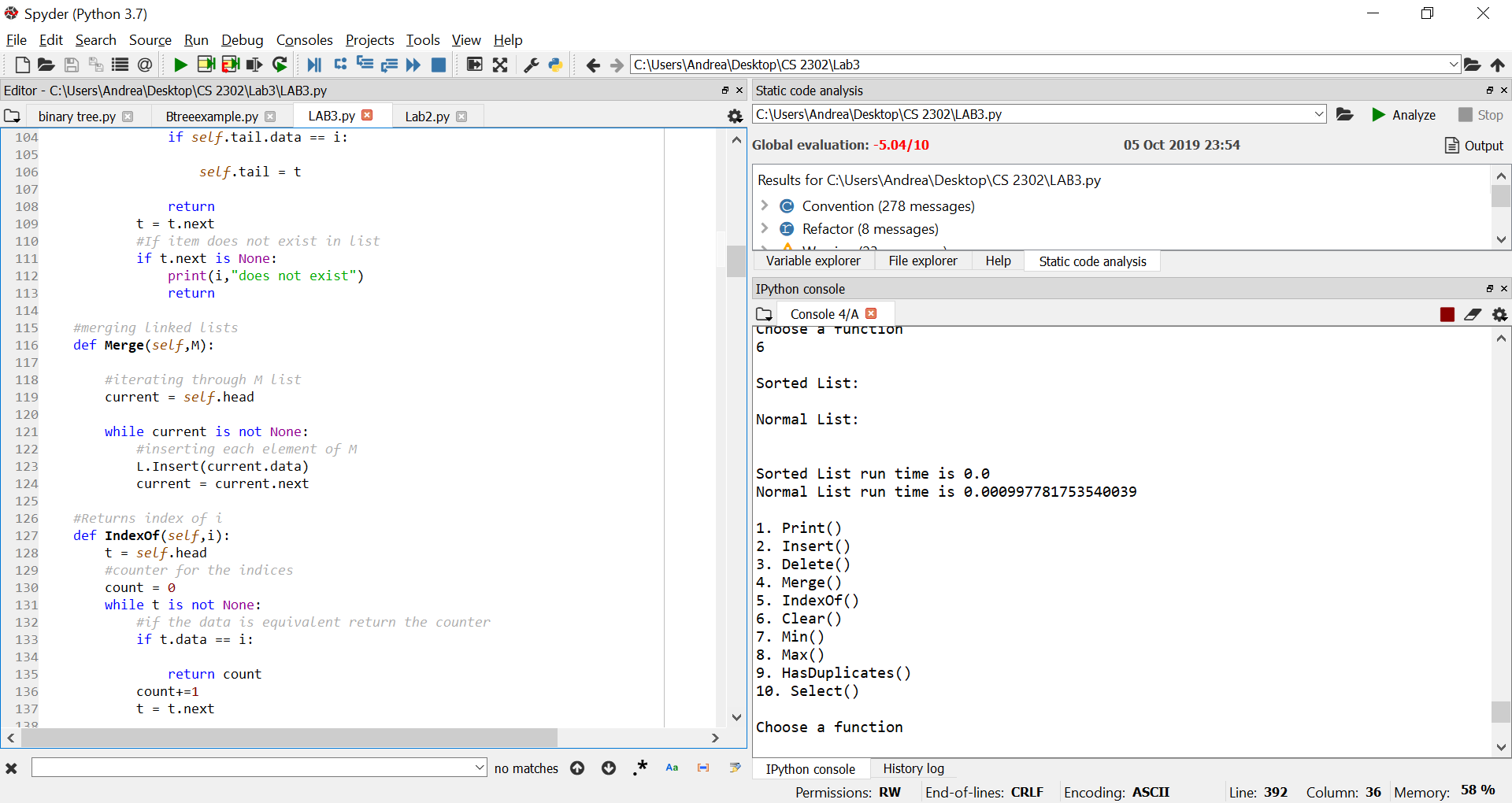
In this lab I learned all about how to implement different types of operations on a linked list that is sorted and on one that is not sorted. I figured out how to insert and delete items of a linked list all while having the list remain sorted. By implementing these functions, I was able to calculate Big-O running times. I then compared the Big-O running times and concluded that the sorted list is faster in most cases. I also learned how to implement two different classes with the same functions. I learned how to adjust certain methods to fit the normal, unsorted list.

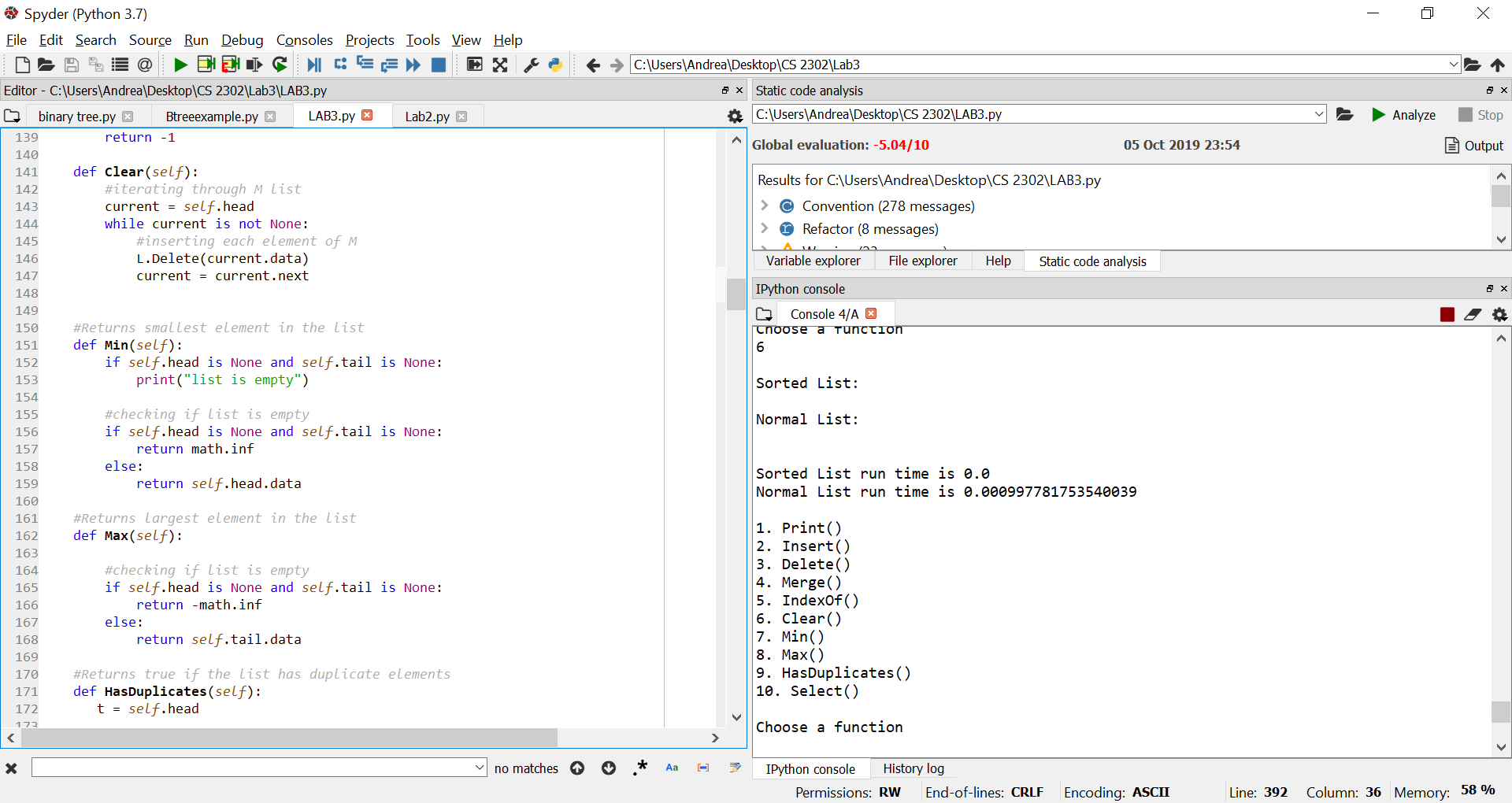
**Appendix**

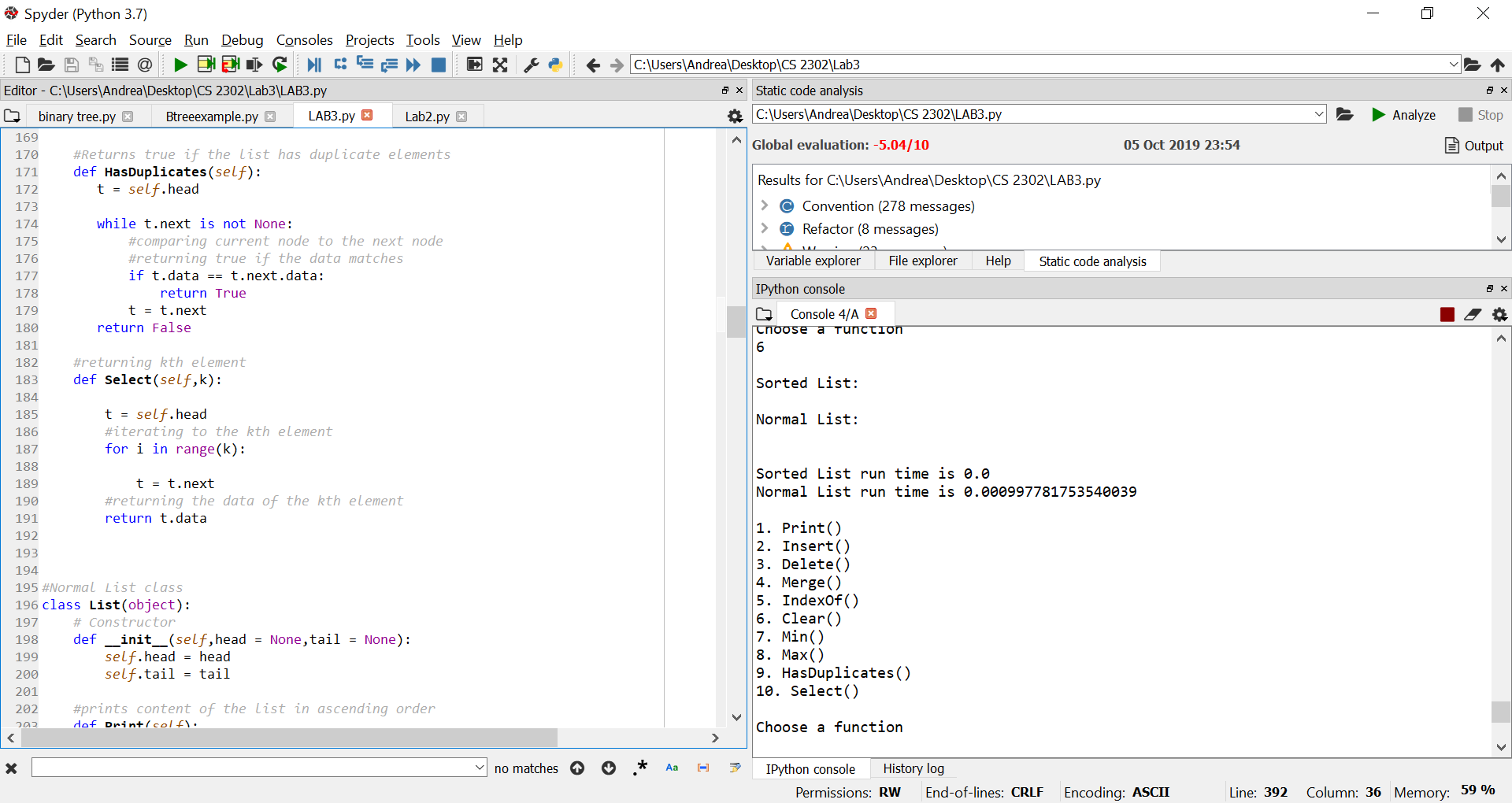


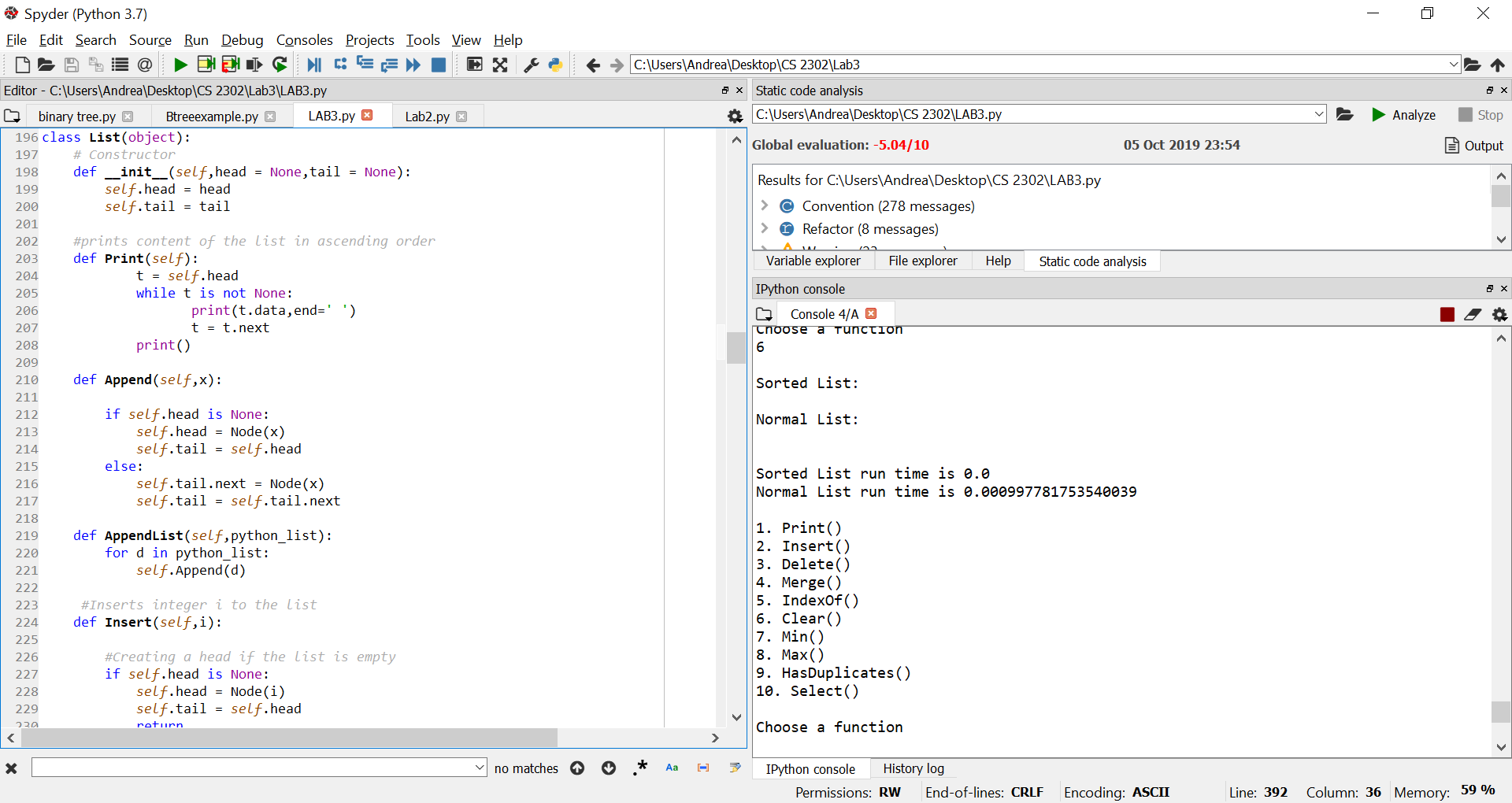


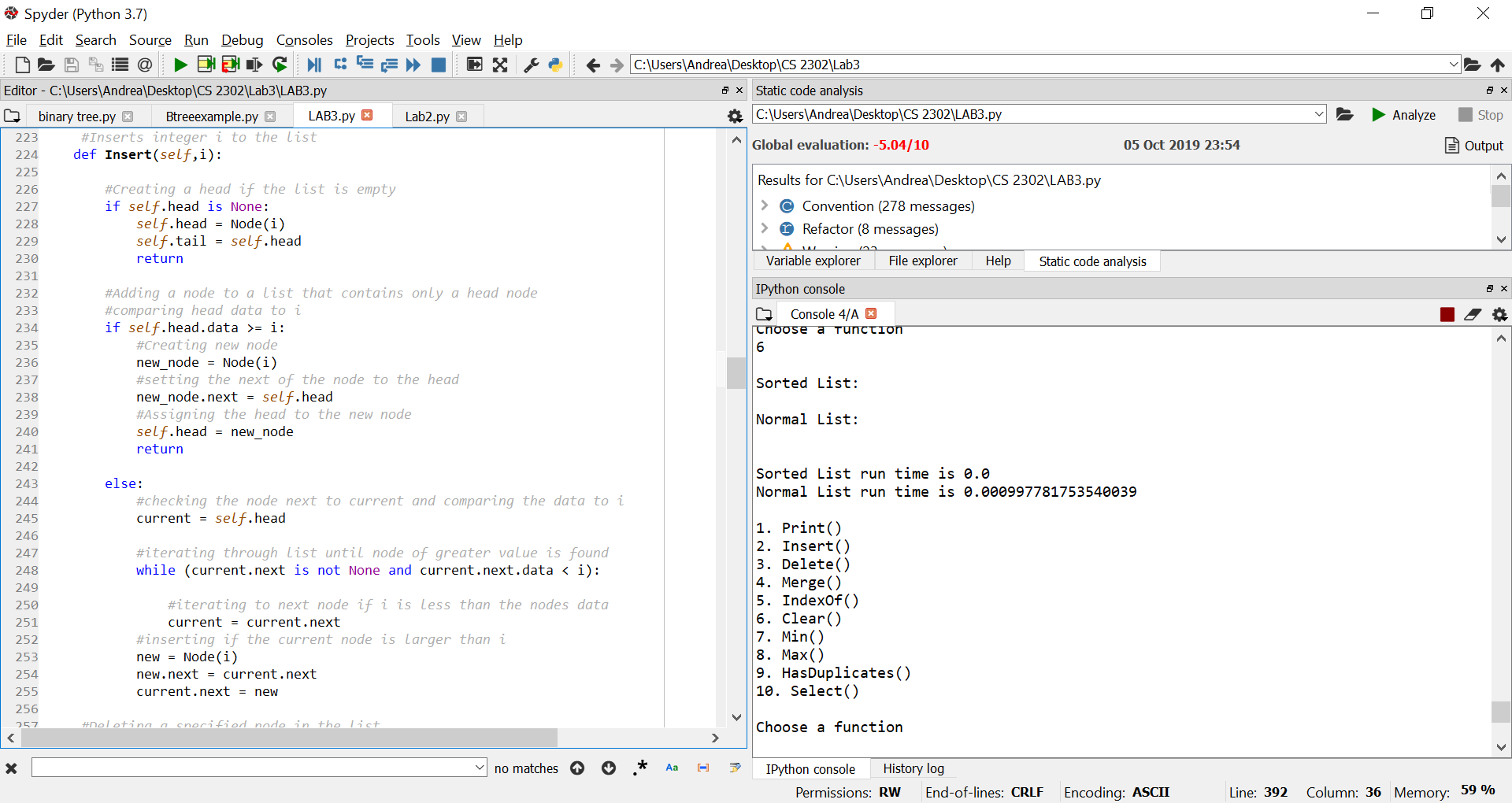


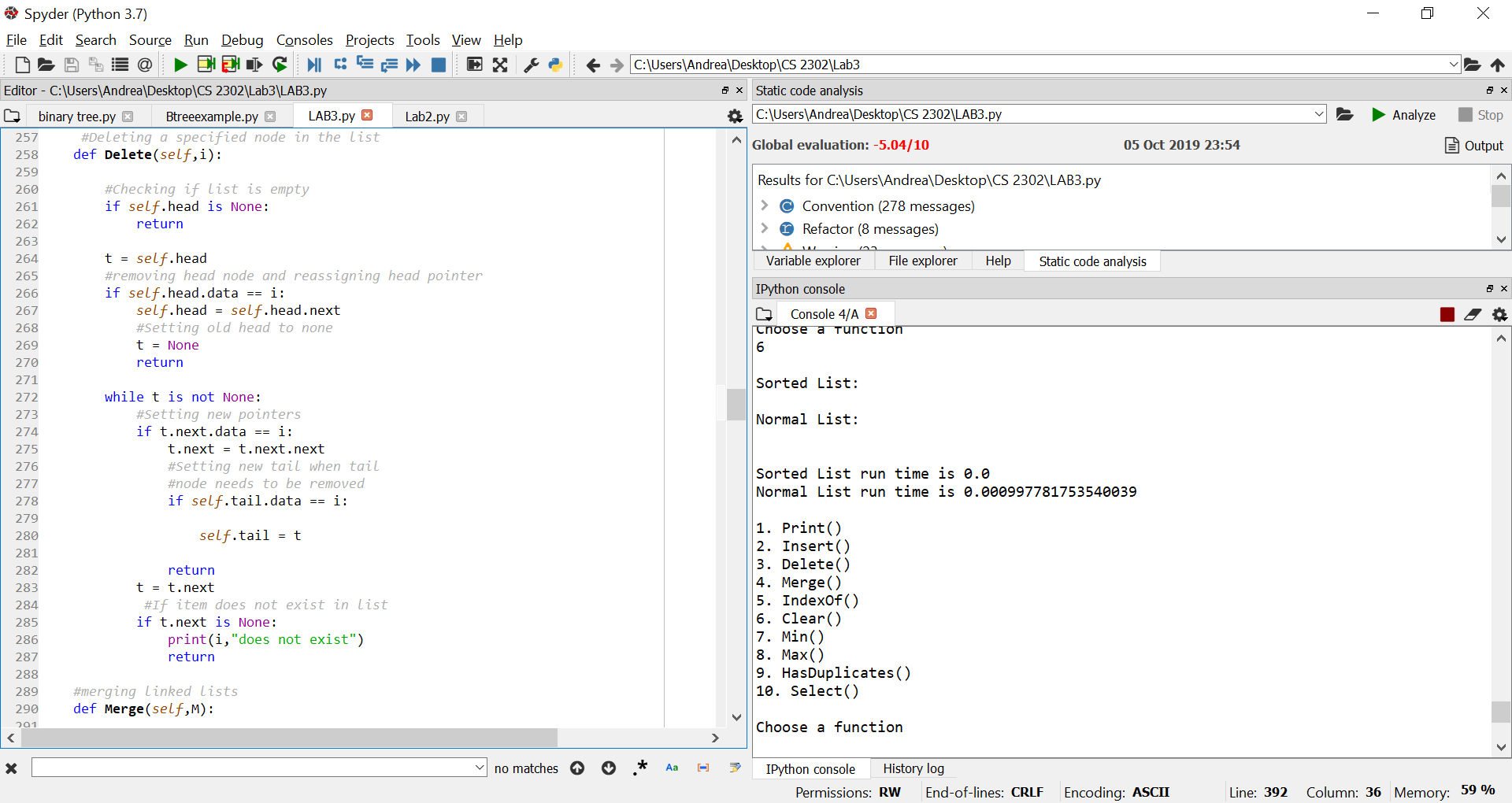


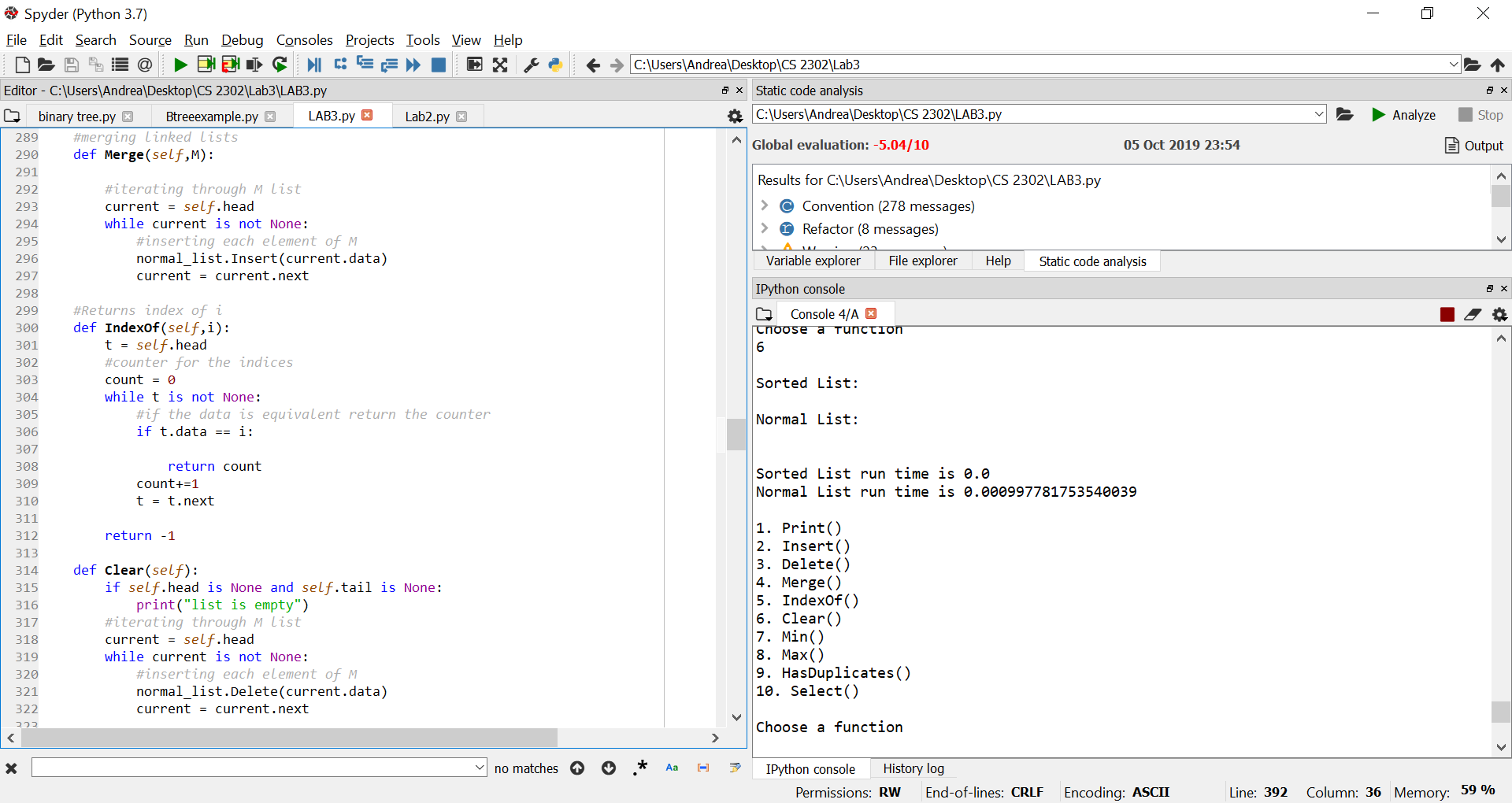


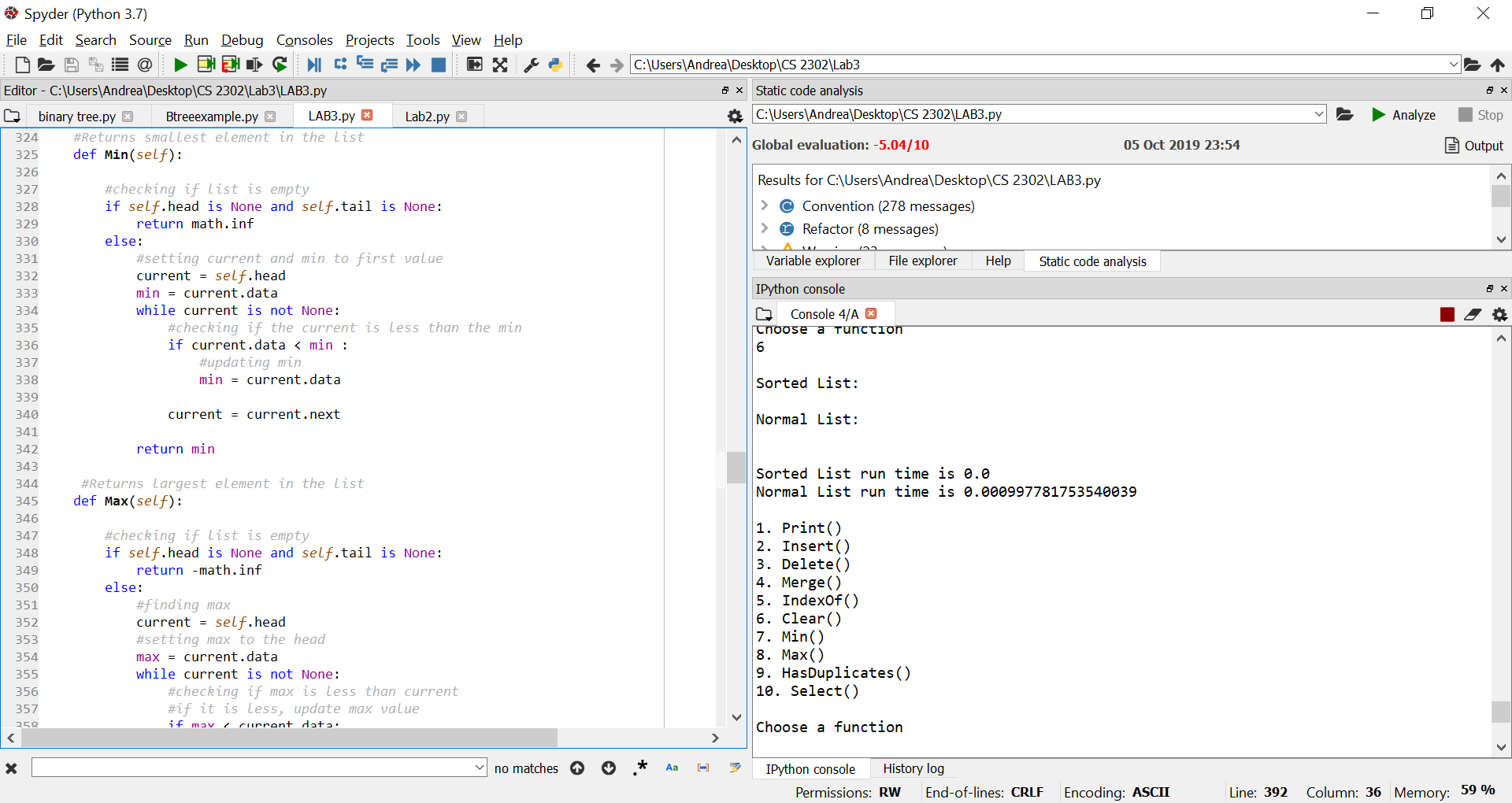


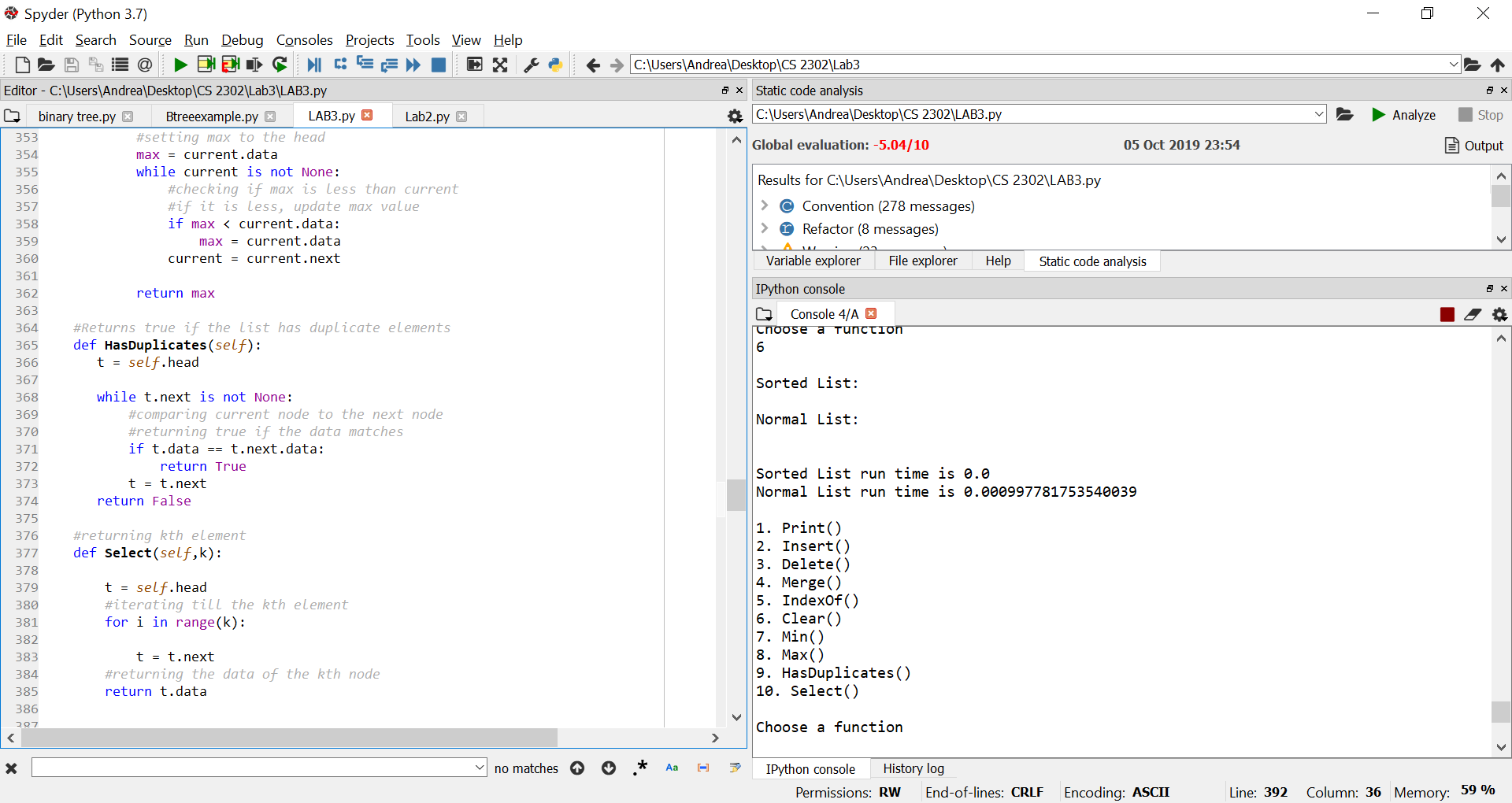


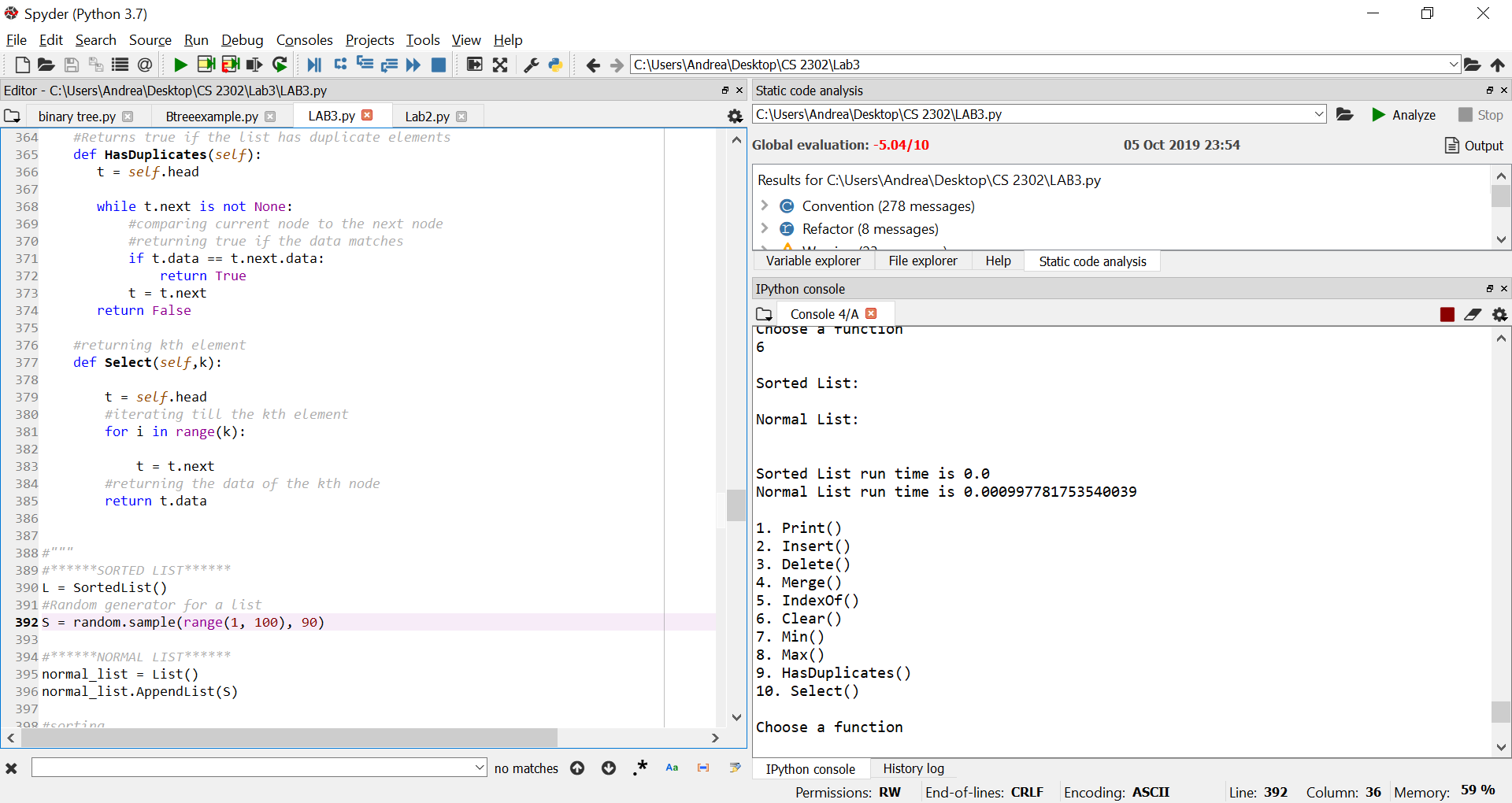


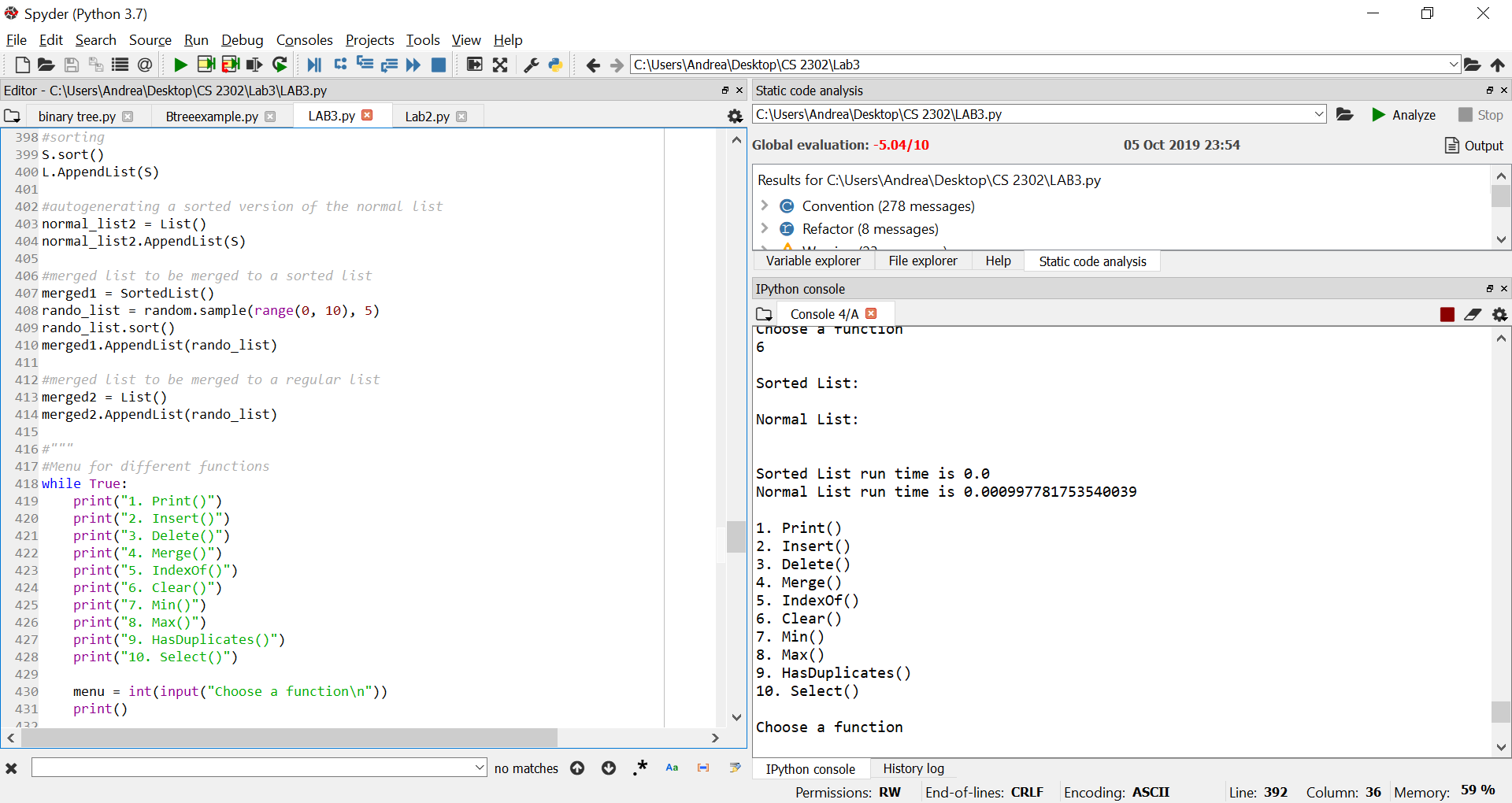


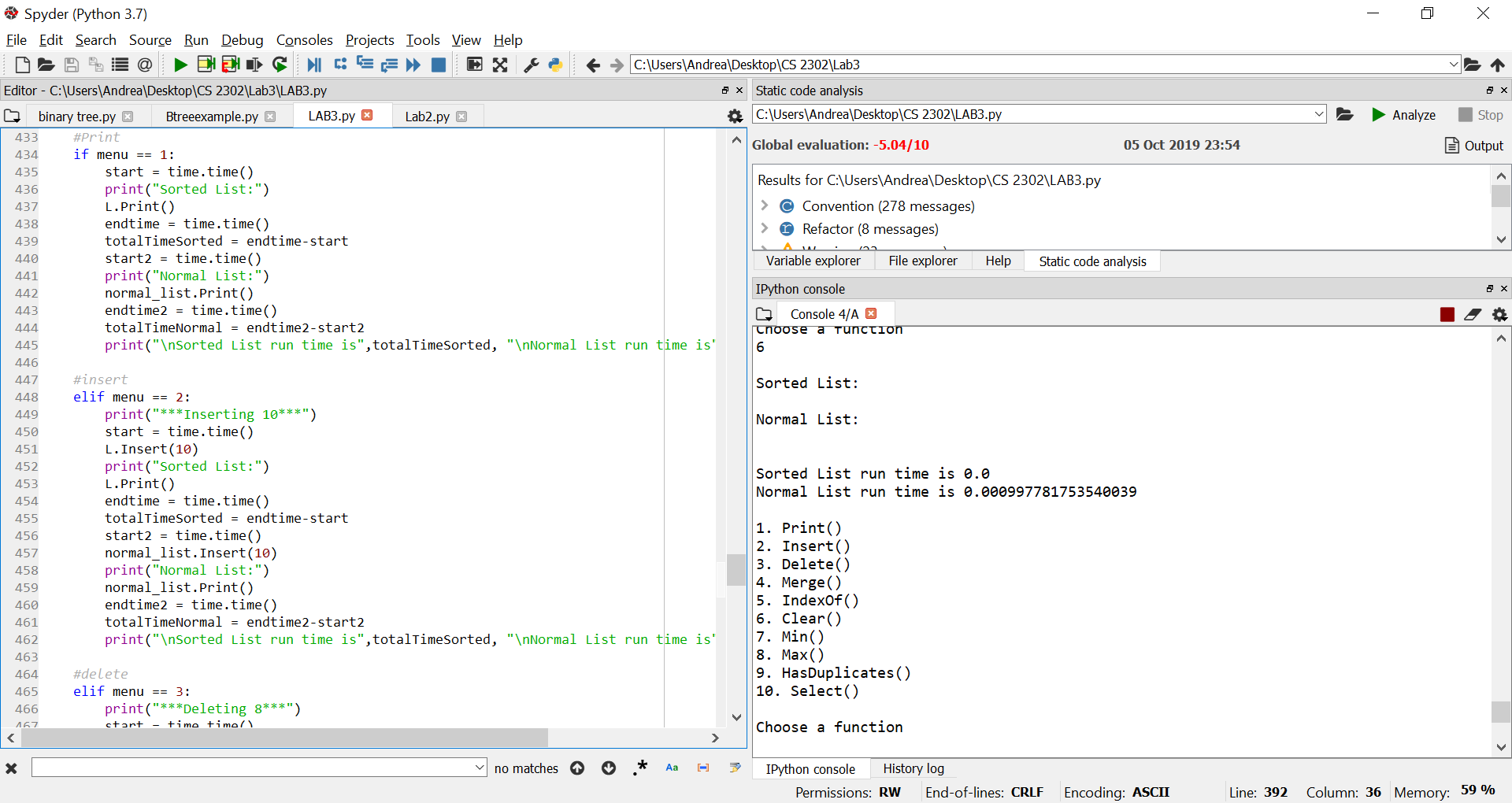


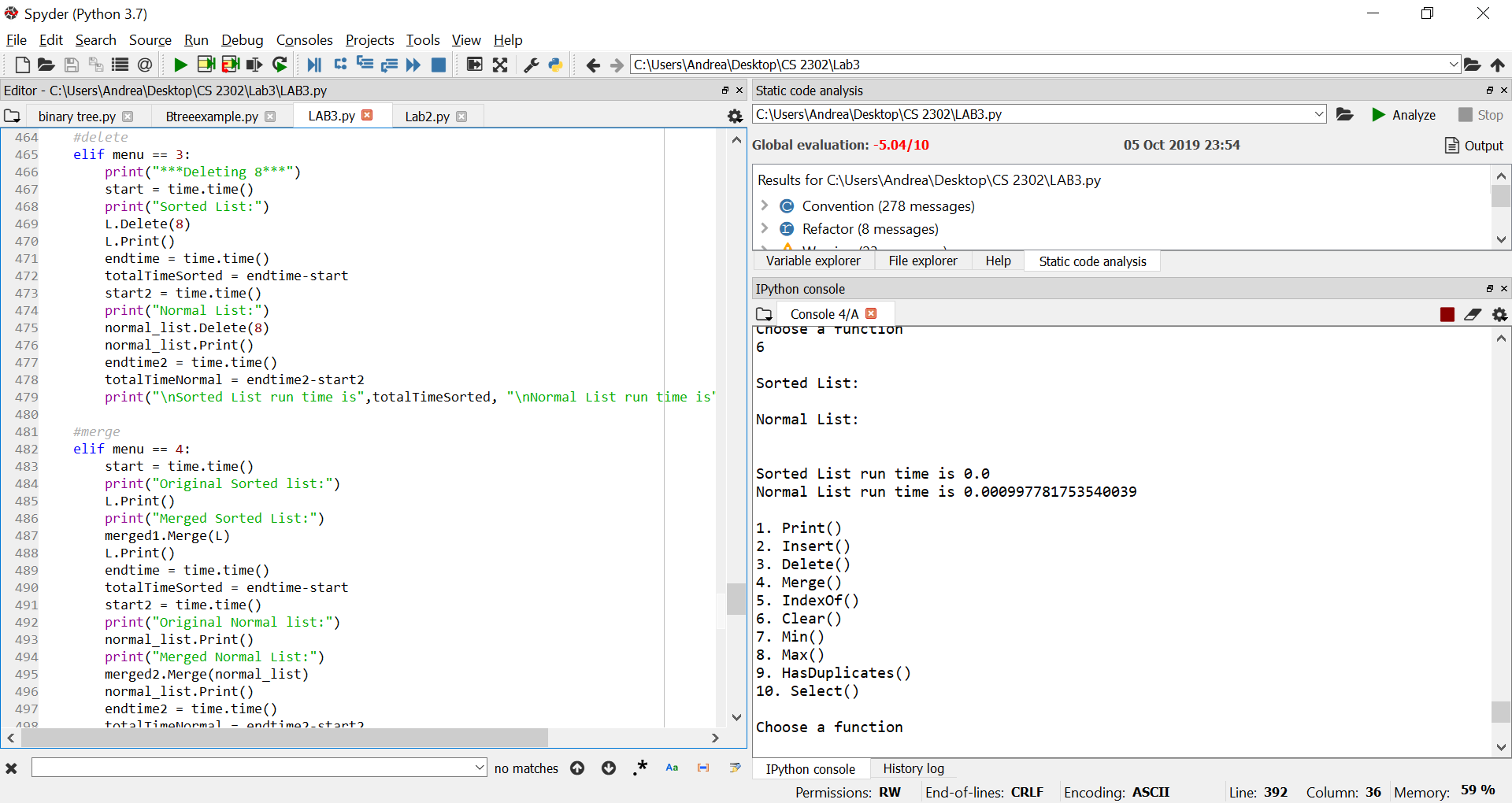


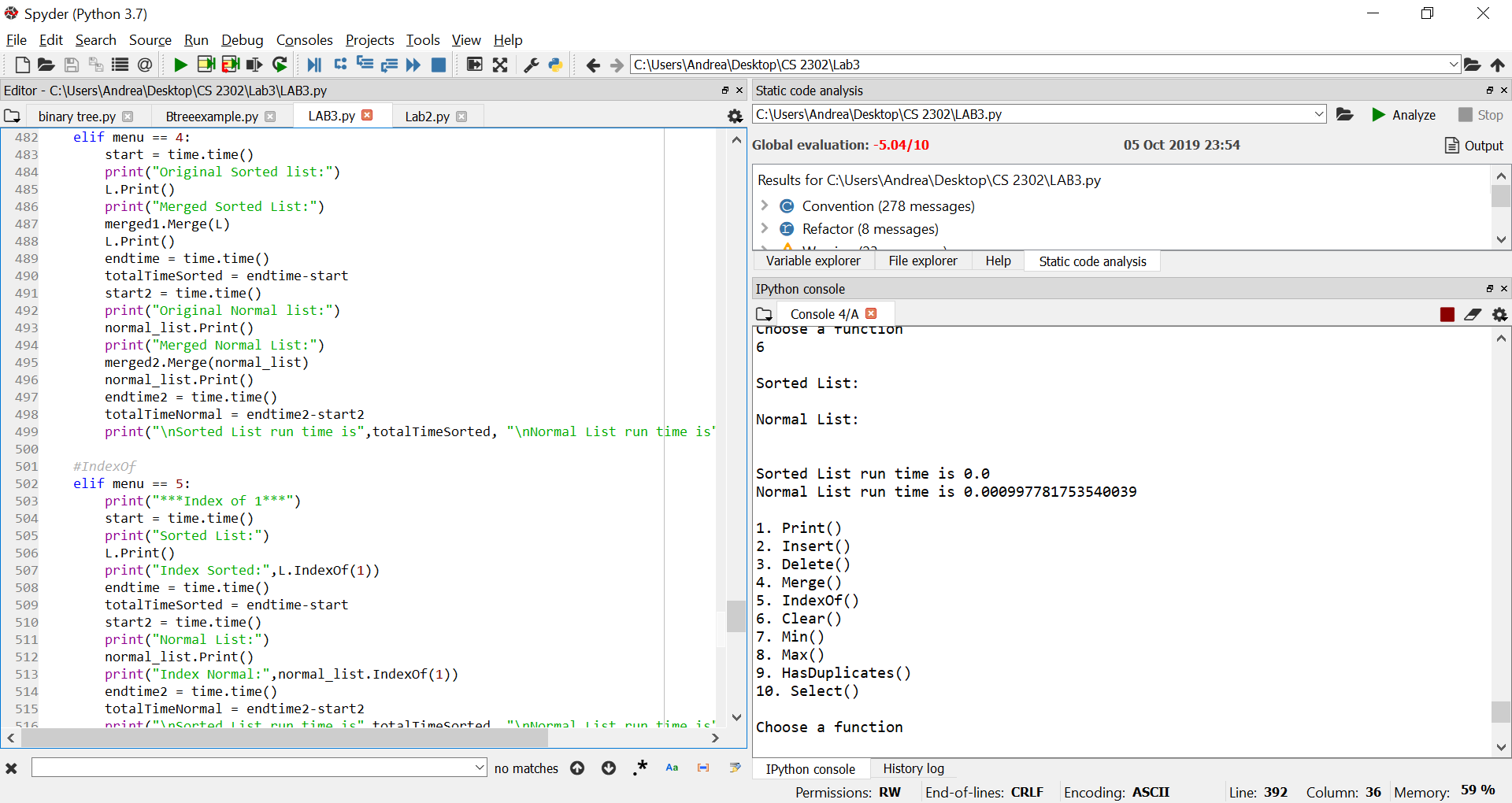


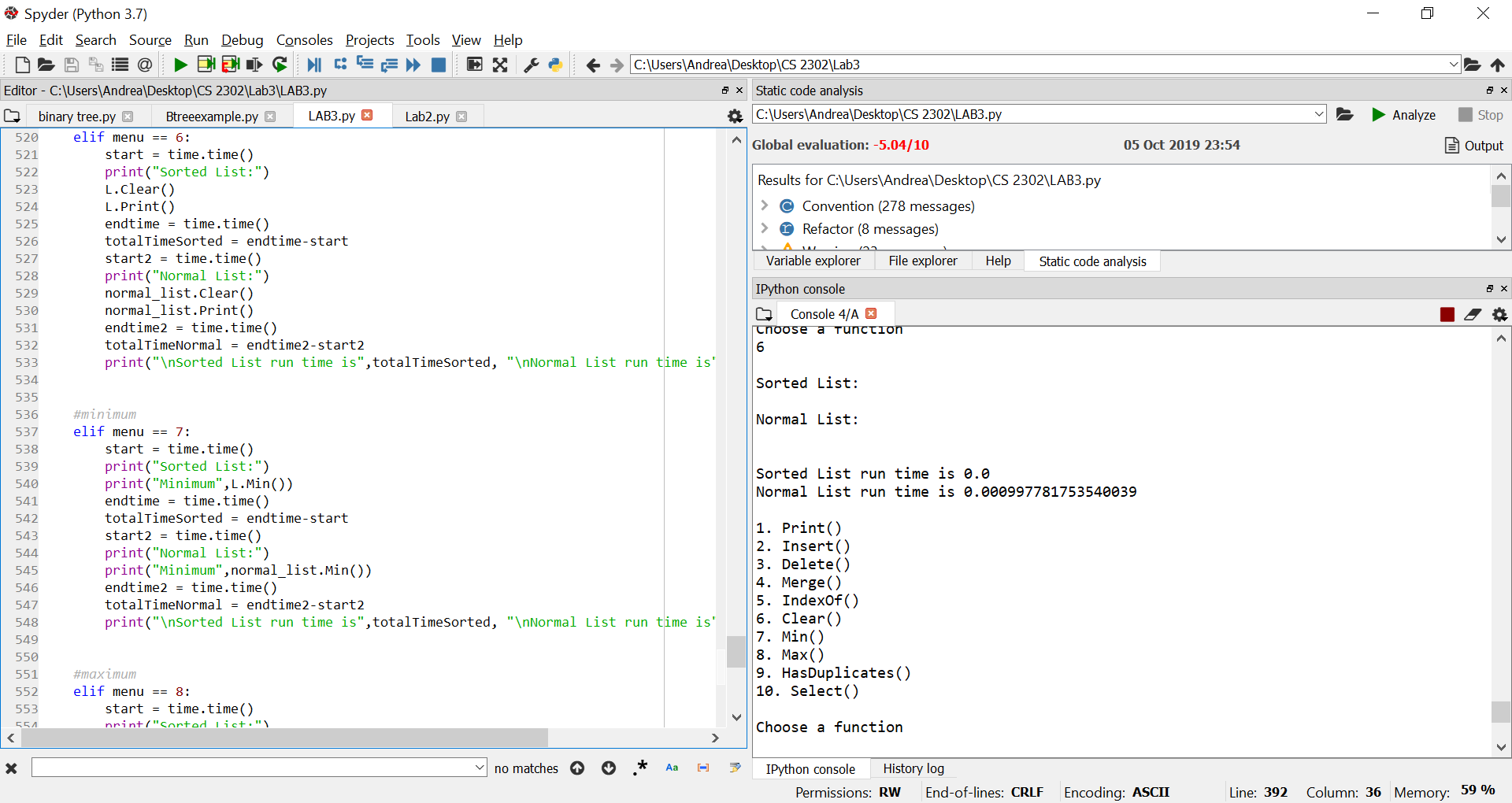


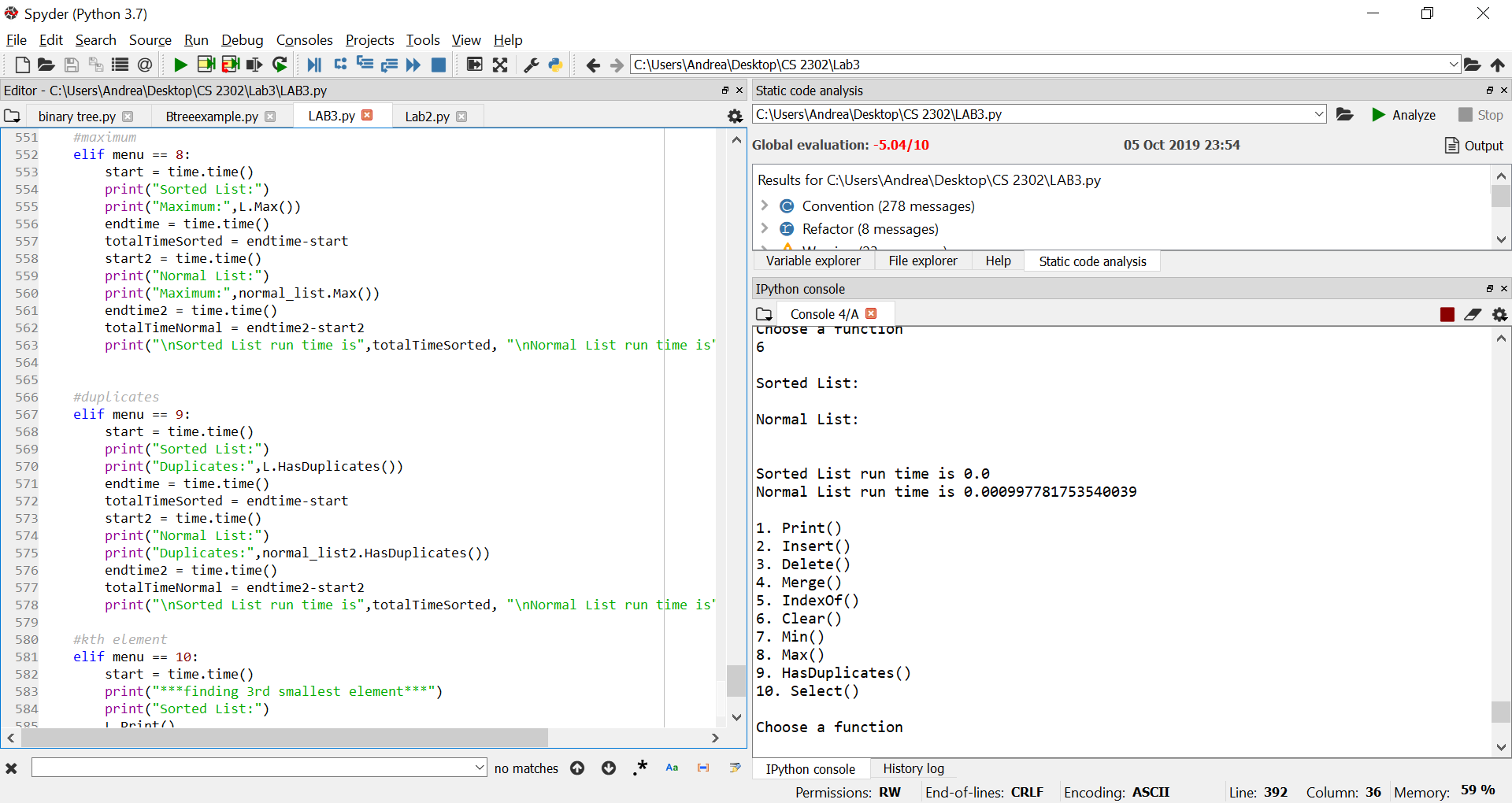


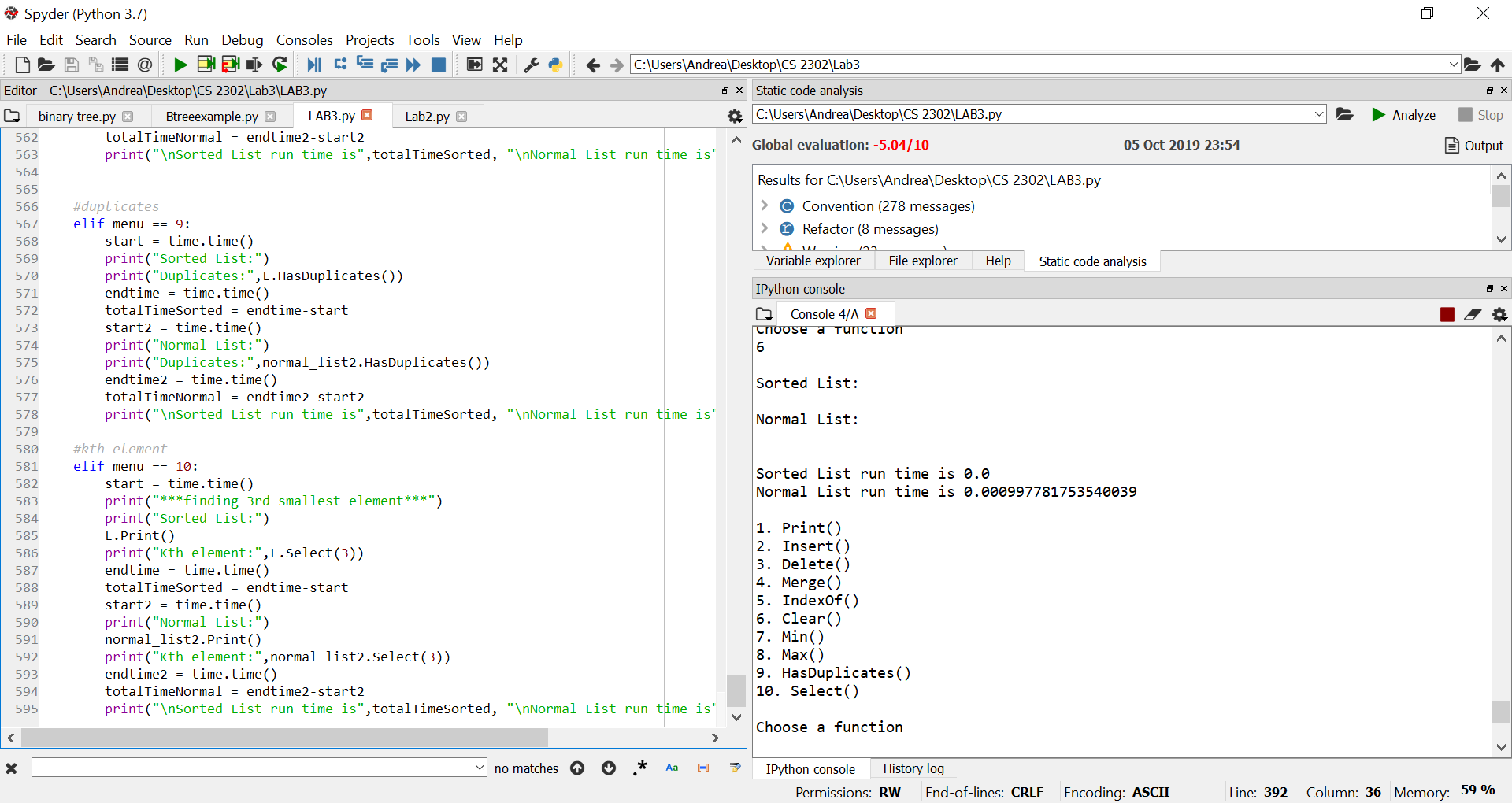












I certify that this project is entirely my own work. I wrote, debugged, and tested the code being presented, performed the experiments, and wrote the report. I also certify that I did not share my code or report or provided inappropriate assistance to any student in the class