**Double Linked Lists**

Allyza Denise Bispo, Ma. Eliza Jebulan, Ishmael Ramirez, John Matthew Remata

Department of Computer Studies and Systems, College of Engineering, University of the East Caloocan

**Introduction**

In a Linked List, each node references the that is next to it. This is a useful demonstration when working with a sequence of elements. However, there are limitations when it comes to working with a Linked List. Double Linked Lists, also known as Doubly Linked Lists(DLL) now comes into the spotlight. These lists allow a greater number of operations, contains an extra pointer called the *previous* pointer, and can be traversed forward and reverse direction.

**Concepts**

Here are some advantages when using Double Linked Lists:

* We can insert and reference a new node before the given node
* The insertion and deletion operation in a DLL are easier to implement
* The deletion operation in a DLL can be more structured if the node to be deleted has been given a pointer to work on.
* Reversal of a DLL is somewhat easy

There are also some disadvantages when working with Double Linked Lists, some of these are:

* When working with DLL, there is the need for additional allotted memory space for storing an extra reference to each node.
* Unlike the Linked List, additional steps are needed to perform insertion and deletion operation in DLL.

In a working DLL, each node has three parts. The first part is the data to be stored, the second is the link to the previous node in the list, and the third is the link to the next node in the list.

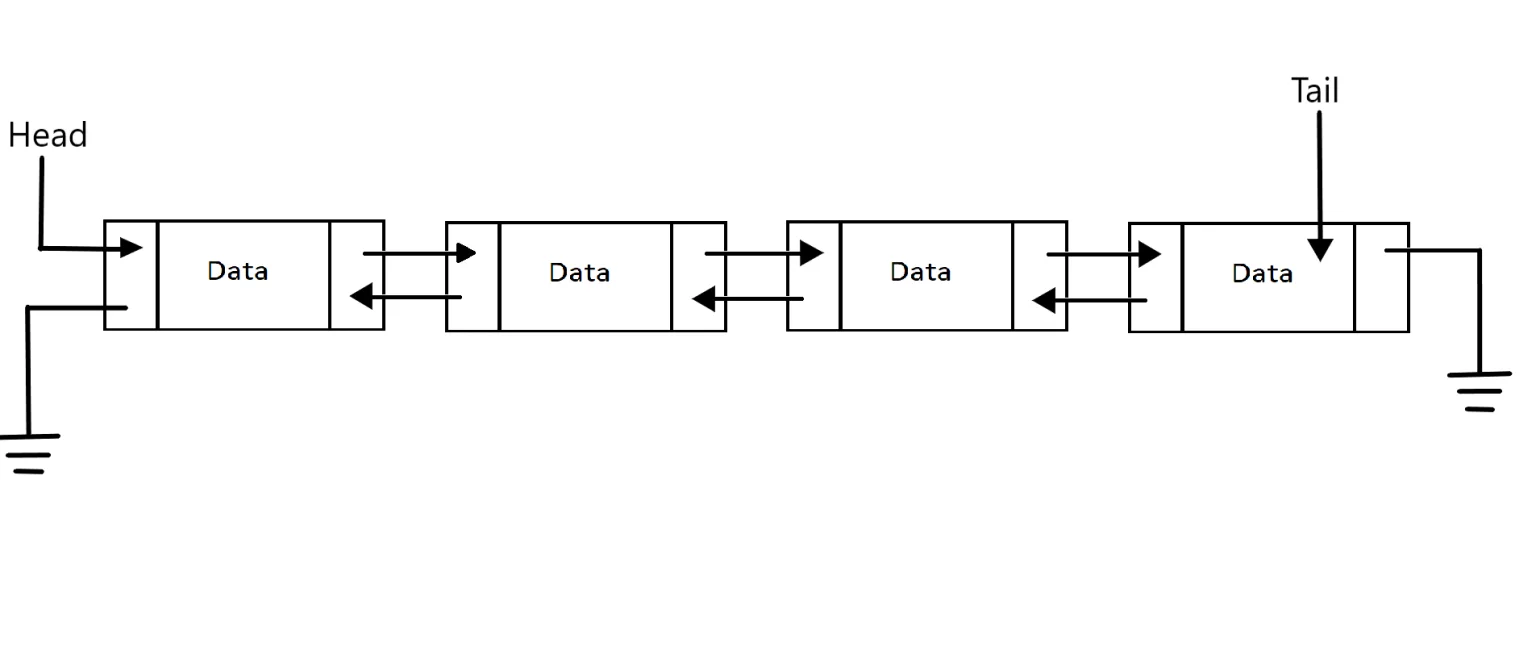
**Application**

With the introduction of traversing forward and backward presented in a DLL, there are various applications that can be integrated into these lists. Some of these are:

* Implementation in web browsers for going forward and going backward in a page.
* The usage of the UNDO and REDO function in some applications
* The possibility of using Double Linked Lists in certain navigational systems that requires forward and backward traversing.
* Can also be used to represent a classic deck of cards

**Simulation**

For a visual representation, here is what a working DLL looks like:



Procedure:

1. The first step is creating a class for the single node in the list

class Node:

    def \_\_init\_\_(self, data):

        self.item = data

        self.nref = None

        self.pref = None

1. Next, we must now create the DoubleLinkedList class that contains its different functions

class DoubleLinkedList:

    def \_\_init\_\_(self):

        self.start\_node = None

1. Inside the DoubleLinkedList class, we will insert an item in an empty list. This method is the easiest way to insert an item.

def insert\_in\_emptylist(self, data):

        if self.start\_node is None:

            new\_node = Node(data)

            self.start\_node = new\_node

        else:

            print("list is not empty")

1. Next method of insertion is inserting an item at the start of the DLL. We can accomplish this by checking first if the list is empty. If this is true, then we can apply the same coding as before. But if the list is not empty, we need to perform three operations:

* For the new node, the reference of the next node is set to self.start\_node.
* For the self.start\_node, the previous node’s reference is now set as the newly inserted node
* Lastly, the self.start\_node now becomes the newly inserted node.

def insert\_at\_start(self, data):

        if self.start\_node is None:

            new\_node = Node(data)

            self.start\_node = new\_node

            print("node inserted")

            return

        else:

            new\_node = Node(data)

            new\_node.nref = self.start\_node

            self.start\_node.pref = new\_node

            self.start\_node = new\_node

1. Having created the insertion at the beginning of the list, we can now proceed to inserting at the end of the list. First, we have to check if the list is empty or not. If the list is empty, we can still use the same logic of insert\_in\_emptylist() method. However, if the list has an element present, we must traverse the list until the reference for the next node is presented as None. This means that we reached the last node

def insert\_at\_end(self, data):

        if self.start\_node is None:

            new\_node = Node(data)

            self.start\_node = new\_node

            return

        n = self.start\_node

        while n.nref is not None:

            n = n.nref

        new\_node = Node(data)

        n.nref = new\_node

        new\_node.pref = n

1. Another method of insertion is inserting an item after another item is called. We now have to routinely check if the list is empty and display the message “the list is empty”. If not, we iterate through all the nodes present in the list. In case we could not find the node after which we want to insert the new node, the program will display a message that the “item is not in the list”. However, if the node is found, we can now perform four operations:

* We can now set the previous reference of the newly inserted node to the node we selected.
* Next is to set the next reference of the node that is newly inserted to the next reference of our selected node.
* If the node we have chosen is not the last node, we can now set the previous reference of the next node after the selected node to the newly added node.
* Finally, set the next reference of the selected node to the newly inserted node.

def insert\_after\_item(self, x, data):

        if self.start\_node is None:

            print("List is empty")

            return

        else:

            n = self.start\_node

            while n is not None:

                if n.item == x:

                    break

                n = n.nref

            if n is None:

                print("item not in the list")

            else:

                new\_node = Node(data)

                new\_node.pref = n

                new\_node.nref = n.nref

                if n.nref is not None:

                    n.nref.prev = new\_node

                n.nref = new\_node

1. For inserting an item before another item, we must check again if the list is empty. If it is, then the program will display “the list is empty”. Otherwise, we apply the same logic as inserting after another item. We iterate through all the nodes present in the list. If our chosen node before which we want to insert a new node is not present, the program will display again a message that “item is not in the list”. Nevertheless, if the node exists, it is now selected, and we can perform four operations that is a little opposite of the ones before:

* We must set the next reference of the newly inserted node.
* The previous reference of the new node is now the previous reference of the selected node.
* The next reference of the previously selected node is now set as the newly added node
* Lastly, the previous reference of the selected node is now the newly added node.

def insert\_before\_item(self, x, data):

        if self.start\_node is None:

            print("List is empty")

            return

        else:

            n = self.start\_node

            while n is not None:

                if n.item == x:

                    break

                n = n.nref

            if n is None:

                print("item not in the list")

            else:

                new\_node = Node(data)

                new\_node.nref = n

                new\_node.pref = n.pref

                if n.pref is not None:

                    n.pref.nref = new\_node

                n.pref = new\_node

1. For deletion, the easiest way to delete an element in the list is from the start. Again, we have to check if the list is empty and if not, we must check again if the list only has one element or not and if it has, we can just set the starting node to none. Then, we can set the value of the starting node to the next node and now set the previous reference of the starting node to none.

def delete\_at\_start(self):

        if self.start\_node is None:

            print("The list has no element to delete")

            return

        if self.start\_node.nref is None:

            self.start\_node = None

            return

        self.start\_node = self.start\_node.nref

        self.start\_prev = None

1. Next method of deletion is deleting an item from the end. First, we must check again if the list is empty. If the list however, contains a single element. We can set again the starting node to none. Now, if the list has more than one element, we traverse the list until we have reached the last node. Once there, we can set the next reference of the node previous to the last node to none. This means that the node is now removed.

def delete\_at\_end(self):

        if self.start\_node is None:

            print("The list has no element to delete")

            return

        if self.start\_node.nref is None:

            self.start\_node = None

            return

        n = self.start\_node

        while n.nref is not None:

            n = n.nref

        n.pref.nref = None

1. The last method of deletion is deleting an item by its value. This is the hardest out of all the deletion methods given. This method takes the given parameter and deletes it from the list. First, the program will check if the list is empty.

if self.start\_node is None:

            print("The list has no element to delete")

            return

If the list is not empty, we have to check if the list has only one item and if that is the item we want to delete. If this is true then we can simply set the self.start\_node to none which deletes that item from the list. However, if there is only one item and that is not what we want to delete, the program will display “item not found”.

if self.start\_node.nref is None:

            if self.start\_node.item == x:

                self.start\_node = None

            else:

                print("Item not found")

            return

Now we move to a list where there is more than one item but the item we want to delete is the first item. For this instance, we can just apply the program we used on delete\_at\_start() method.

if self.start\_node.item == x:

            self.start\_node = self.start\_node.nref

            self.start\_node.pref = None

            return

Lastly, if the list actually has more than one item and the item we want to be removed is not the first item, we can go through the list except the last node and check if the node we want to be deleted matches any of the nodes in the list. If the node is found, we can now do 2 operations:

* We can set the value of the next reference of the previous node to be the next reference of the node that will be removed.
* Lastly, we can set the previous value of the next node to the previous reference of the node that will be removed.

 n = self.start\_node

        while n.nref is not None:

            if n.item == x:

                break;

            n = n.nref

        if n.nref is not None:

            n.pref.nref = n.nref

            n.nref.pref = n.pref

        else:

            if n.item == x:

                n.pref.nref = None

            else:

                print("Element not found")

For the method of traversing the DLL, the program will be:

def traverse\_list(self):

        if self.start\_node is None:

            print("List has no element")

            return

        else:

            n = self.start\_node

            while n is not None:

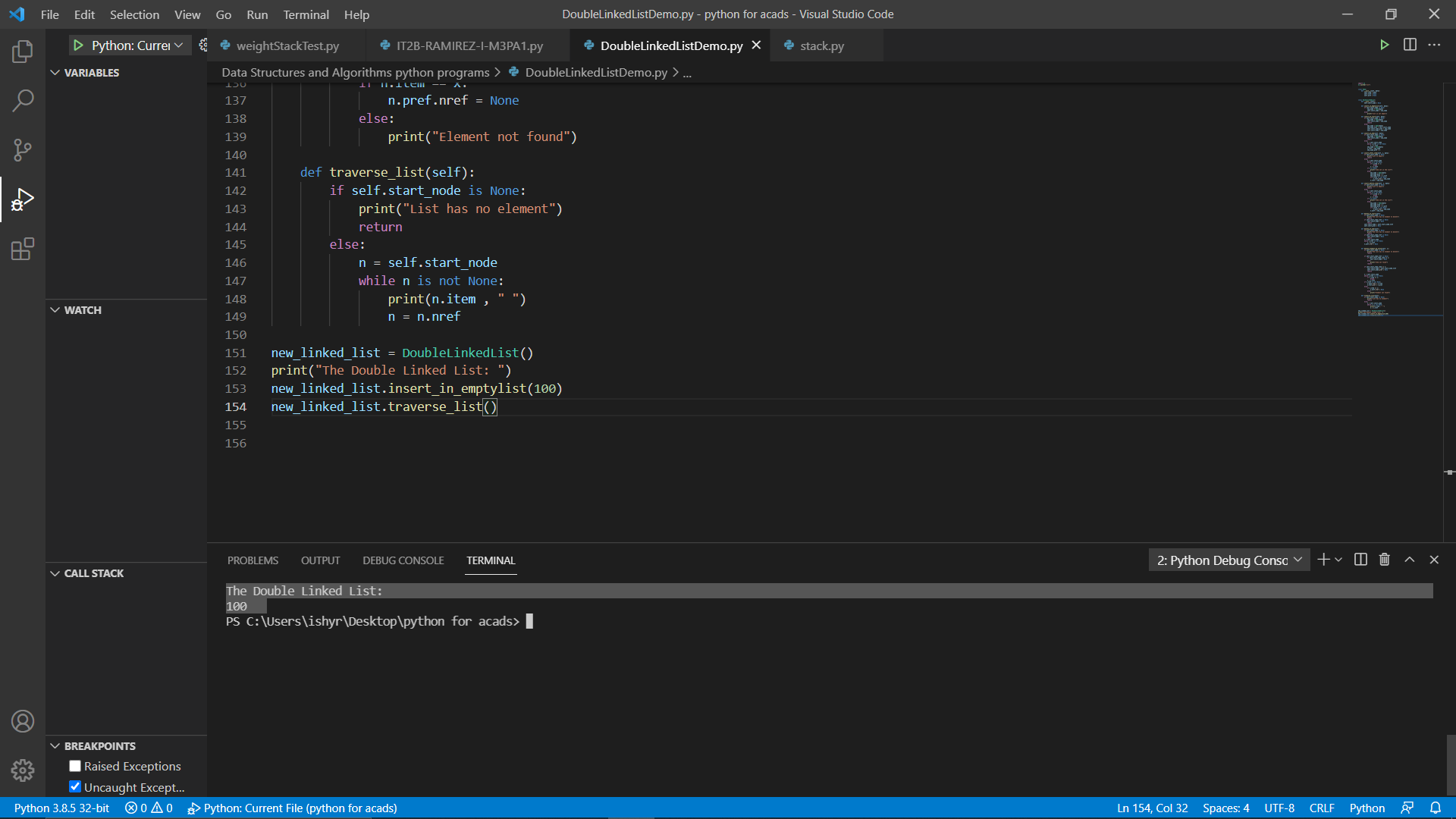
                print(n.item , " ")

                n = n.nref

After creating all those methods for DLL, we can now apply and test them. For testing insertion function:

new\_linked\_list.insert\_in\_emptylist(100)

For traversing the list, the output will be:

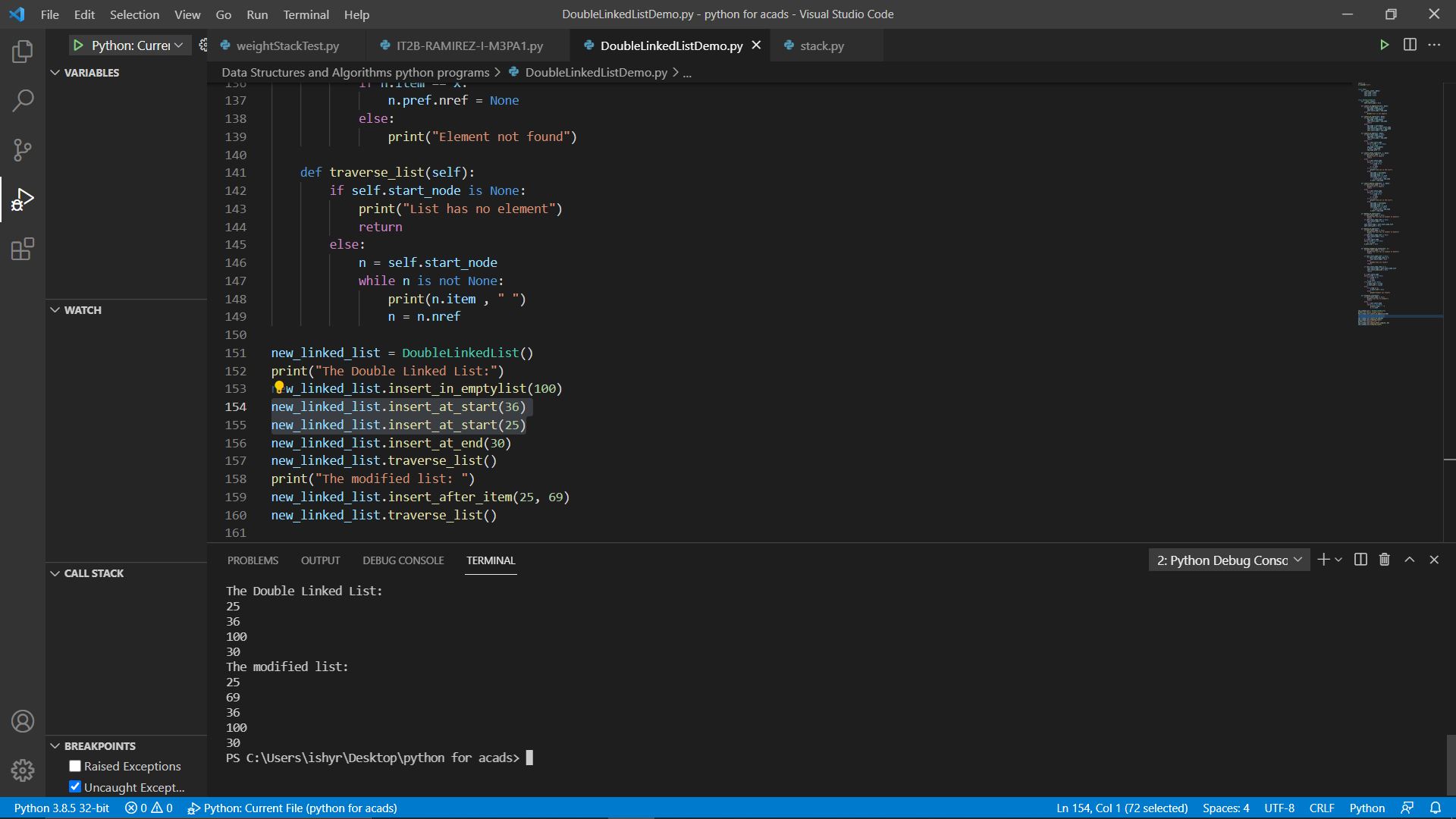


Inserting additional items:

new\_linked\_list.insert\_at\_start(36)

new\_linked\_list.insert\_at\_start(25)

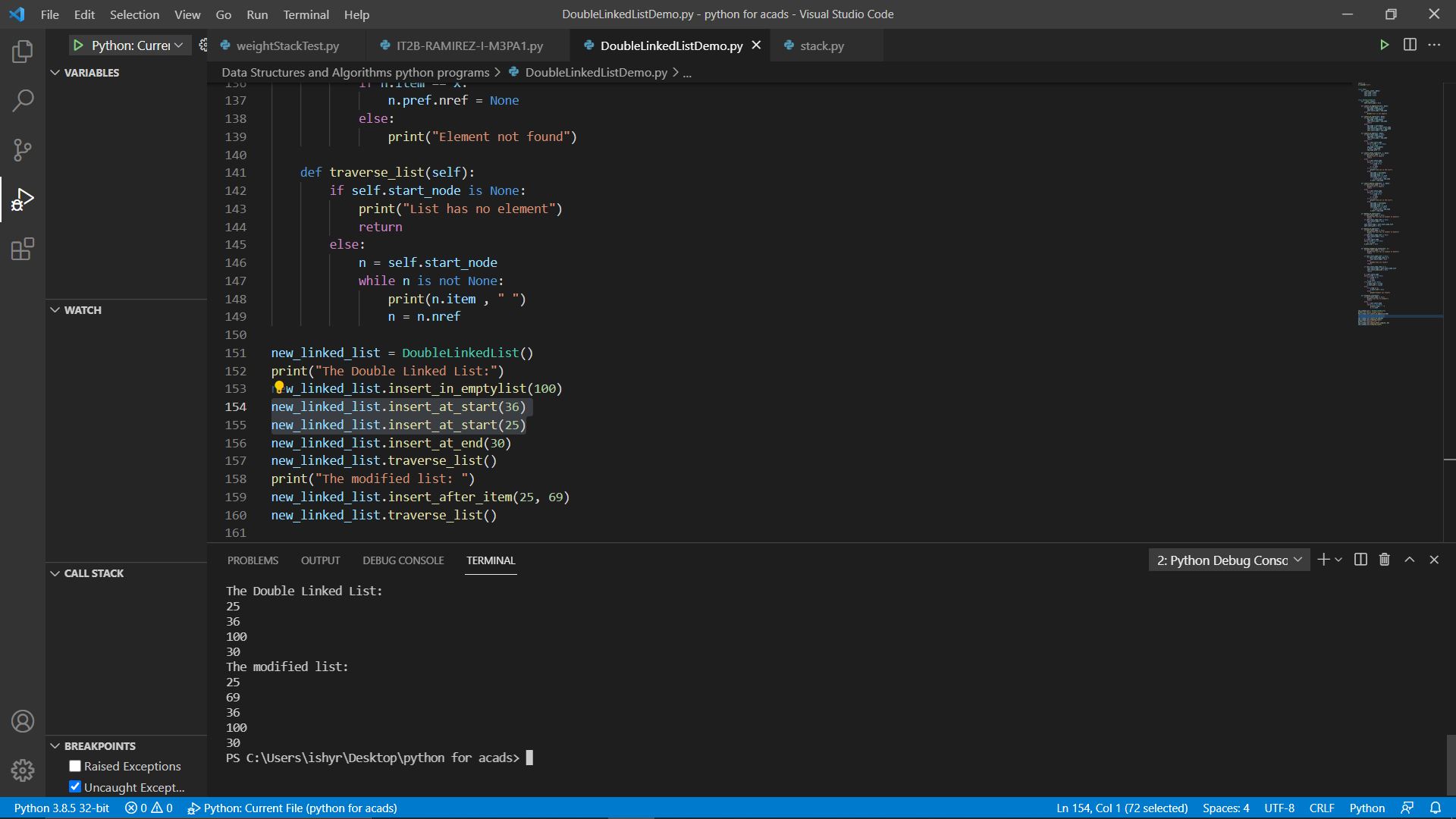
The output now will be:



Adding an item at the end of the list:

new\_linked\_list.insert\_at\_end(30)

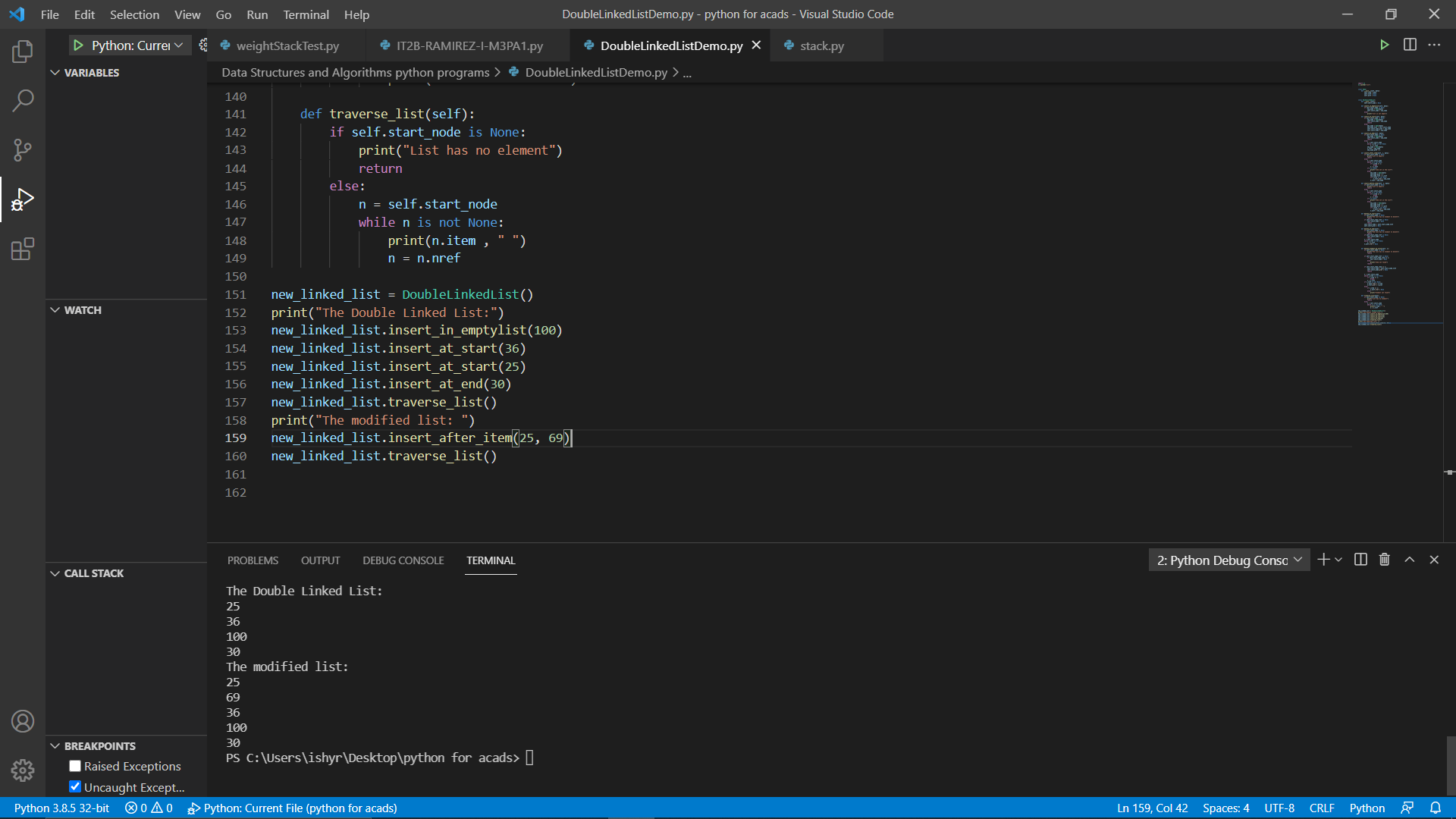
The new output will be:



We have an item number 25. if we want to insert an item after that, the program will be:

new\_linked\_list.insert\_after\_item(25, 69)

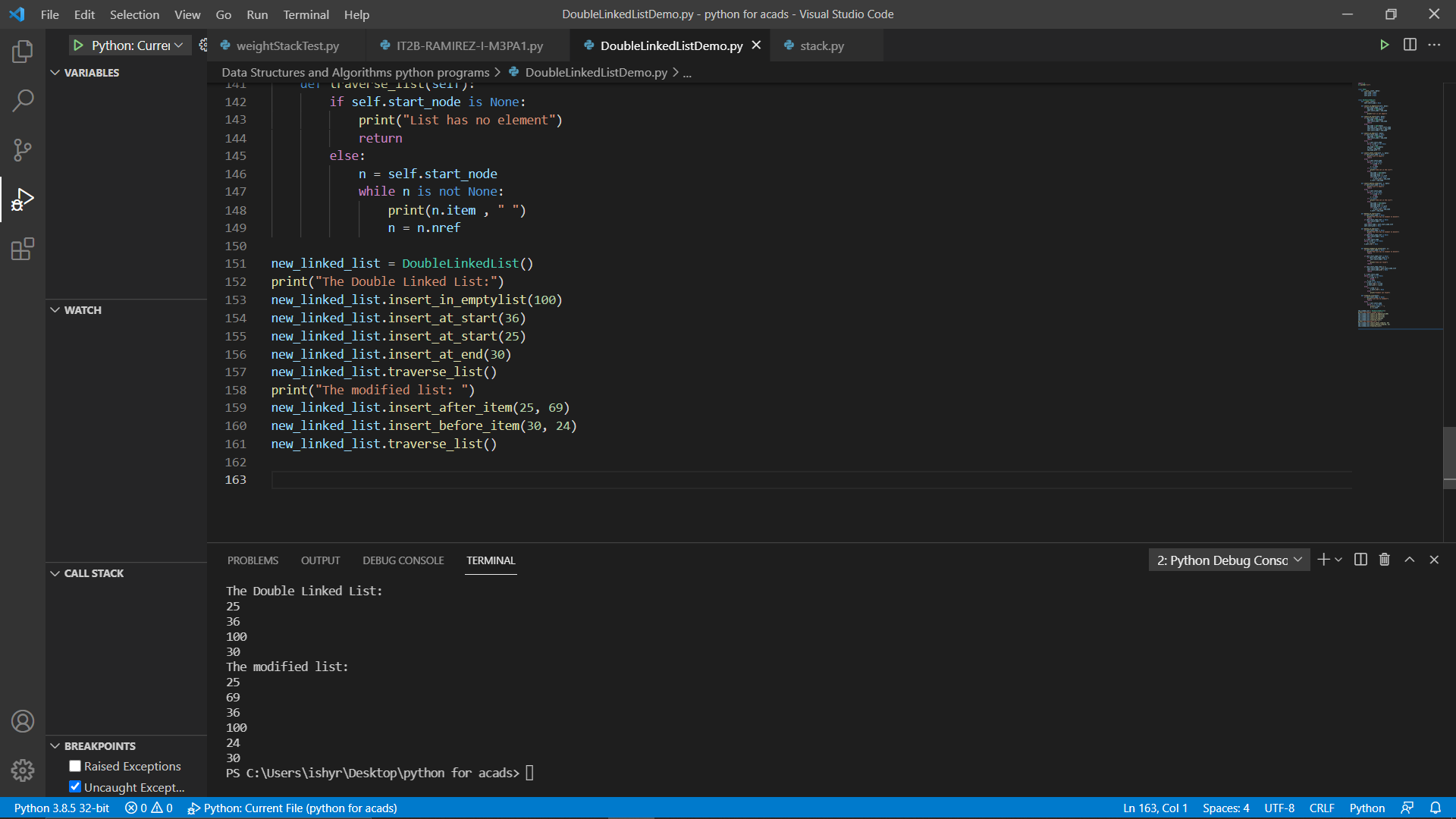
And the output will be:



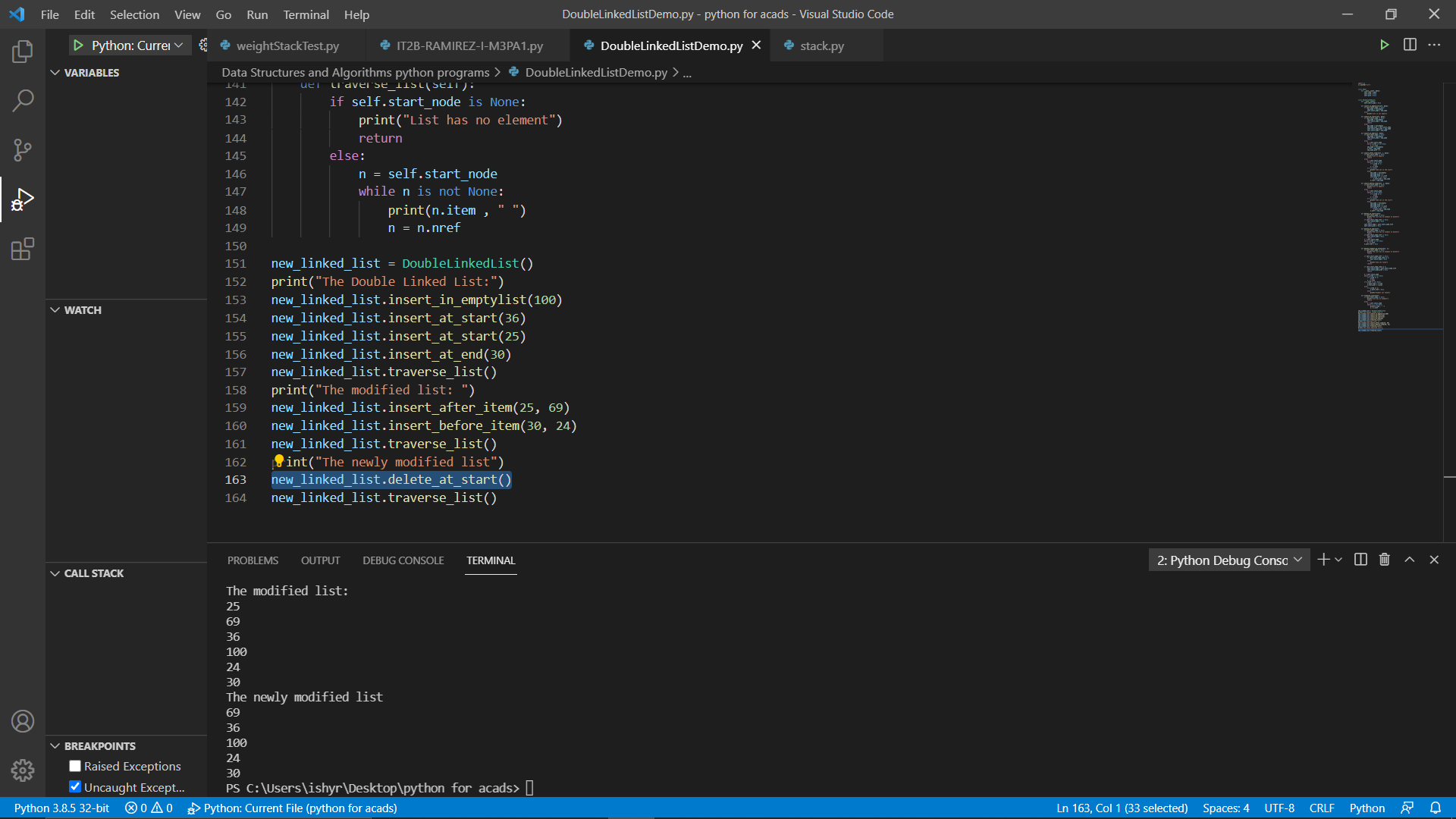
Inserting a number before 30, the program will be:

new\_linked\_list.insert\_before\_item(30, 24)

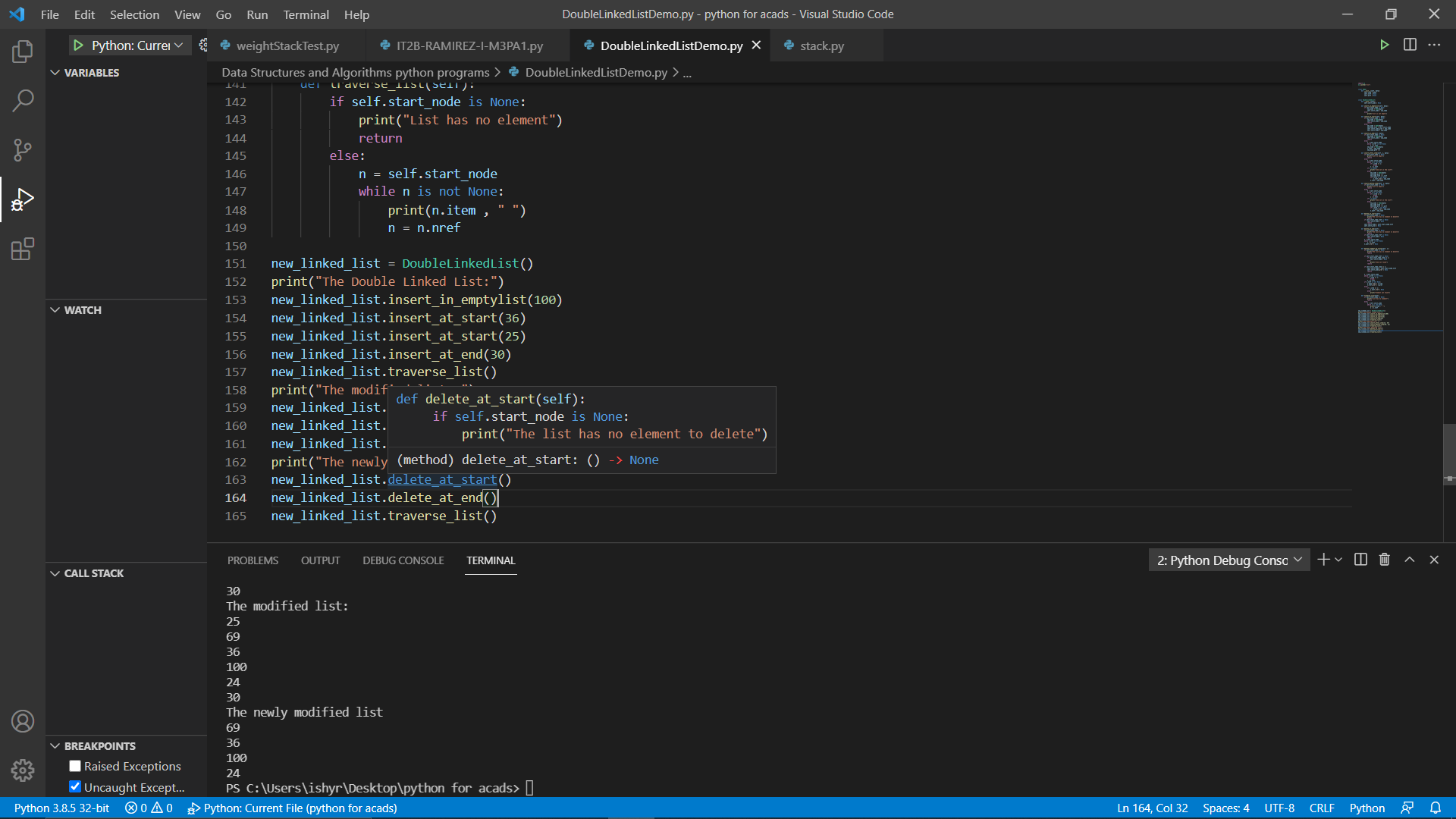
The output is:



Next, we can now test deletion functions in DLL. For deleting the first item in DLL, we will use the delete\_at\_start() method. The output will be:



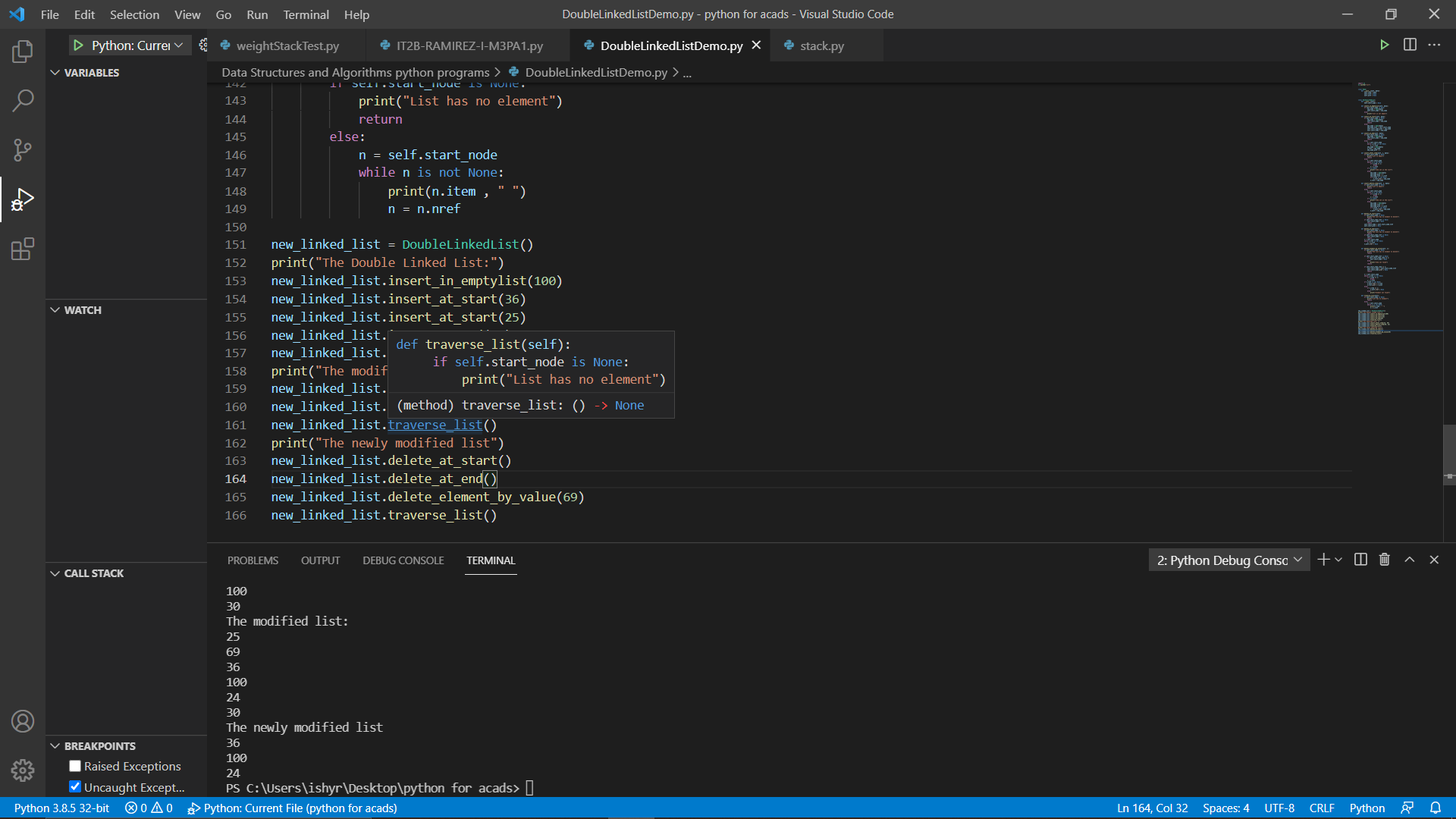
The item 25 is now removed from the DLL. After deleting the starting item, we can now delete the last item in the list using the delete\_at\_end() method. The output should now be:



Lastly, we can delete an item by its value using the method delete\_element\_by\_value() as shown:

new\_linked\_list.delete\_element\_by\_value(69)

The output is:



The Double Linked List is effective specifically when used in performing many insertion and deletion operations. Having links between the nodes makes it easy in inserting and deleting items on the list.

**Glossary**

***Deletion***– the method of removing an item in a list

***Double Linked List*** – collection of nodes linked in a sequence. Has a link to the previous and next node.

***Head*** – the starting node in a list

***Insertion*** – the method of adding an item in a list

***Node*** – the elements present in the list

***Traversing*** – going through the entire list

**References**

* Doubly Linked List: Set 1 (Introduction and Insertion). GeeksforGeeks. (2021, March 9). https://www.geeksforgeeks.org/doubly-linked-list/.
* Goodrich, T., Tamassia R., & Goldwasser, M. (2013) Data Structure & Algorithms in Python, John Wiley & Sons Inc.
* Malik, U. (n.d.). Doubly Linked List with Python Examples. Stack Abuse. <https://stackabuse.com/doubly-linked-list-with-python-examples/>