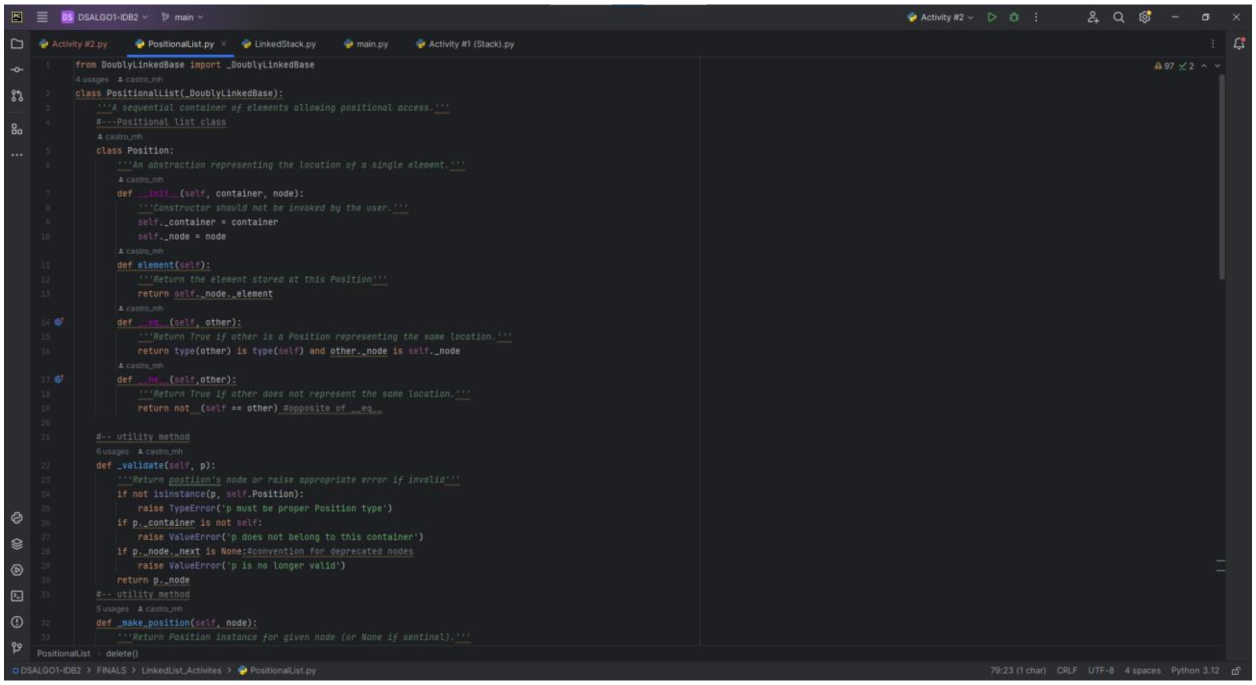
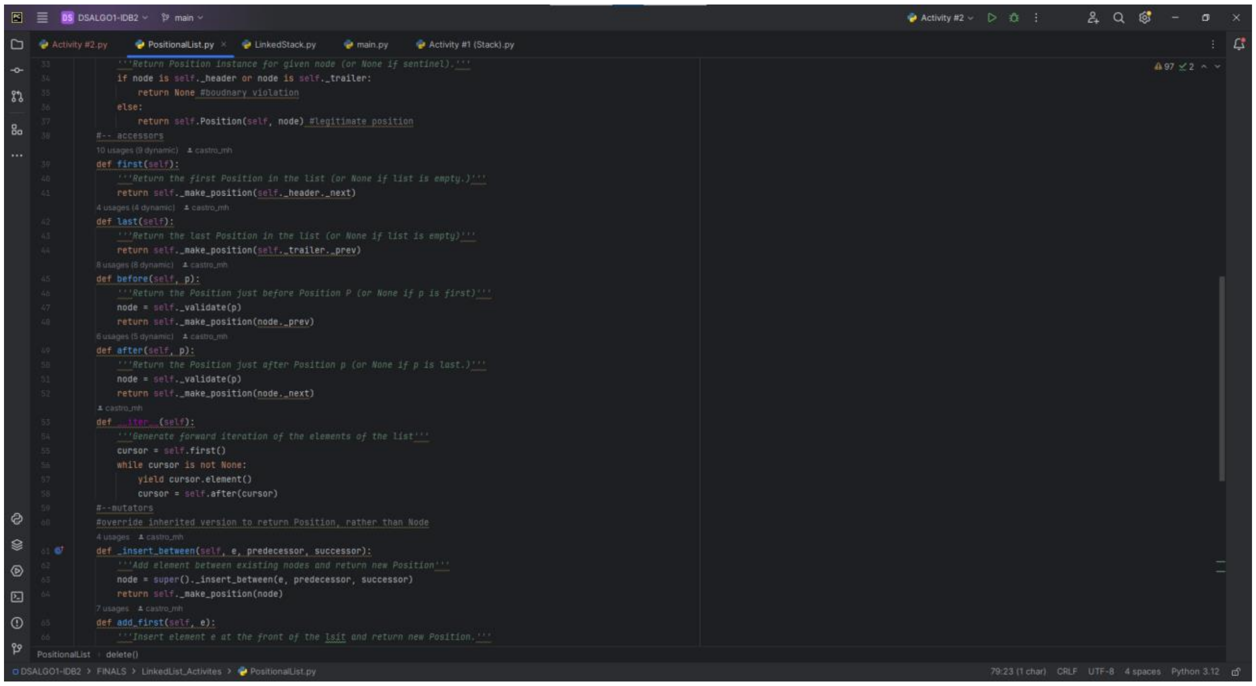
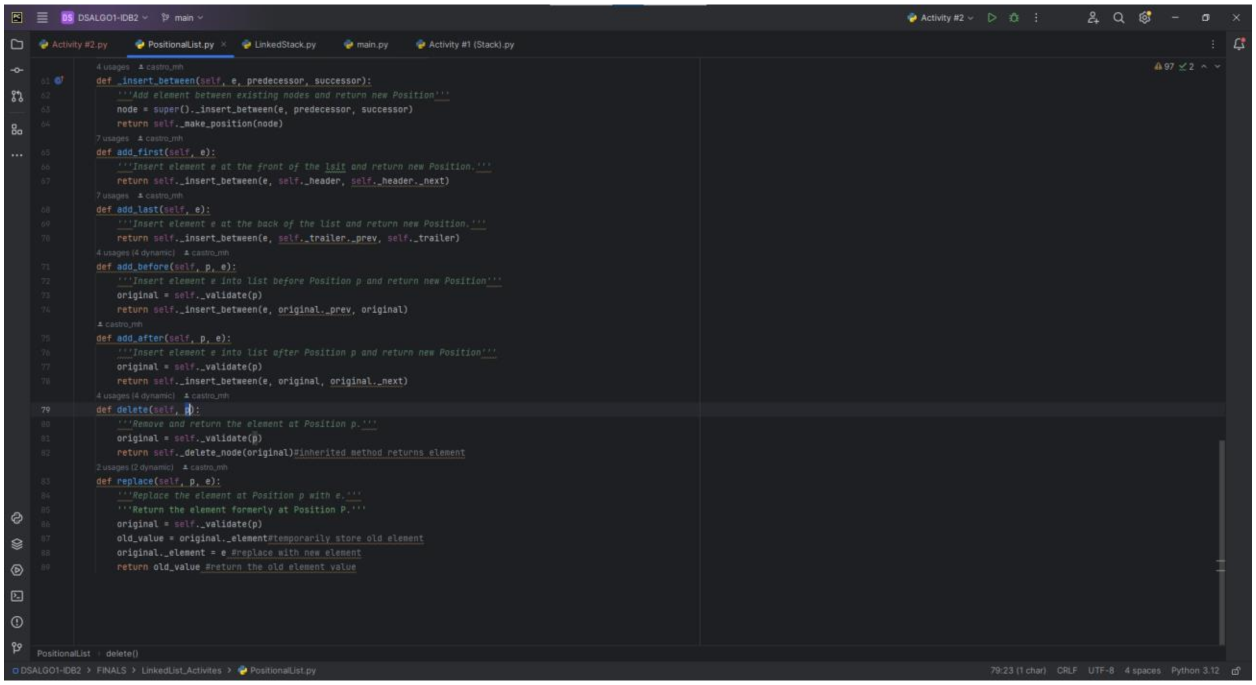
CODE:



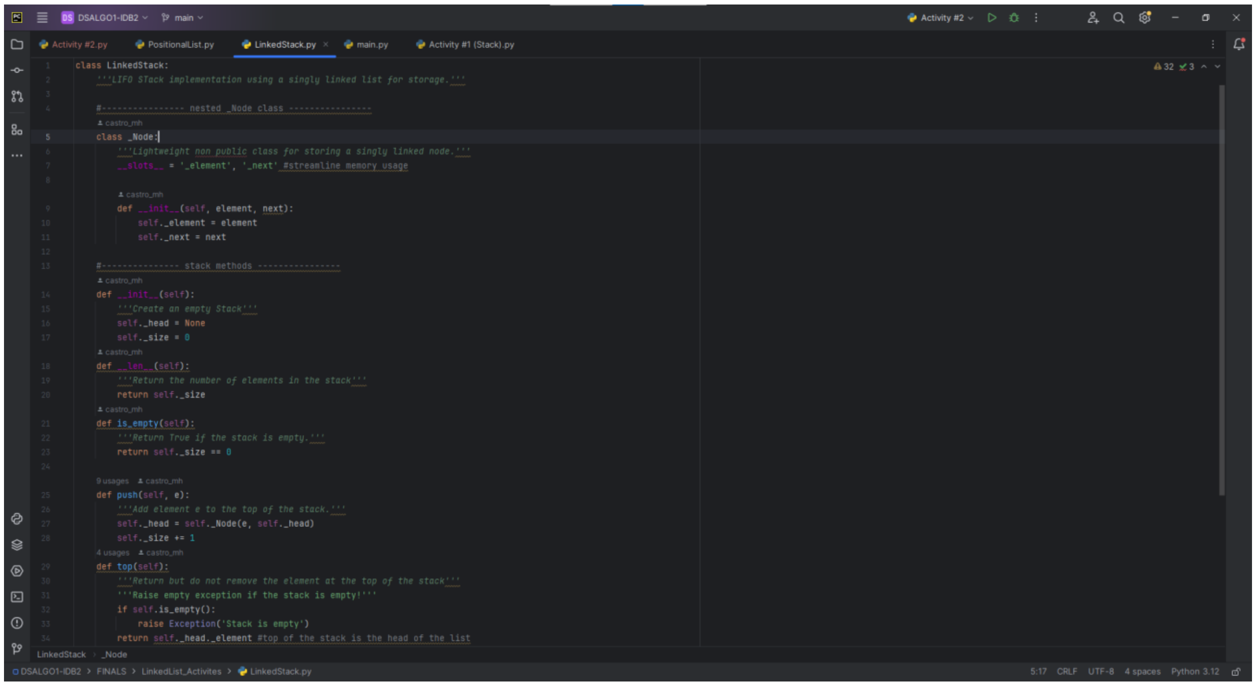
```python
from DoublyLinkedBase import _DoublyLinkedBase
class PositionalList(_DoublyLinkedBase):
    '''A sequential container of elements allowing positional access.'''
    #---Positional list class
    class Position:
        '''An abstraction representing the location of a single element.'''
        def __init__(self, container, node):
            '''Constructor should not be invoked by the user.'''
            self._container = container
            self._node = node
        def element(self):
            '''Return the element stored at this Position'''
            return self._node._element
        def __eq__(self, other):
            '''Return True if other is a Position representing the same location.'''
            return type(other) is type(self) and other._node is self._node
        def __ne__(self,other):
            '''Return True if other does not represent the same location.'''
            return not  (self == other) #opposite of __eq__

    #-- utility method
    def _validate(self, p):
        '''Return postiion's node or raise appropriate error if invalid'''
        if not isinstance(p, self.Position):
            raise TypeError('p must be proper Position type')
        if p._container is not self:
            raise ValueError('p does not belong to this container')
        if p._node._next is None:#convention for deprecated nodes
            raise ValueError('p is no longer valid')
        return p._node
    #-- utility method
    def _make_position(self, node):
        '''Return Position instance for given node (or None if sentinel).'''
```



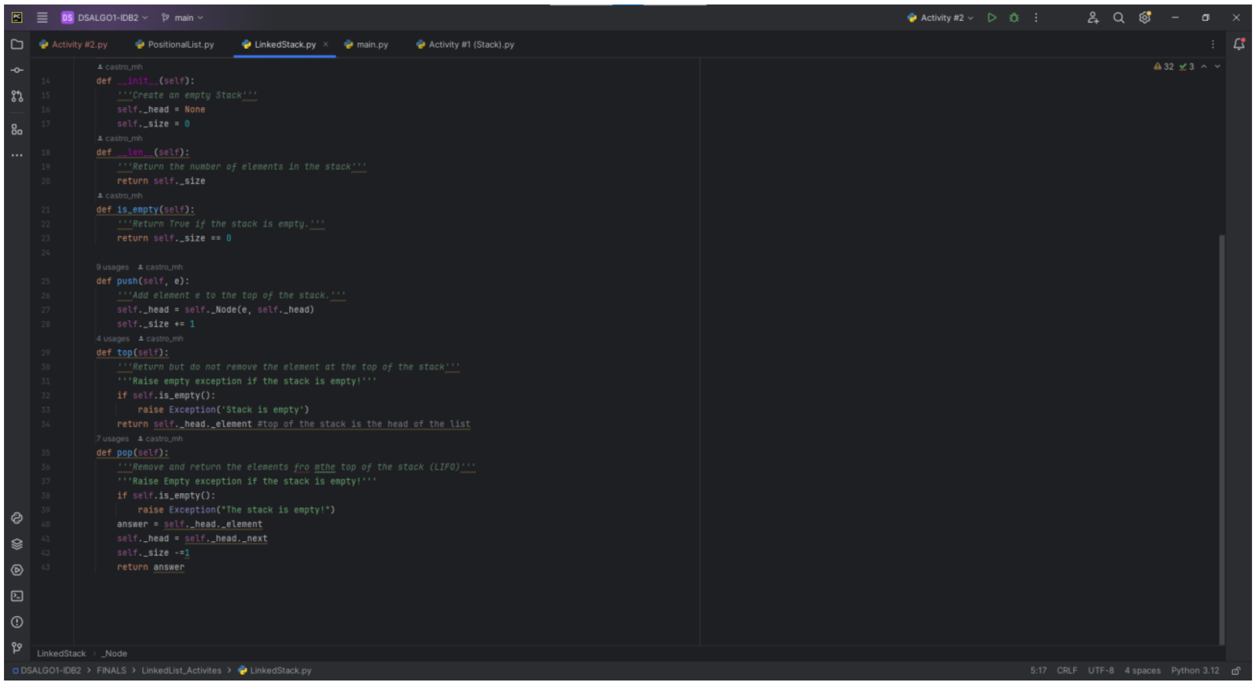
```python
        '''Return Position instance for given node (or None if sentinel).'''
        if node is self._header or node is self._trailer:
            return None #boudnary violation
        else:
            return self.Position(self, node) #legitimate position
    #-- accessors
    def first(self):
        '''Return the first Position in the list (or None if list is empty.)'''
        return self._make_position(self._header._next)
    def last(self):
        '''Return the last Position in the list (or None if list is empty)'''
        return self._make_position(self._trailer._prev)
    def before(self, p):
        '''Return the Position just before Position P (or None if p is first)'''
        node = self._validate(p)
        return self._make_position(node._prev)
    def after(self, p):
        '''Return the Position just after Position p (or None if p is last.)'''
        node = self._validate(p)
        return self._make_position(node._next)
    def __iter__(self):
        '''Generate forward iteration of the elements of the list'''
        cursor = self.first()
        while cursor is not None:
            yield cursor.element()
            cursor = self.after(cursor)
    #--mutators
    #override inherited version to return Position, rather than Node
    def _insert_between(self, e, predecessor, successor):
        '''Add element between existing nodes and return new Position'''
        node = super()._insert_between(e, predecessor, successor)
        return self._make_position(node)
    def add_first(self, e):
        '''Insert element e at the front of the lsit and return new Position.'''
```


DSALGO1-IDB2
main
Activity #2.py
PositionalList.py
LinkedStack.py
main.py
Activity #1 (Stack).py
Activity #2
PositionalList › delete()
DSALGO1-IDB2 › FINALS › LinkedList_Activites › PositionalList.py
79:23 (1 char) CRLF UTF-8 4 spaces Python 3.12

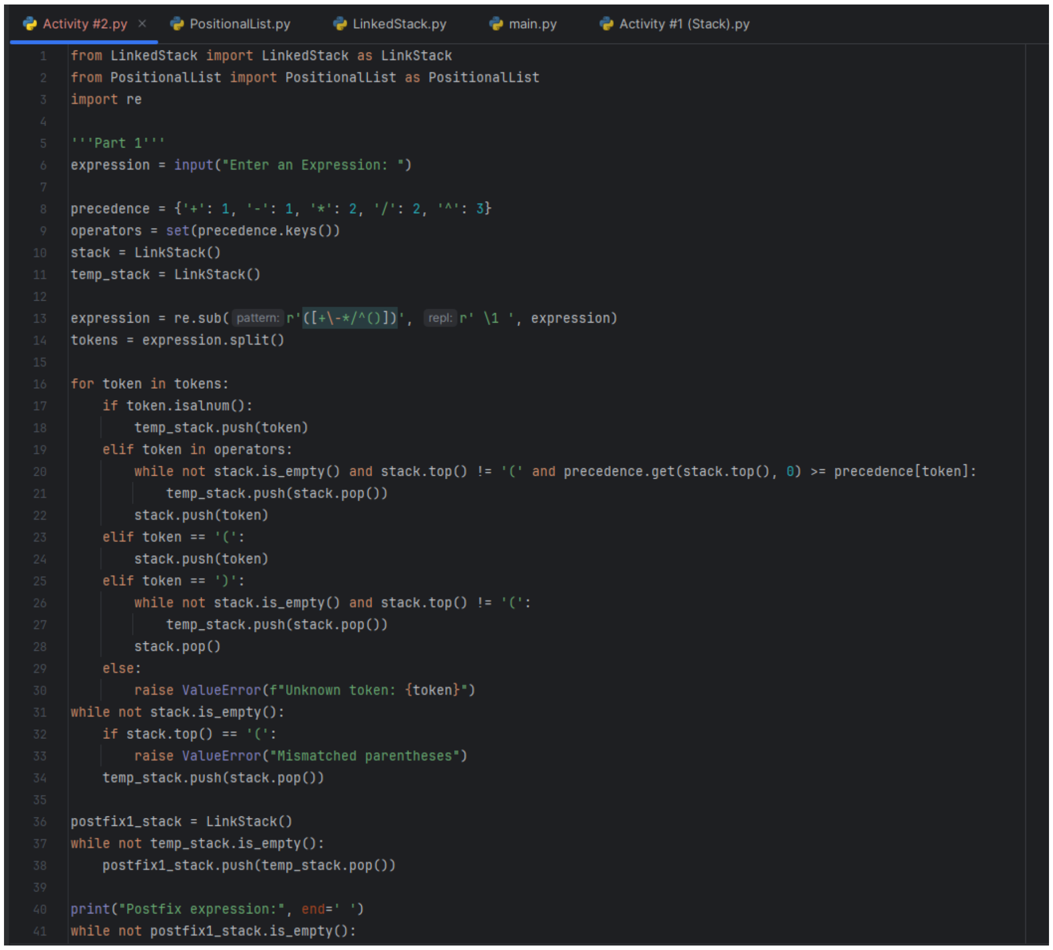
```python
    def _insert_between(self, e, predecessor, successor):
        '''Add element between existing nodes and return new Position'''
        node = super()._insert_between(e, predecessor, successor)
        return self._make_position(node)
    def add_first(self, e):
        '''Insert element e at the front of the lsit and return new Position.'''
        return self._insert_between(e, self._header, self._header._next)
    def add_last(self, e):
        '''Insert element e at the back of the list and return new Position.'''
        return self._insert_between(e, self._trailer._prev, self._trailer)
    def add_before(self, p, e):
        '''Insert element e into list before Position p and return new Position'''
        original = self._validate(p)
        return self._insert_between(e, original._prev, original)
    def add_after(self, p, e):
        '''Insert element e into list after Position p and return new Position'''
        original = self._validate(p)
        return self._insert_between(e, original, original._next)
    def delete(self, p):
        '''Remove and return the element at Position p.'''
        original = self._validate(p)
        return self._delete_node(original)#inherited method returns element
    def replace(self, p, e):
        '''Replace the element at Position p with e.'''
        '''Return the element formerly at Position P.'''
        original = self._validate(p)
        old_value = original._element#temporarily store old element
        original._element = e #replace with new element
        return old_value #return the old element value
```



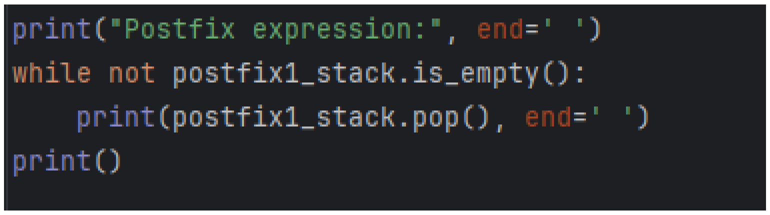
```python
class LinkedStack:
    '''LIFO STack implementation using a singly linked list for storage.'''

    #--------------- nested _Node class ---------------
    class _Node:
        '''Lightweight non public class for storing a singly linked node.'''
        __slots__ = '_element', '_next' #streamline memory usage

        def __init__(self, element, next):
            self._element = element
            self._next = next

    #-------------- stack methods ---------------
    def __init__(self):
        '''Create an empty Stack'''
        self._head = None
        self._size = 0
    def __len__(self):
        '''Return the number of elements in the stack'''
        return self._size
    def is_empty(self):
        '''Return True if the stack is empty.'''
        return self._size == 0

    def push(self, e):
        '''Add element e to the top of the stack.'''
        self._head = self._Node(e, self._head)
        self._size += 1
    def top(self):
        '''Return but do not remove the element at the top of the stack'''
        '''Raise empty exception if the stack is empty!'''
        if self.is_empty():
            raise Exception('Stack is empty')
        return self._head._element #top of the stack is the head of the list
```



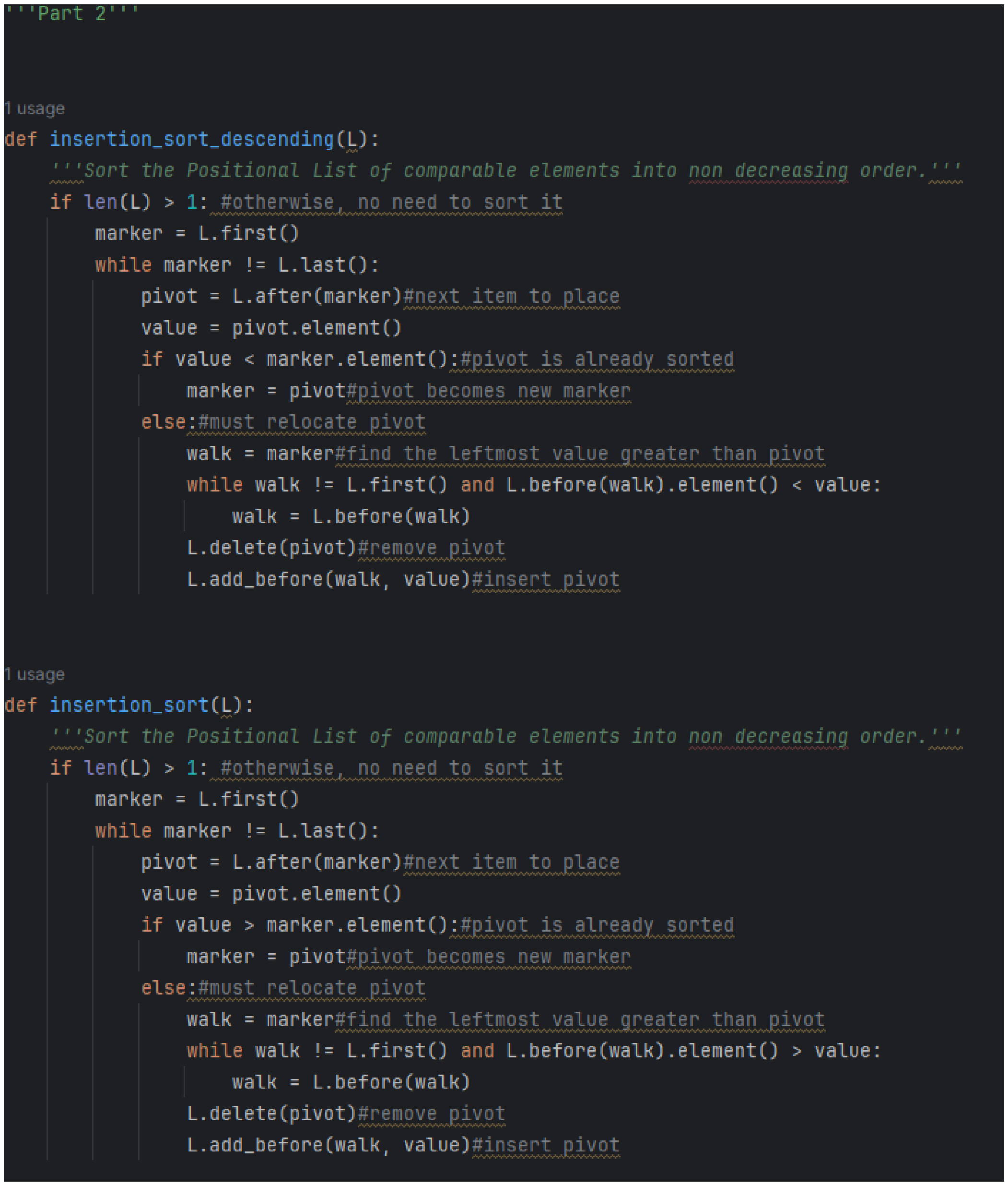
```python
    def __init__(self):
        '''Create an empty Stack'''
        self._head = None
        self._size = 0

    def __len__(self):
        '''Return the number of elements in the stack'''
        return self._size

    def is_empty(self):
        '''Return True if the stack is empty.'''
        return self._size == 0

    def push(self, e):
        '''Add element e to the top of the stack.'''
        self._head = self._Node(e, self._head)
        self._size += 1

    def top(self):
        '''Return but do not remove the element at the top of the stack'''
        '''Raise empty exception if the stack is empty!'''
        if self.is_empty():
            raise Exception('Stack is empty')
        return self._head._element #top of the stack is the head of the list

    def pop(self):
        '''Remove and return the elements fro mthe top of the stack (LIFO)'''
        '''Raise Empty exception if the stack is empty!'''
        if self.is_empty():
            raise Exception("The stack is empty!")
        answer = self._head._element
        self._head = self._head._next
        self._size -=1
        return answer
```



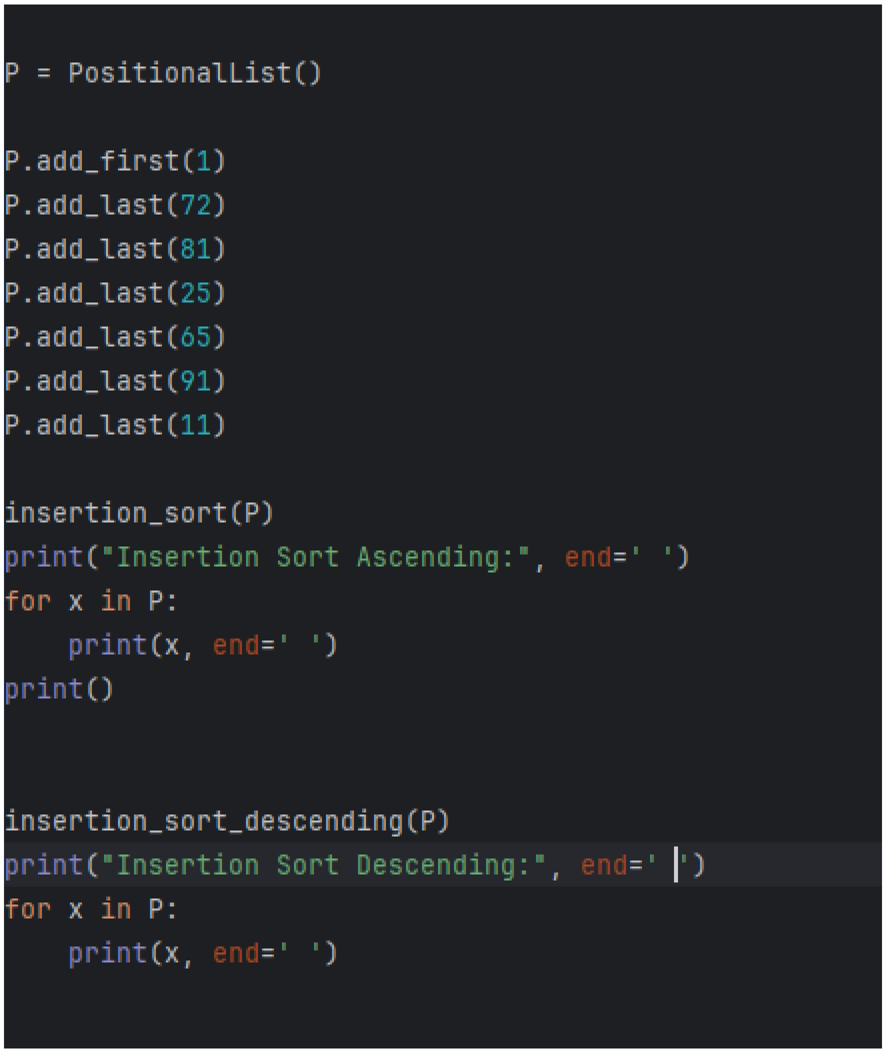
```python
from LinkedStack import LinkedStack as LinkStack
from PositionalList import PositionalList as PositionalList
import re

'''Part 1'''
expression = input("Enter an Expression: ")

precedence = {'+': 1, '-': 1, '*': 2, '/': 2, '^': 3}
operators = set(precedence.keys())
stack = LinkStack()
temp_stack = LinkStack()

expression = re.sub(r'([+\-*/^()])', r' \1 ', expression)
tokens = expression.split()

for token in tokens:
    if token.isalnum():
        temp_stack.push(token)
    elif token in operators:
        while not stack.is_empty() and stack.top() != '(' and precedence.get(stack.top(), 0) >= precedence[token]:
            temp_stack.push(stack.pop())
        stack.push(token)
    elif token == '(':
        stack.push(token)
    elif token == ')':
        while not stack.is_empty() and stack.top() != '(':
            temp_stack.push(stack.pop())
        stack.pop()
    else:
        raise ValueError(f"Unknown token: {token}")
while not stack.is_empty():
    if stack.top() == '(':
        raise ValueError("Mismatched parentheses")
    temp_stack.push(stack.pop())

postfix1_stack = LinkStack()
while not temp_stack.is_empty():
    postfix1_stack.push(temp_stack.pop())

print("Postfix expression:", end=' ')
while not postfix1_stack.is_empty():
```



```python
print("Postfix expression:", end=' ')
while not postfix1_stack.is_empty():
    print(postfix1_stack.pop(), end=' ')
print()
```

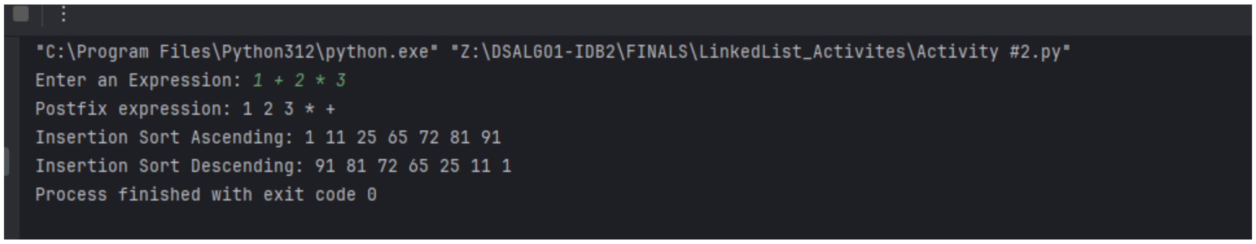


```python
'''Part 2'''


1 usage
def insertion_sort_descending(L):
    '''Sort the Positional List of comparable elements into non decreasing order.'''
    if len(L) > 1: #otherwise, no need to sort it
        marker = L.first()
        while marker != L.last():
            pivot = L.after(marker)#next item to place
            value = pivot.element()
            if value < marker.element():#pivot is already sorted
                marker = pivot#pivot becomes new marker
            else:#must relocate pivot
                walk = marker#find the leftmost value greater than pivot
                while walk != L.first() and L.before(walk).element() < value:
                    walk = L.before(walk)
                L.delete(pivot)#remove pivot
                L.add_before(walk, value)#insert pivot


1 usage
def insertion_sort(L):
    '''Sort the Positional List of comparable elements into non decreasing order.'''
    if len(L) > 1: #otherwise, no need to sort it
        marker = L.first()
        while marker != L.last():
            pivot = L.after(marker)#next item to place
            value = pivot.element()
            if value > marker.element():#pivot is already sorted
                marker = pivot#pivot becomes new marker
            else:#must relocate pivot
                walk = marker#find the leftmost value greater than pivot
                while walk != L.first() and L.before(walk).element() > value:
                    walk = L.before(walk)
                L.delete(pivot)#remove pivot
                L.add_before(walk, value)#insert pivot
```



```python
P = PositionalList()

P.add_first(1)
P.add_last(72)
P.add_last(81)
P.add_last(25)
P.add_last(65)
P.add_last(91)
P.add_last(11)

insertion_sort(P)
print("Insertion Sort Ascending:", end=' ')
for x in P:
    print(x, end=' ')
print()


insertion_sort_descending(P)
print("Insertion Sort Descending:", end=' ')
for x in P:
    print(x, end=' ')
```

OUTPUT FOR PROBLEM:



```
"C:\Program Files\Python312\python.exe" "Z:\DSALGO1-IDB2\FINALS\LinkedList_Activites\Activity #2.py"
Enter an Expression: 1 + 2 * 3
Postfix expression: 1 2 3 * +
Insertion Sort Ascending: 1 11 25 65 72 81 91
Insertion Sort Descending: 91 81 72 65 25 11 1
Process finished with exit code 0
```