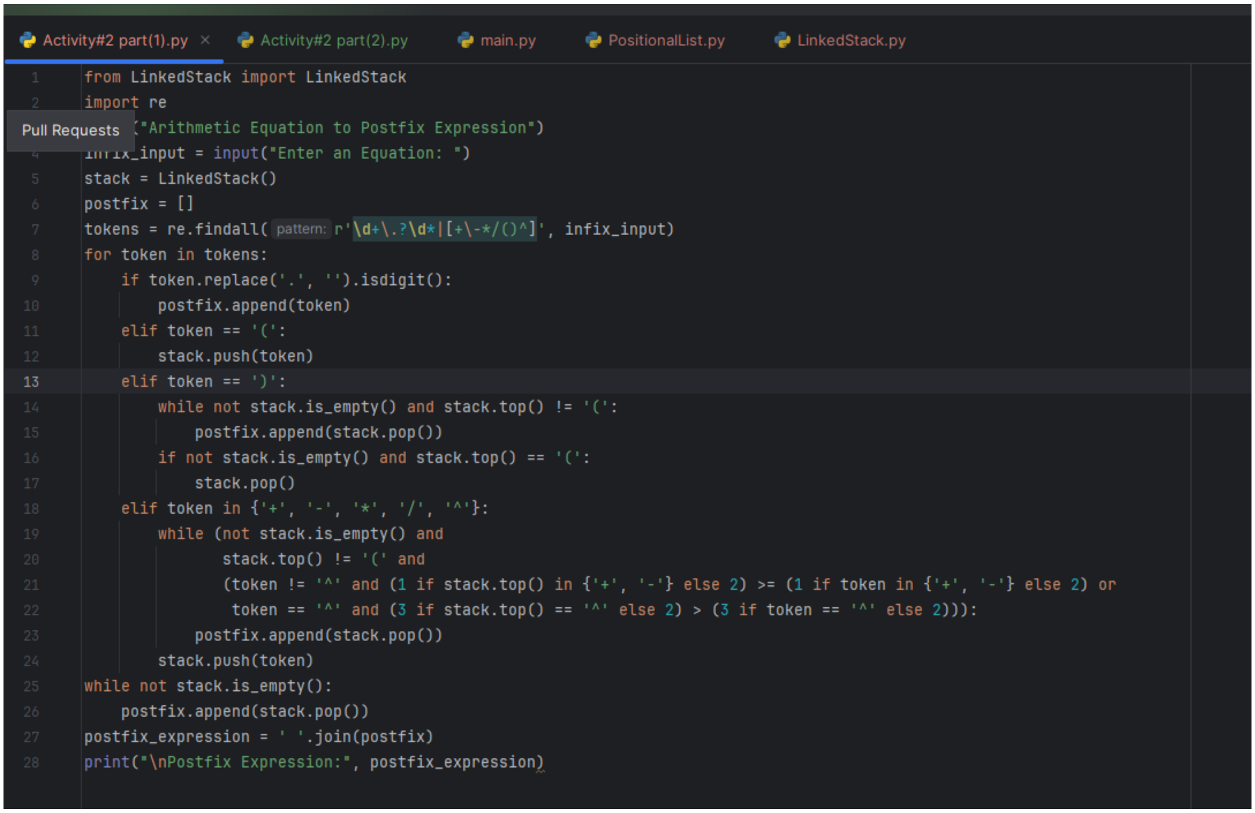
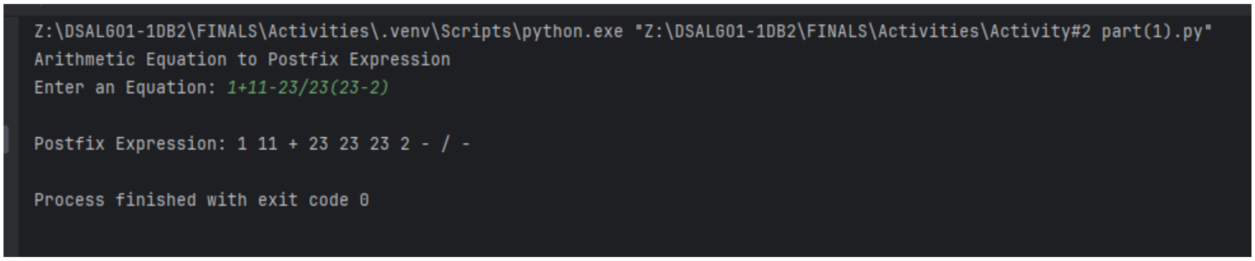
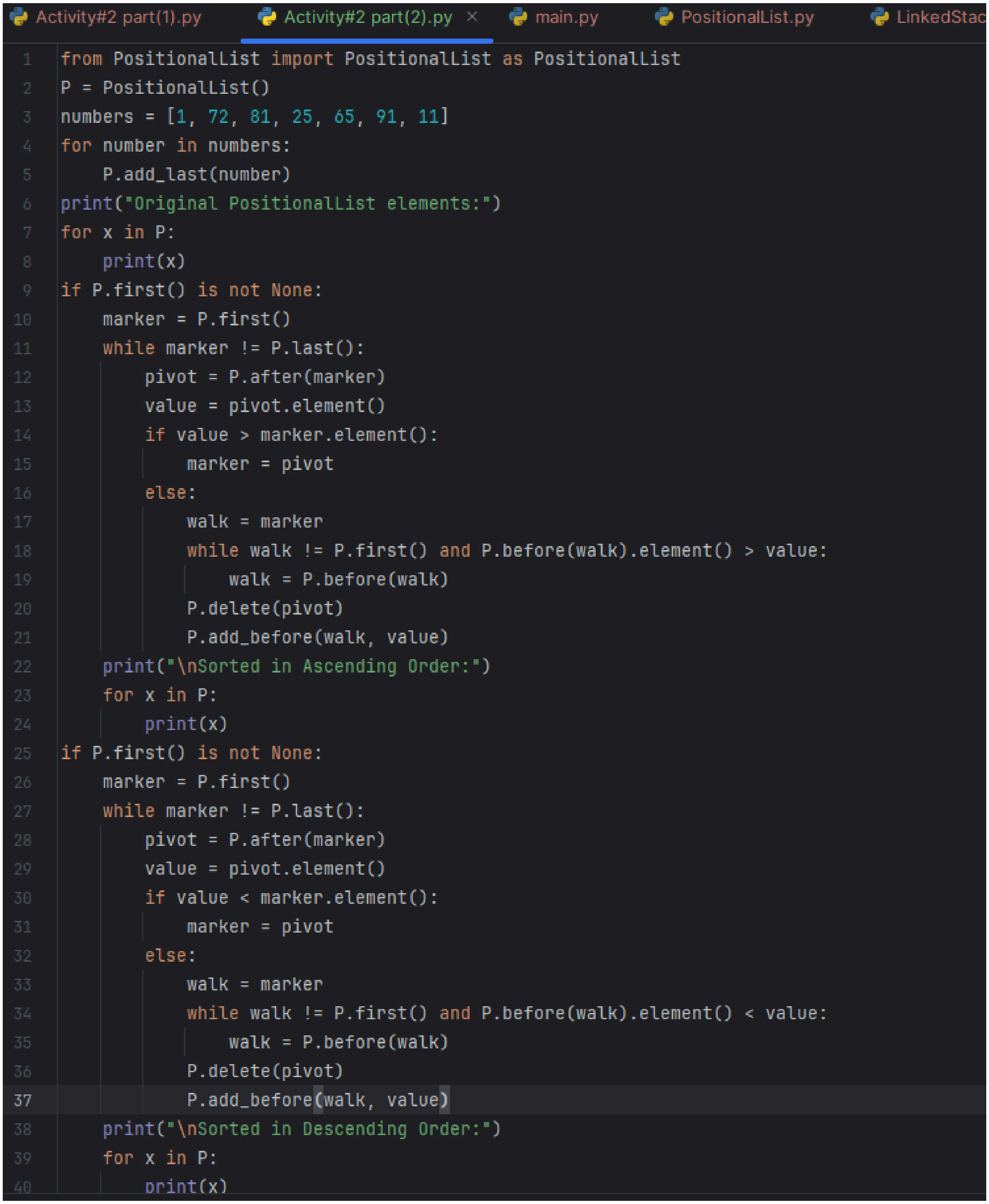


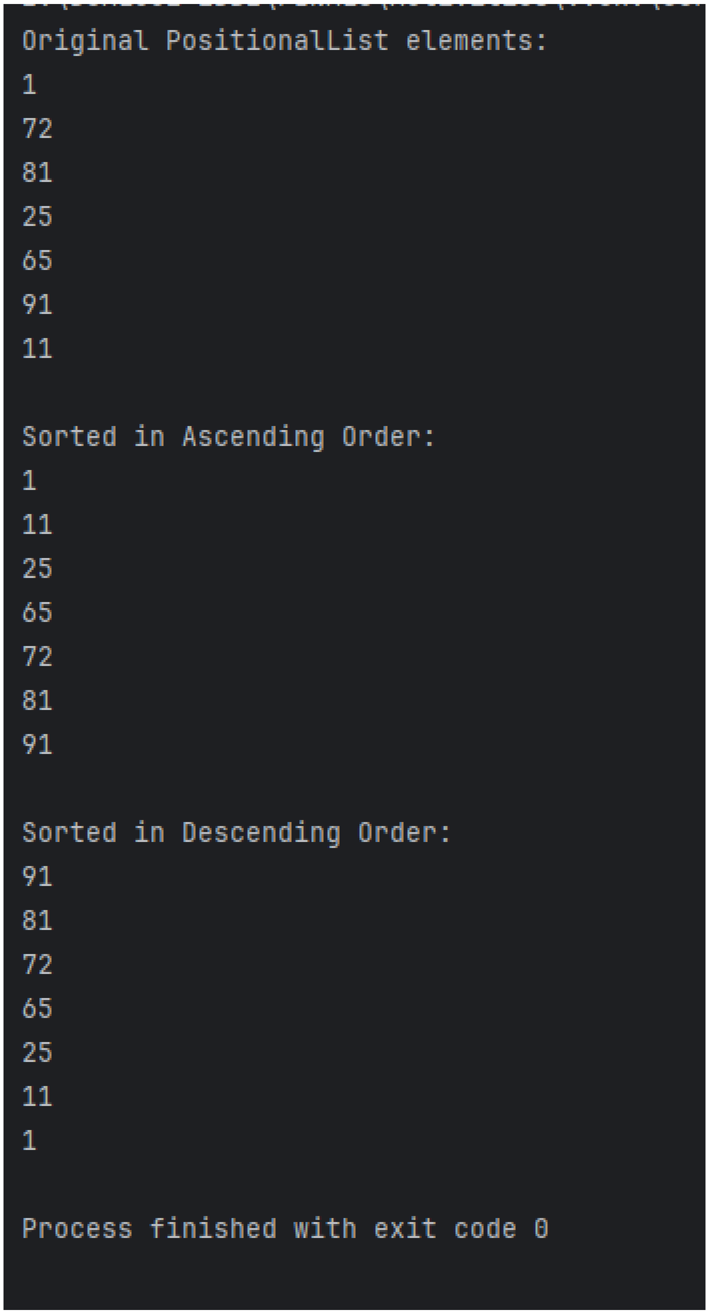
```python
from LinkedStack import LinkedStack
import re
("Arithmetic Equation to Postfix Expression")
infix_input = input("Enter an Equation: ")
stack = LinkedStack()
postfix = []
tokens = re.findall(r'\d+\.?\d*|[+\-*/()^]', infix_input)
for token in tokens:
    if token.replace('.', '').isdigit():
        postfix.append(token)
    elif token == '(':
        stack.push(token)
    elif token == ')':
        while not stack.is_empty() and stack.top() != '(':
            postfix.append(stack.pop())
        if not stack.is_empty() and stack.top() == '(':
            stack.pop()
    elif token in {'+', '-', '*', '/', '^'}:
        while (not stack.is_empty() and
               stack.top() != '(' and
               (token != '^' and (1 if stack.top() in {'+', '-'} else 2) >= (1 if token in {'+', '-'} else 2) or
                token == '^' and (3 if stack.top() == '^' else 2) > (3 if token == '^' else 2))):
            postfix.append(stack.pop())
        stack.push(token)
while not stack.is_empty():
    postfix.append(stack.pop())
postfix_expression = ' '.join(postfix)
print("\nPostfix Expression:", postfix_expression)
```



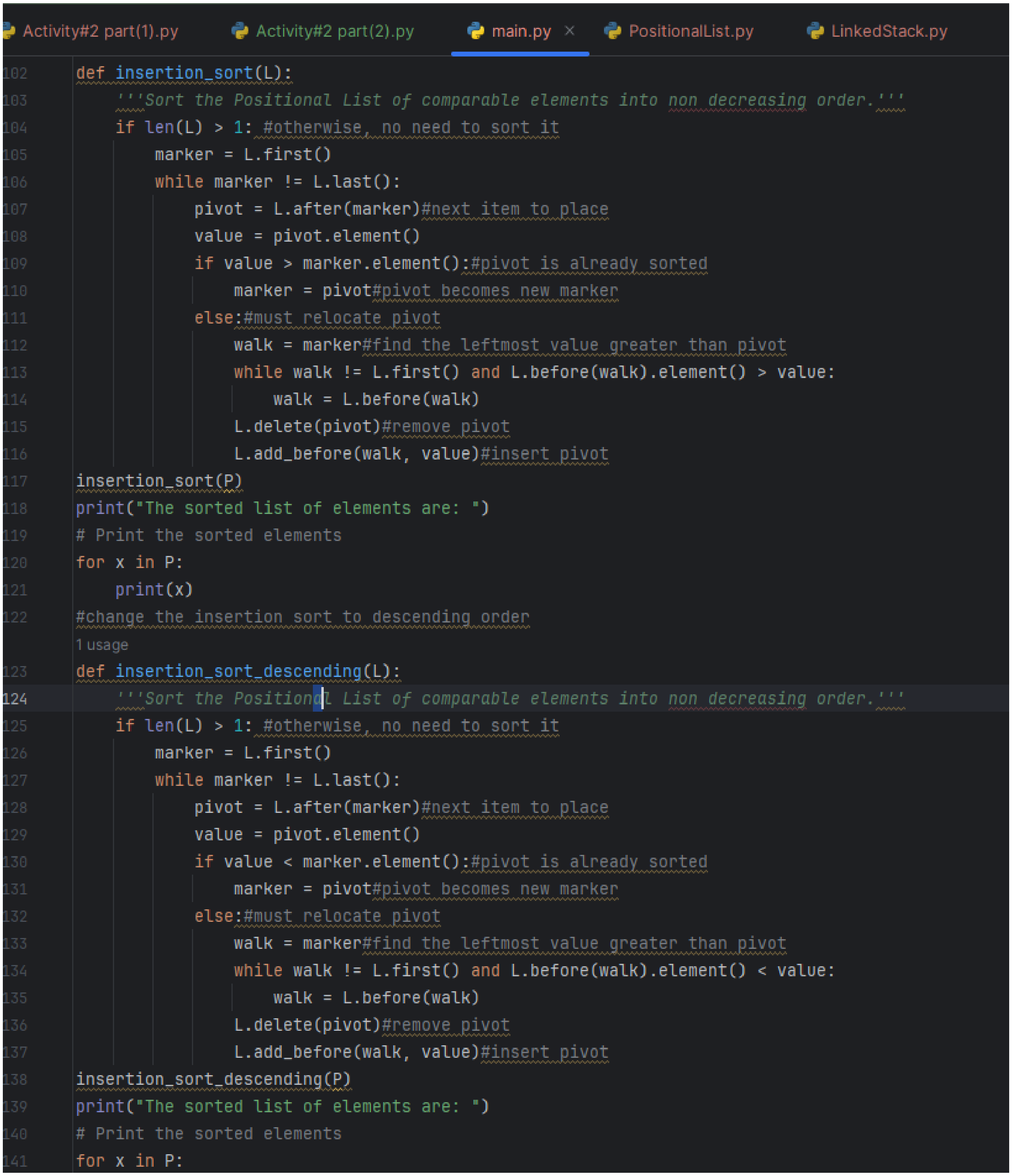
```
Z:\DSALG01-1DB2\FINALS\Activities\.venv\Scripts\python.exe "Z:\DSALG01-1DB2\FINALS\Activities\Activity#2 part(1).py"
Arithmetic Equation to Postfix Expression
Enter an Equation: 1+11-23/23(23-2)

Postfix Expression: 1 11 + 23 23 23 2 - / -

Process finished with exit code 0
```



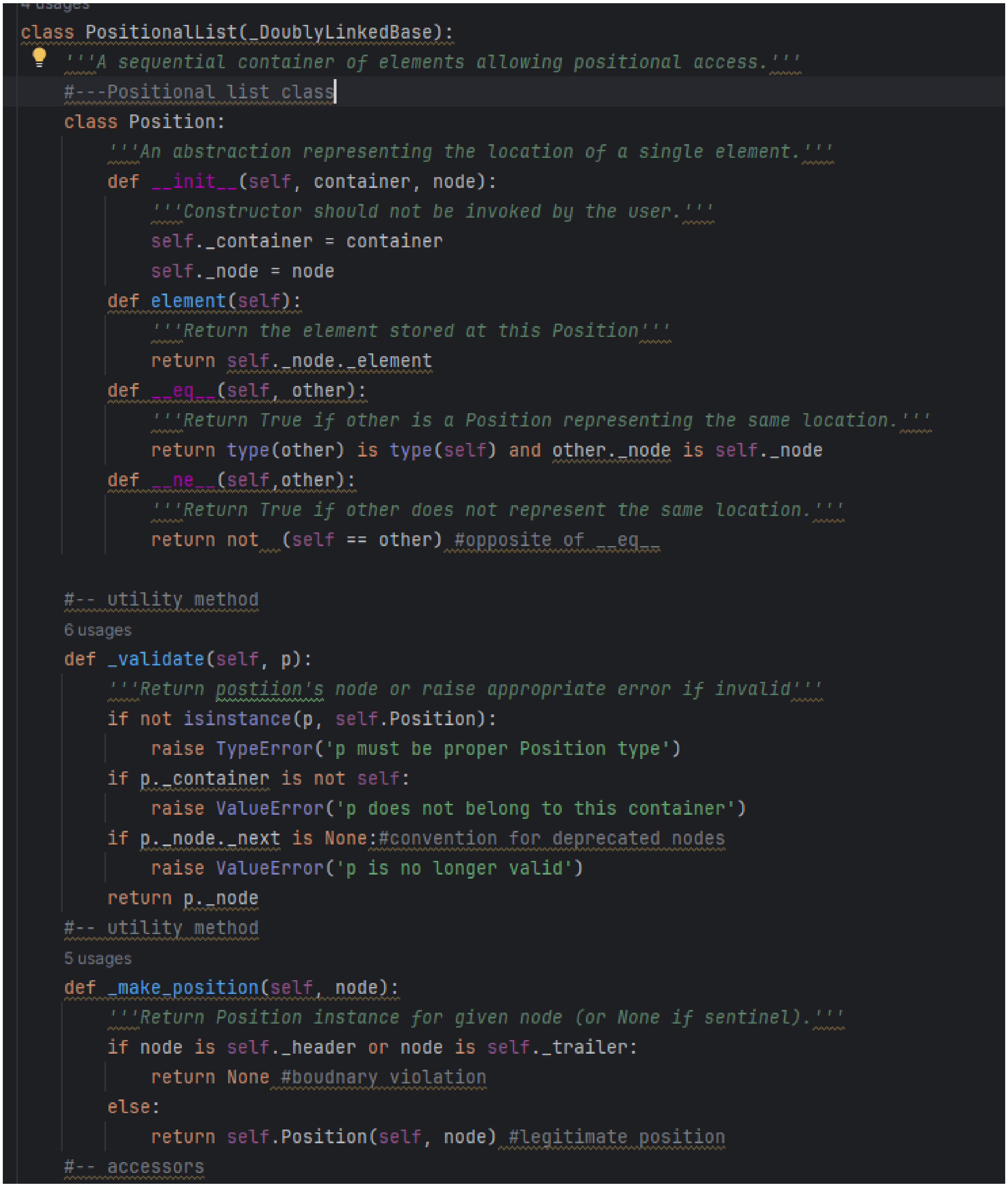
```python
from PositionalList import PositionalList as PositionalList
P = PositionalList()
numbers = [1, 72, 81, 25, 65, 91, 11]
for number in numbers:
    P.add_last(number)
print("Original PositionalList elements:")
for x in P:
    print(x)
if P.first() is not None:
    marker = P.first()
    while marker != P.last():
        pivot = P.after(marker)
        value = pivot.element()
        if value > marker.element():
            marker = pivot
        else:
            walk = marker
            while walk != P.first() and P.before(walk).element() > value:
                walk = P.before(walk)
            P.delete(pivot)
            P.add_before(walk, value)
    print("\nSorted in Ascending Order:")
    for x in P:
        print(x)
if P.first() is not None:
    marker = P.first()
    while marker != P.last():
        pivot = P.after(marker)
        value = pivot.element()
        if value < marker.element():
            marker = pivot
        else:
            walk = marker
            while walk != P.first() and P.before(walk).element() < value:
                walk = P.before(walk)
            P.delete(pivot)
            P.add_before(walk, value)
    print("\nSorted in Descending Order:")
    for x in P:
        print(x)
```



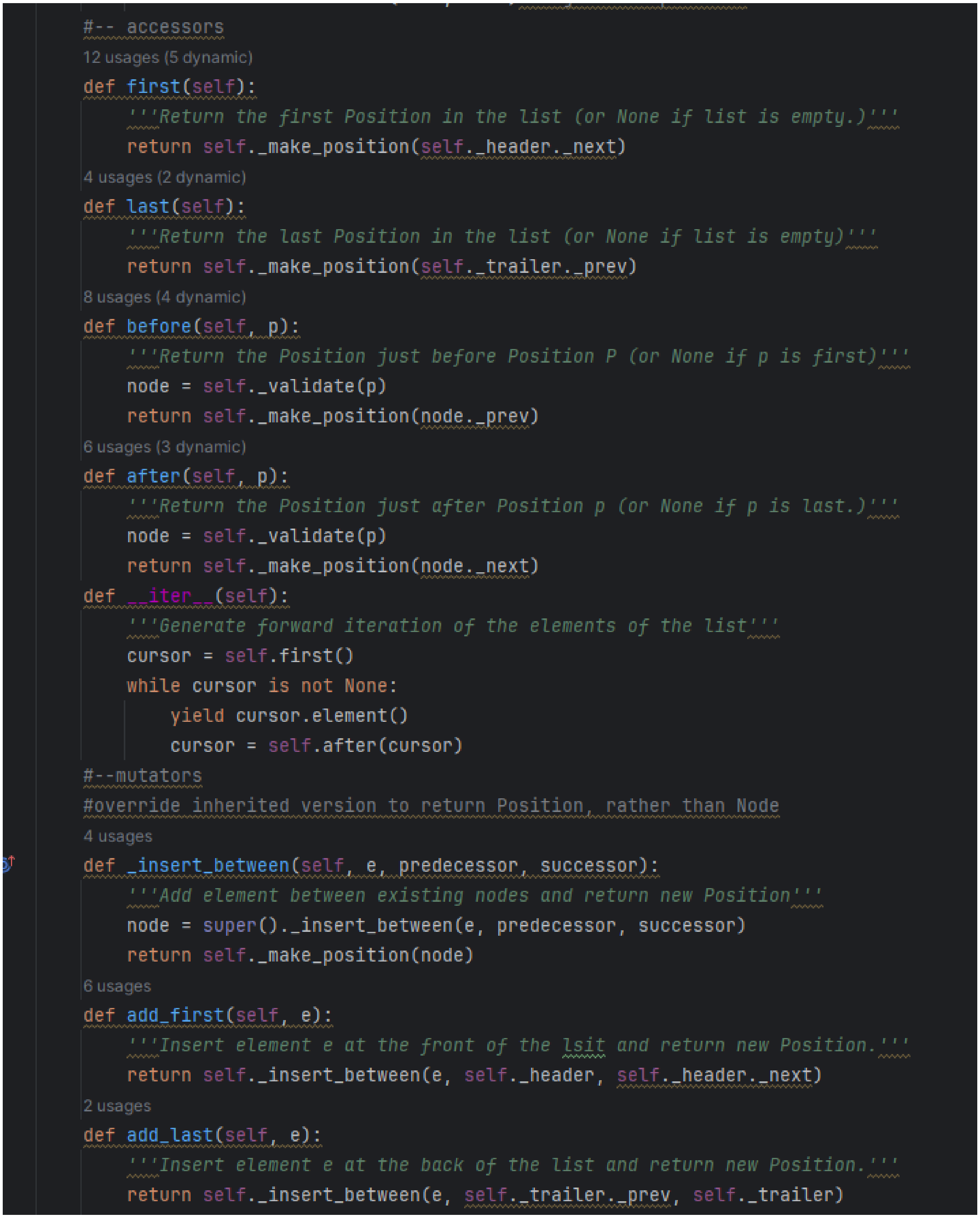
```
Original PositionalList elements:
1
72
81
25
65
91
11

Sorted in Ascending Order:
1
11
25
65
72
81
91

Sorted in Descending Order:
91
81
72
65
25
11
1

Process finished with exit code 0
```



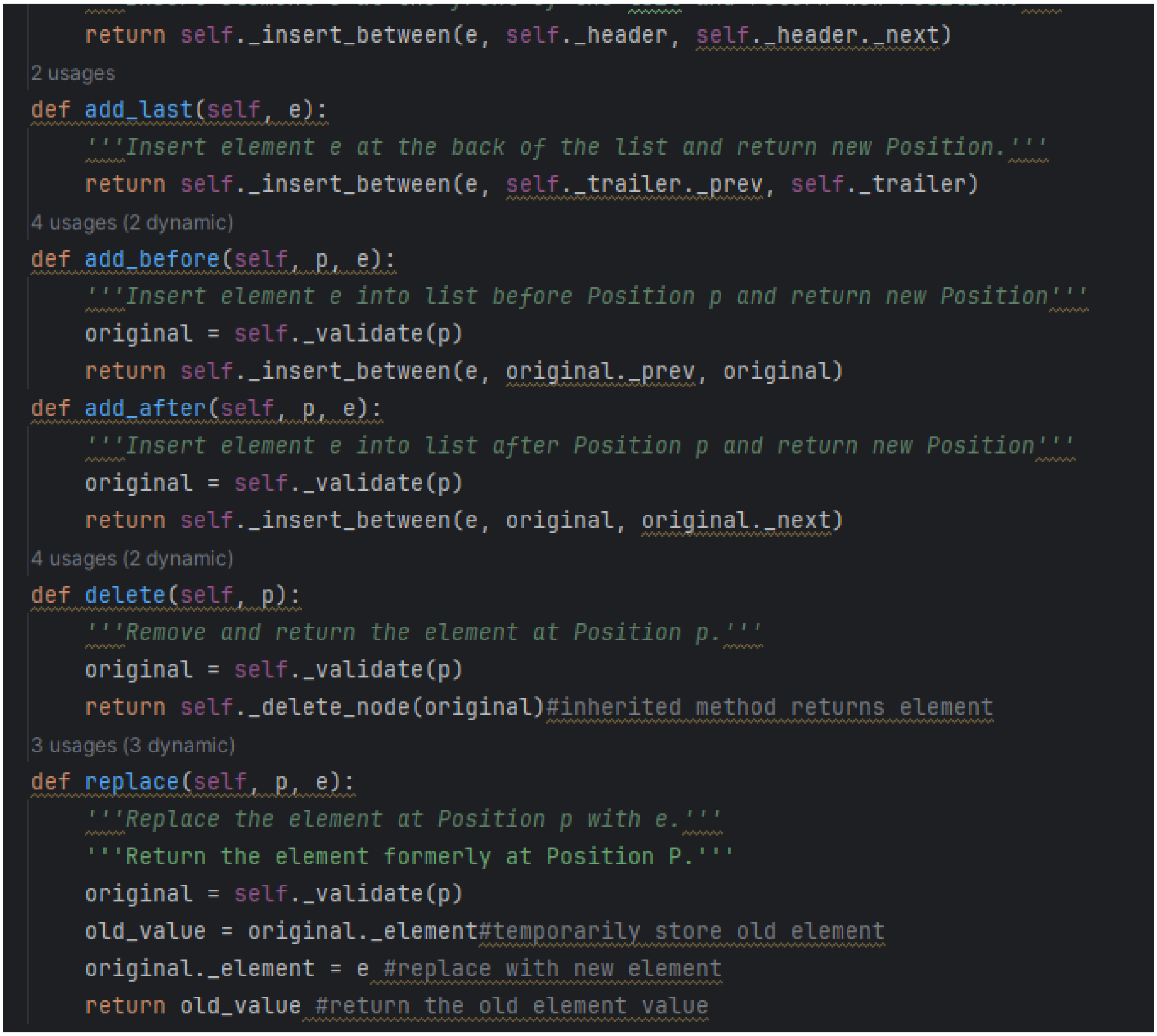
```python
def insertion_sort(L):
    '''Sort the Positional List of comparable elements into non decreasing order.'''
    if len(L) > 1: #otherwise, no need to sort it
        marker = L.first()
        while marker != L.last():
            pivot = L.after(marker)#next item to place
            value = 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
insertion_sort(P)
print("The sorted list of elements are: ")
# Print the sorted elements
for x in P:
    print(x)
#change the insertion sort to descending order
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
insertion_sort_descending(P)
print("The sorted list of elements are: ")
# Print the sorted elements
for x in P:
```



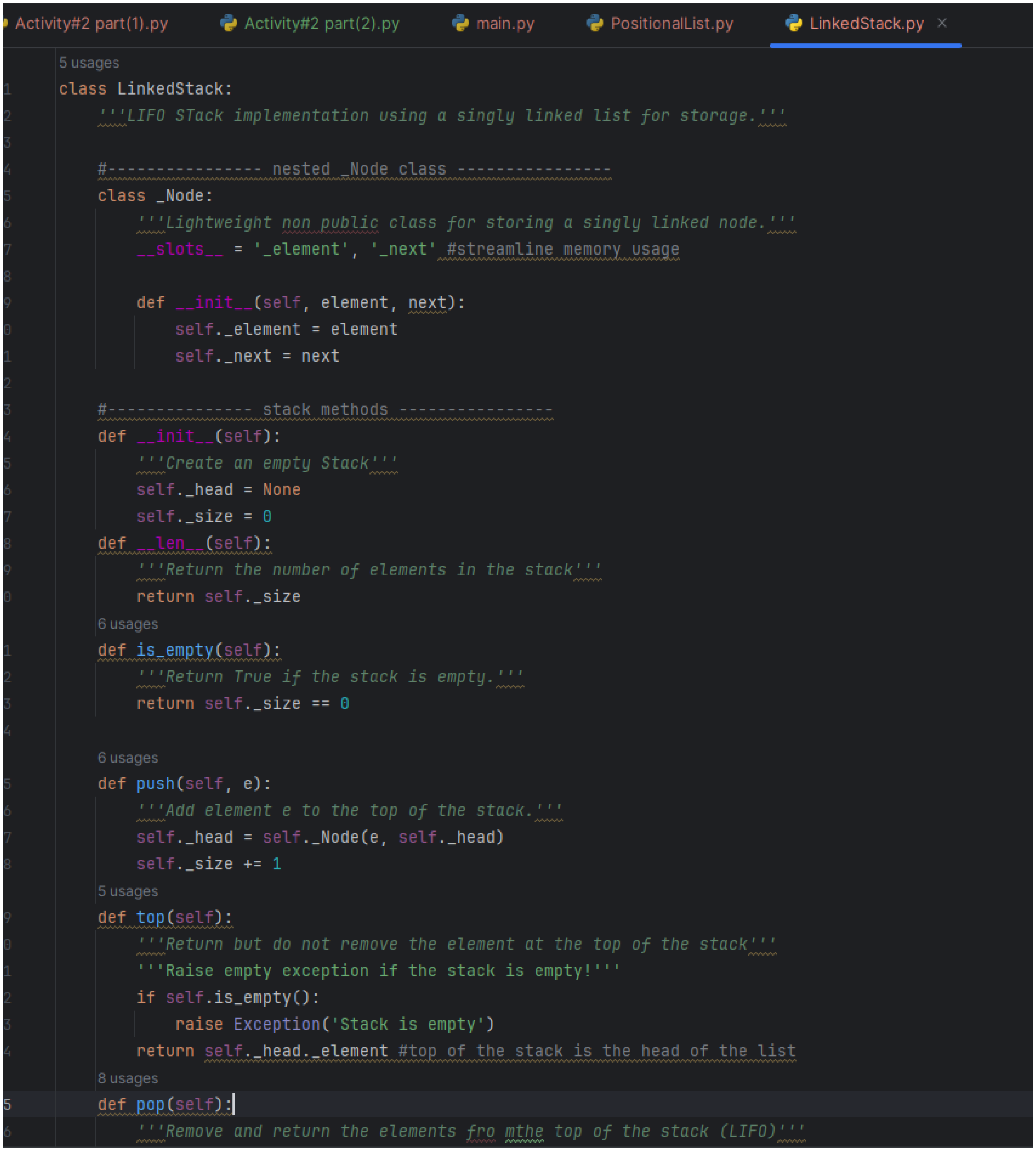
```
4 usages
class PositionalList(_DoublyLinkedBase):
    '''A sequential container of elements allowing positional access.'''
    #---Positional list class
    class Position:
        '''An abstraction representing the location of a single element.'''
        def __init__(self, container, node):
            '''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
    6 usages
    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
    5 usages
    def _make_position(self, node):
        '''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
```



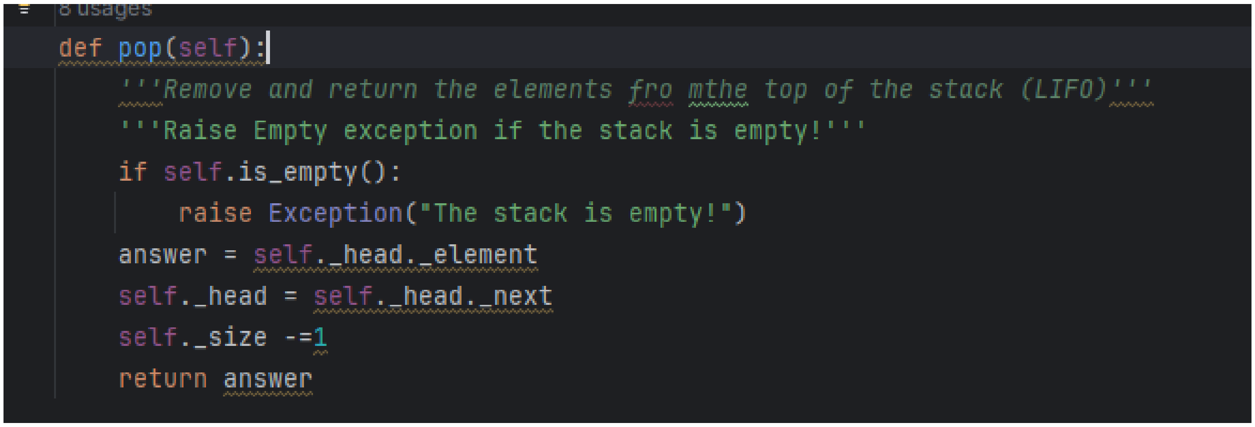
```
    #-- accessors
    12 usages (5 dynamic)
    def first(self):
        '''Return the first Position in the list (or None if list is empty.)'''
        return self._make_position(self._header._next)
    4 usages (2 dynamic)
    def last(self):
        '''Return the last Position in the list (or None if list is empty)'''
        return self._make_position(self._trailer._prev)
    8 usages (4 dynamic)
    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)
    6 usages (3 dynamic)
    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
    4 usages
    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)
    6 usages
    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)
    2 usages
    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)
```



```
    return self._insert_between(e, self._header, self._header._next)
2 usages
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)
4 usages (2 dynamic)
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)
4 usages (2 dynamic)
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
3 usages (3 dynamic)
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
    def pop(self):
        '''Remove and return the elements fro mthe top of the stack (LIFO)'''
```



```
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
```