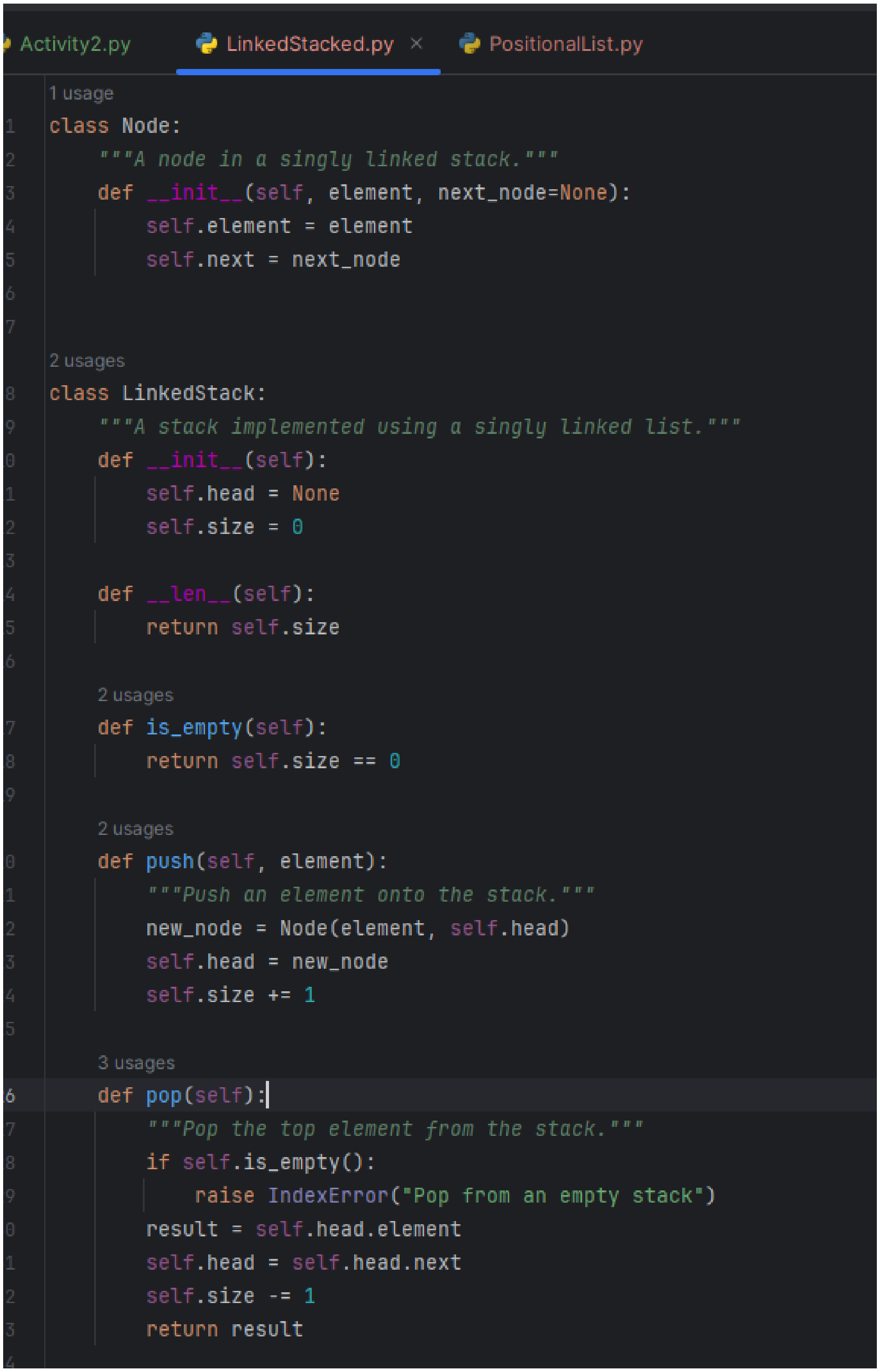
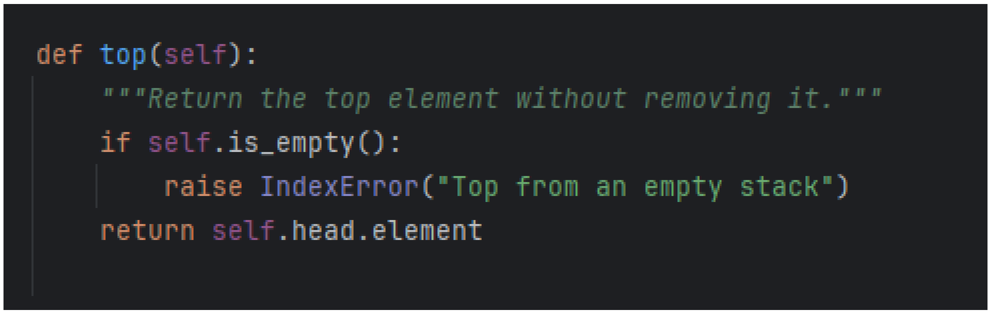
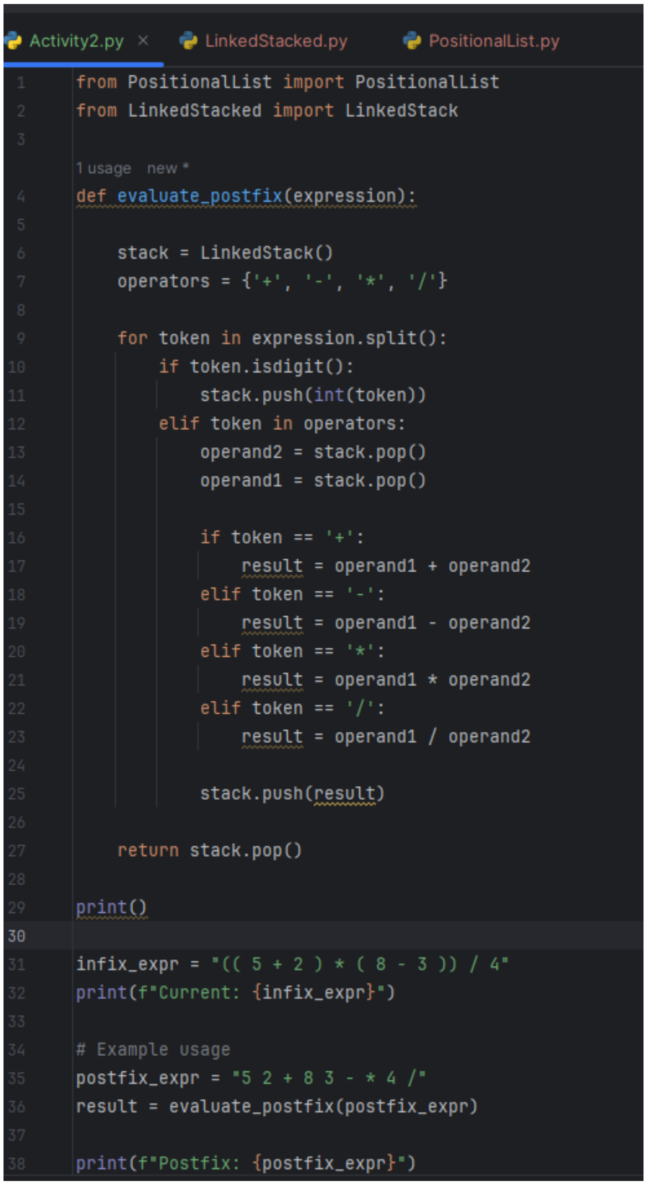
POSTFIX:



```python
class Node:
    """A node in a singly linked stack."""
    def __init__(self, element, next_node=None):
        self.element = element
        self.next = next_node


class LinkedStack:
    """A stack implemented using a singly linked list."""
    def __init__(self):
        self.head = None
        self.size = 0

    def __len__(self):
        return self.size

    def is_empty(self):
        return self.size == 0

    def push(self, element):
        """Push an element onto the stack."""
        new_node = Node(element, self.head)
        self.head = new_node
        self.size += 1

    def pop(self):
        """Pop the top element from the stack."""
        if self.is_empty():
            raise IndexError("Pop from an empty stack")
        result = self.head.element
        self.head = self.head.next
        self.size -= 1
        return result
```

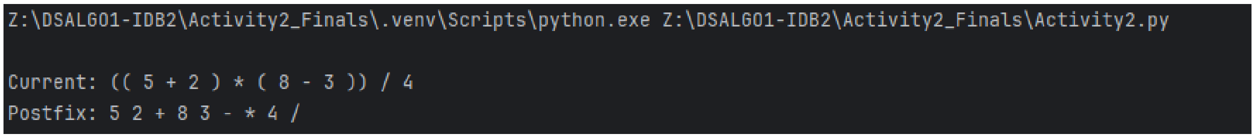


```python
def top(self):
    """Return the top element without removing it."""
    if self.is_empty():
        raise IndexError("Top from an empty stack")
    return self.head.element
```



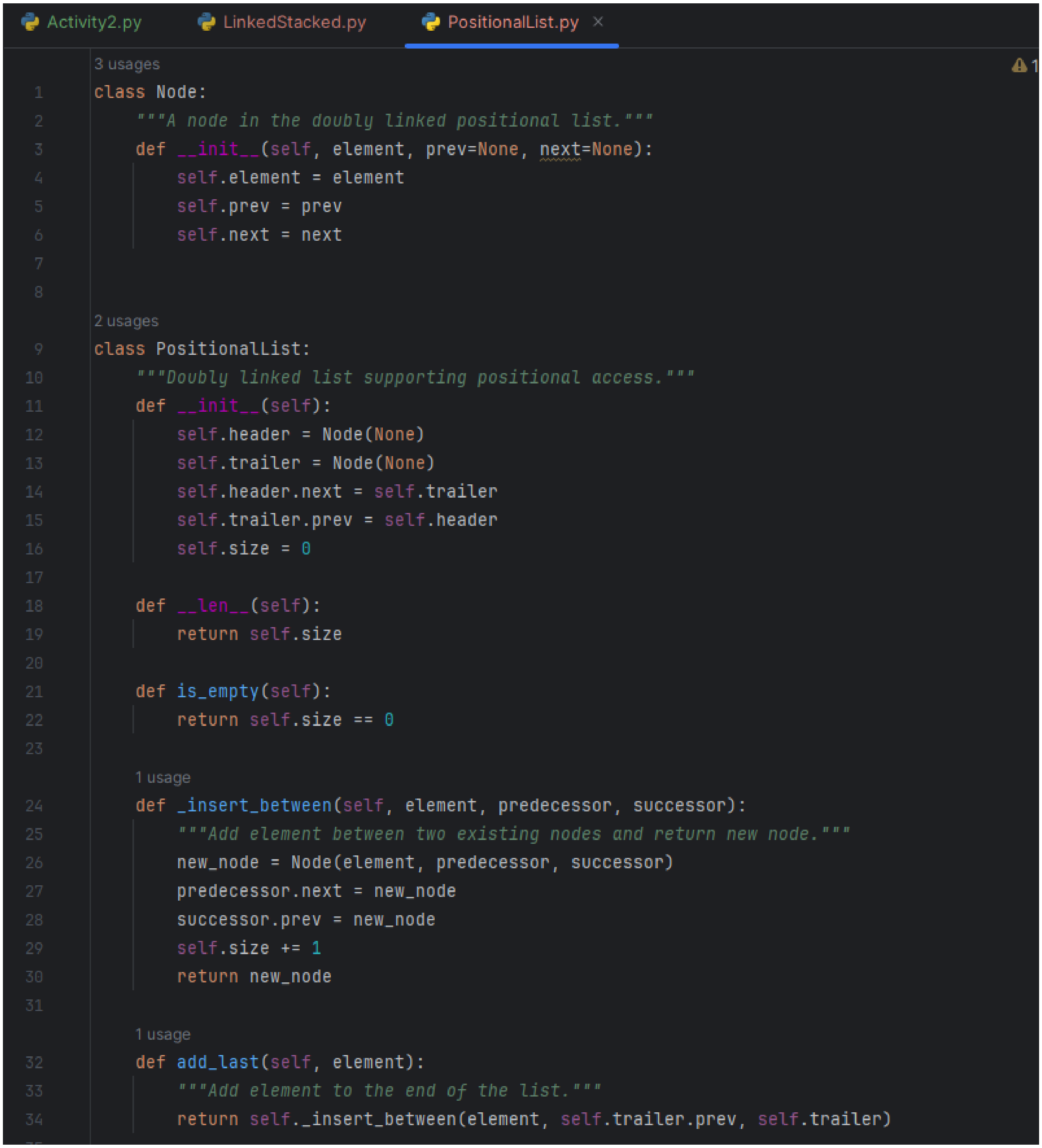
```python
from PositionalList import PositionalList
from LinkedStacked import LinkedStack

def evaluate_postfix(expression):

    stack = LinkedStack()
    operators = {'+', '-', '*', '/'}

    for token in expression.split():
        if token.isdigit():
            stack.push(int(token))
        elif token in operators:
            operand2 = stack.pop()
            operand1 = stack.pop()

            if token == '+':
                result = operand1 + operand2
            elif token == '-':
                result = operand1 - operand2
            elif token == '*':
                result = operand1 * operand2
            elif token == '/':
                result = operand1 / operand2

            stack.push(result)

    return stack.pop()

print()

infix_expr = "(( 5 + 2 ) * ( 8 - 3 )) / 4"
print(f"Current: {infix_expr}")

# Example usage
postfix_expr = "5 2 + 8 3 - * 4 /"
result = evaluate_postfix(postfix_expr)

print(f"Postfix: {postfix_expr}")
```

Output:

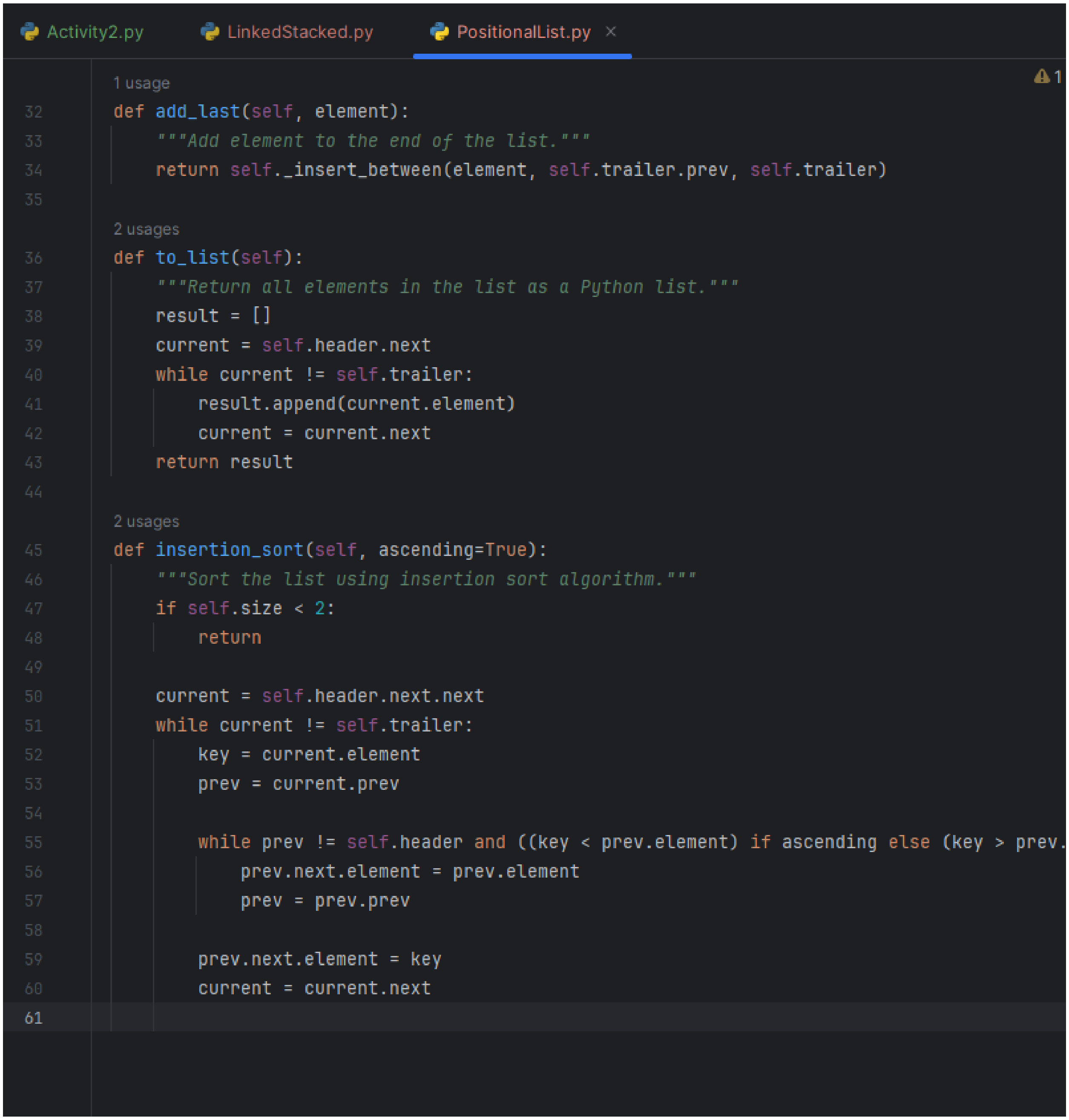


```
Z:\DSALG01-IDB2\Activity2_Finals\.venv\Scripts\python.exe Z:\DSALG01-IDB2\Activity2_Finals\Activity2.py

Current: (( 5 + 2 ) * ( 8 - 3 )) / 4
Postfix: 5 2 + 8 3 - * 4 /
```

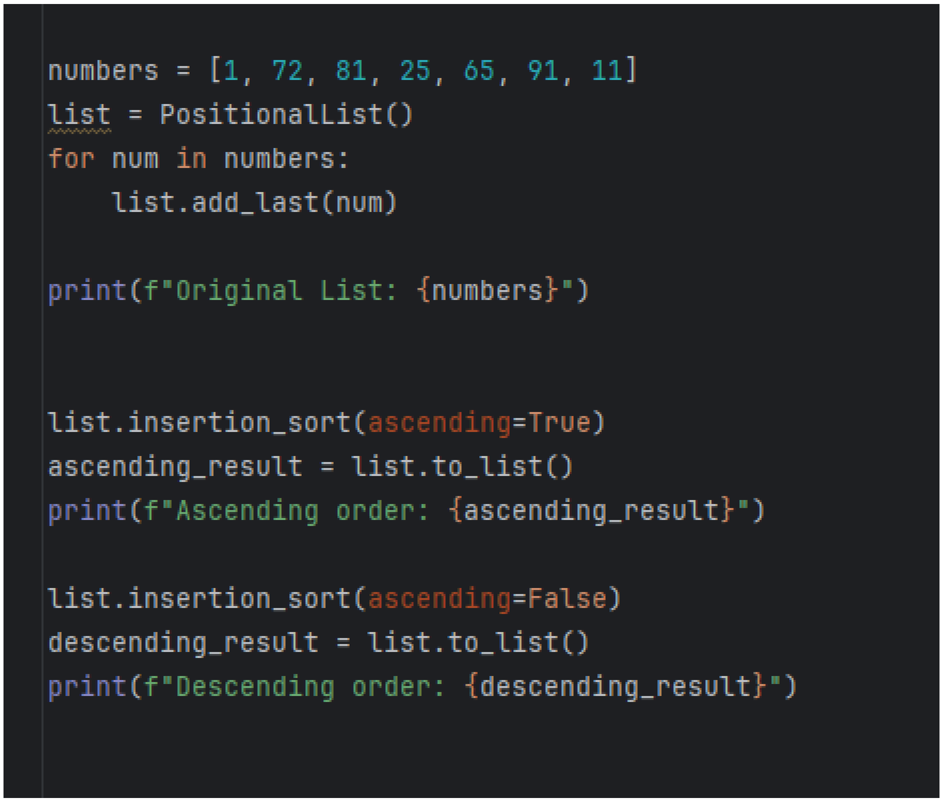
Insertion Sort:



```python
class Node:
    """A node in the doubly linked positional list."""
    def __init__(self, element, prev=None, next=None):
        self.element = element
        self.prev = prev
        self.next = next


class PositionalList:
    """Doubly linked list supporting positional access."""
    def __init__(self):
        self.header = Node(None)
        self.trailer = Node(None)
        self.header.next = self.trailer
        self.trailer.prev = self.header
        self.size = 0

    def __len__(self):
        return self.size

    def is_empty(self):
        return self.size == 0

    def _insert_between(self, element, predecessor, successor):
        """Add element between two existing nodes and return new node."""
        new_node = Node(element, predecessor, successor)
        predecessor.next = new_node
        successor.prev = new_node
        self.size += 1
        return new_node

    def add_last(self, element):
        """Add element to the end of the list."""
        return self._insert_between(element, self.trailer.prev, self.trailer)
```

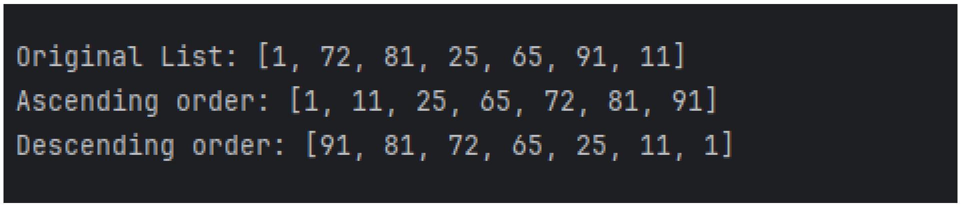


```
1 usage
def add_last(self, element):
    """Add element to the end of the list."""
    return self._insert_between(element, self.trailer.prev, self.trailer)

2 usages
def to_list(self):
    """Return all elements in the list as a Python list."""
    result = []
    current = self.header.next
    while current != self.trailer:
        result.append(current.element)
        current = current.next
    return result

2 usages
def insertion_sort(self, ascending=True):
    """Sort the list using insertion sort algorithm."""
    if self.size < 2:
        return

    current = self.header.next.next
    while current != self.trailer:
        key = current.element
        prev = current.prev

        while prev != self.header and ((key < prev.element) if ascending else (key > prev.
            prev.next.element = prev.element
            prev = prev.prev

        prev.next.element = key
        current = current.next

```



```python
numbers = [1, 72, 81, 25, 65, 91, 11]
list = PositionalList()
for num in numbers:
    list.add_last(num)

print(f"Original List: {numbers}")


list.insertion_sort(ascending=True)
ascending_result = list.to_list()
print(f"Ascending order: {ascending_result}")

list.insertion_sort(ascending=False)
descending_result = list.to_list()
print(f"Descending order: {descending_result}")
```

Output:



```
Original List: [1, 72, 81, 25, 65, 91, 11]
Ascending order: [1, 11, 25, 65, 72, 81, 91]
Descending order: [91, 81, 72, 65, 25, 11, 1]
```