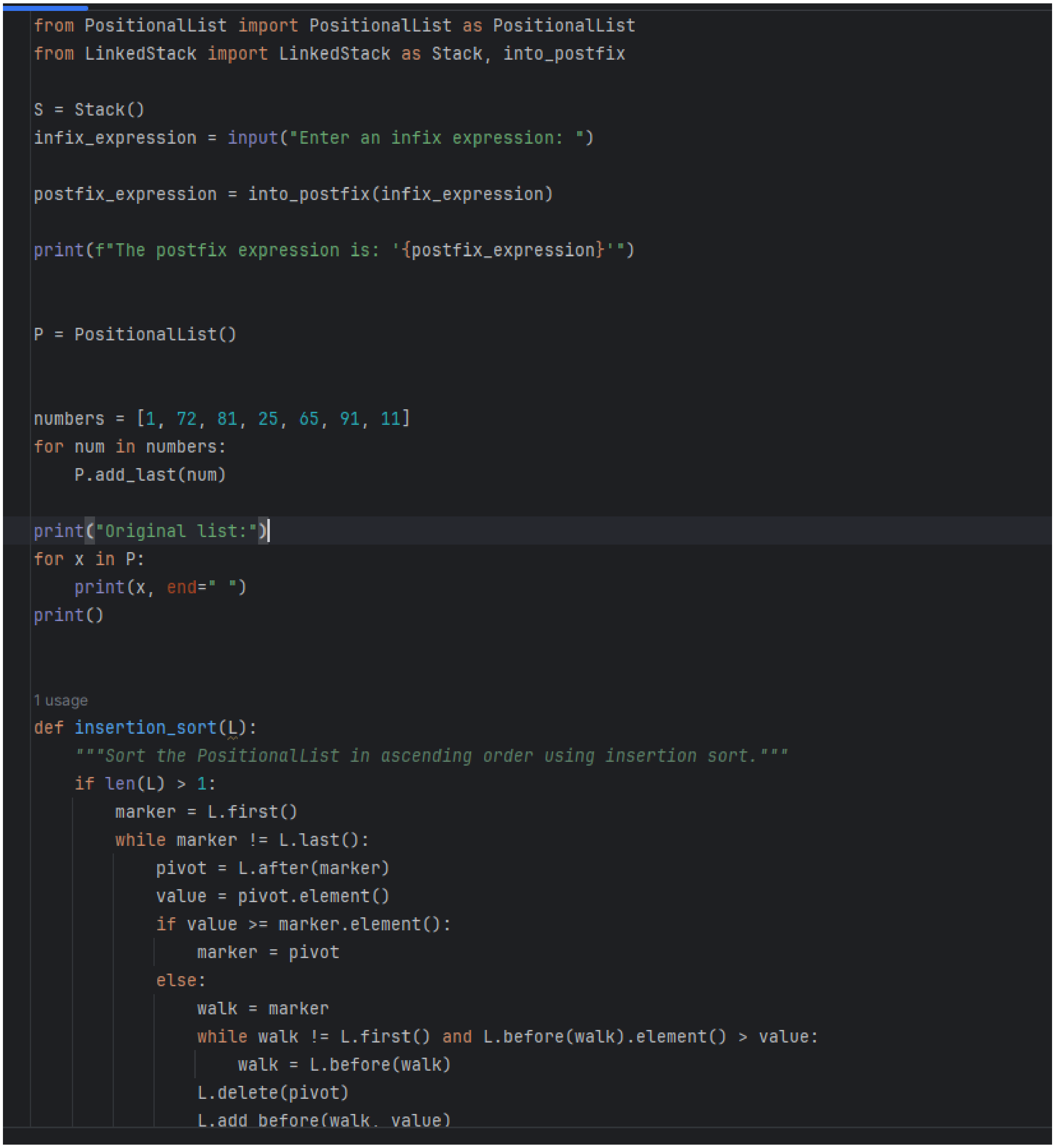
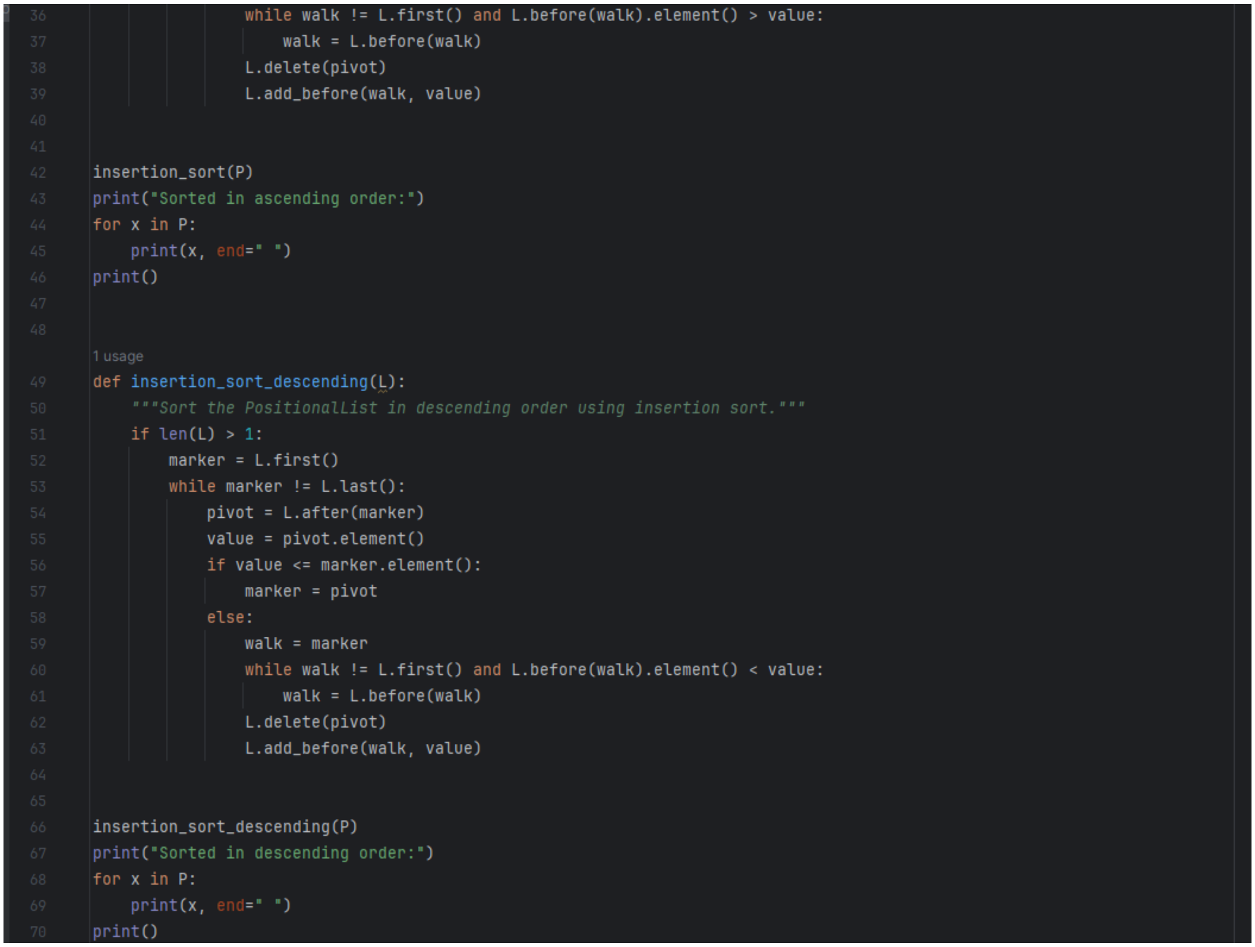
Erece, Kian Relijoe B.

IDB2

Main.py

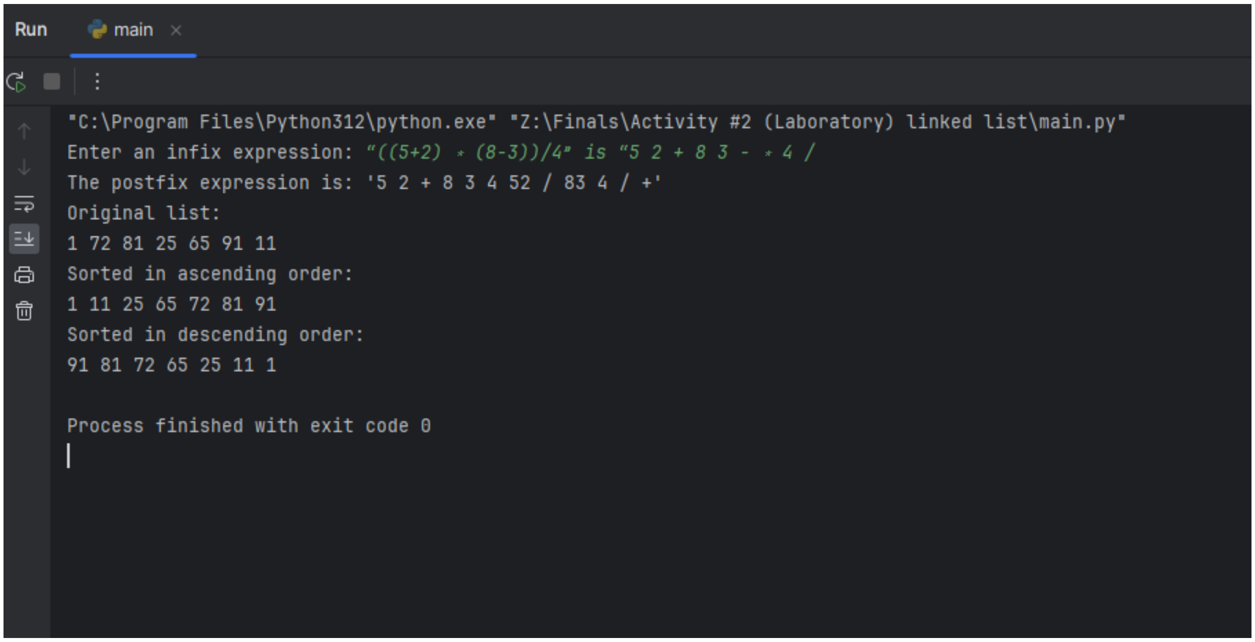


```python
from PositionalList import PositionalList as PositionalList
from LinkedStack import LinkedStack as Stack, into_postfix

S = Stack()
infix_expression = input("Enter an infix expression: ")

postfix_expression = into_postfix(infix_expression)

print(f"The postfix expression is: '{postfix_expression}'")


P = PositionalList()


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

print("Original list:")
for x in P:
    print(x, end=" ")
print()


1 usage
def insertion_sort(L):
    """Sort the PositionalList in ascending order using insertion sort."""
    if len(L) > 1:
        marker = L.first()
        while marker != L.last():
            pivot = L.after(marker)
            value = pivot.element()
            if value >= marker.element():
                marker = pivot
            else:
                walk = marker
                while walk != L.first() and L.before(walk).element() > value:
                    walk = L.before(walk)
                L.delete(pivot)
                L.add_before(walk, value)
```



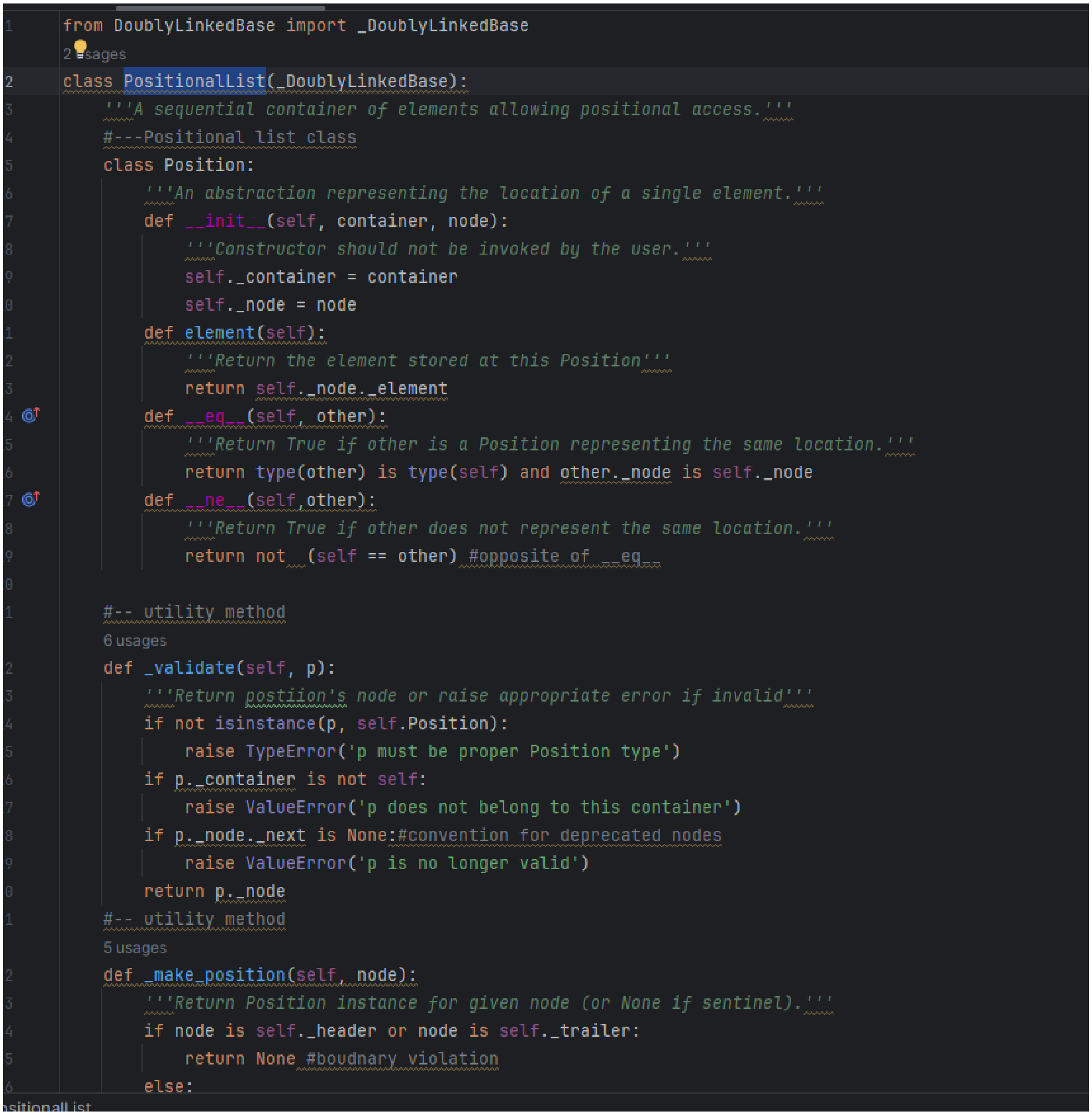
```
            while walk != L.first() and L.before(walk).element() > value:
                walk = L.before(walk)
            L.delete(pivot)
            L.add_before(walk, value)


insertion_sort(P)
print("Sorted in ascending order:")
for x in P:
    print(x, end=" ")
print()


1 usage
def insertion_sort_descending(L):
    """Sort the PositionalList in descending order using insertion sort."""
    if len(L) > 1:
        marker = L.first()
        while marker != L.last():
            pivot = L.after(marker)
            value = pivot.element()
            if value <= marker.element():
                marker = pivot
            else:
                walk = marker
                while walk != L.first() and L.before(walk).element() < value:
                    walk = L.before(walk)
                L.delete(pivot)
                L.add_before(walk, value)


insertion_sort_descending(P)
print("Sorted in descending order:")
for x in P:
    print(x, end=" ")
print()
```

out put:

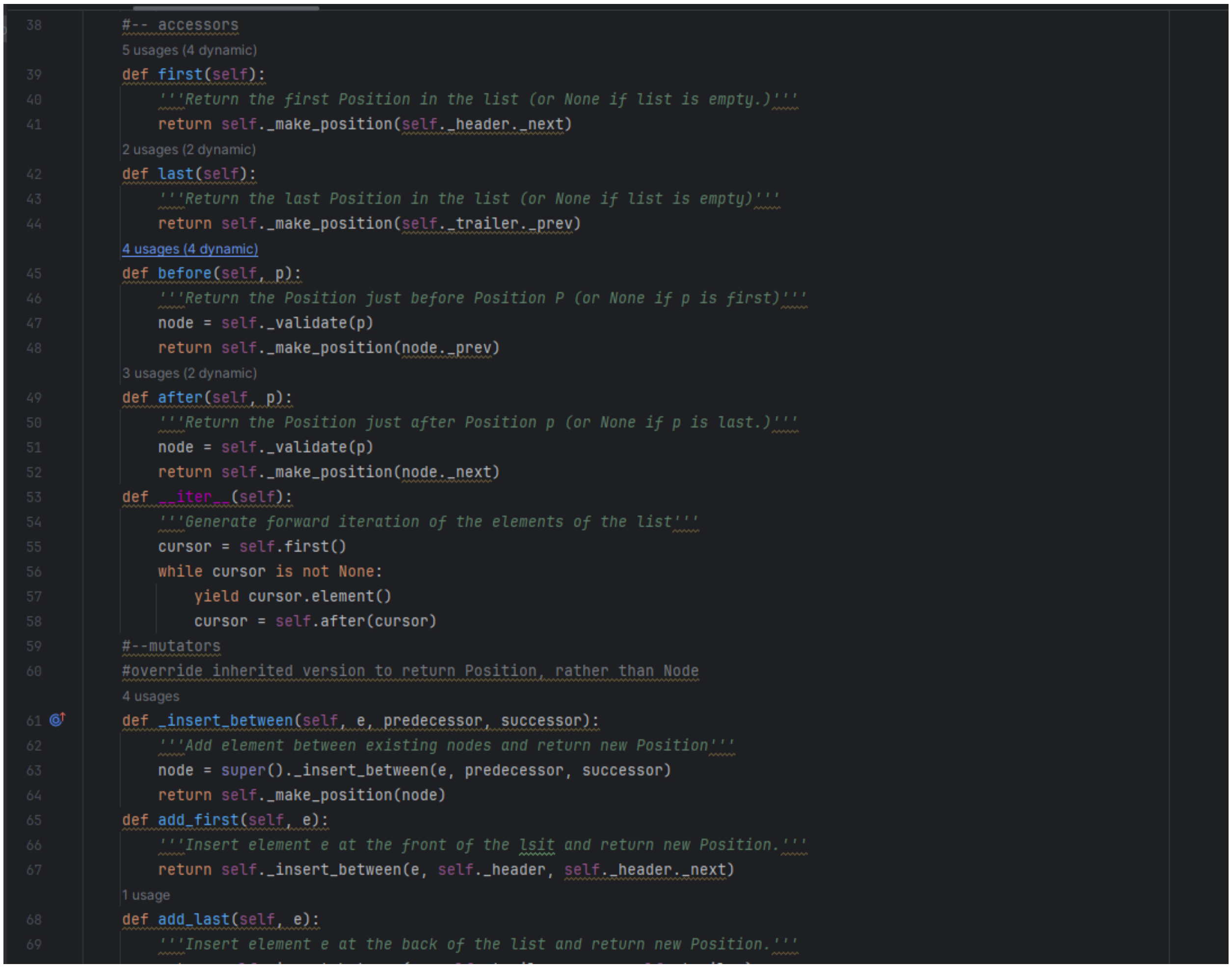


```
"C:\Program Files\Python312\python.exe" "Z:\Finals\Activity #2 (Laboratory) linked list\main.py"
Enter an infix expression: "((5+2) * (8-3))/4" is "5 2 + 8 3 - * 4 /
The postfix expression is: '5 2 + 8 3 4 52 / 83 4 / +'
Original list:
1 72 81 25 65 91 11
Sorted in ascending order:
1 11 25 65 72 81 91
Sorted in descending order:
91 81 72 65 25 11 1

Process finished with exit code 0
```

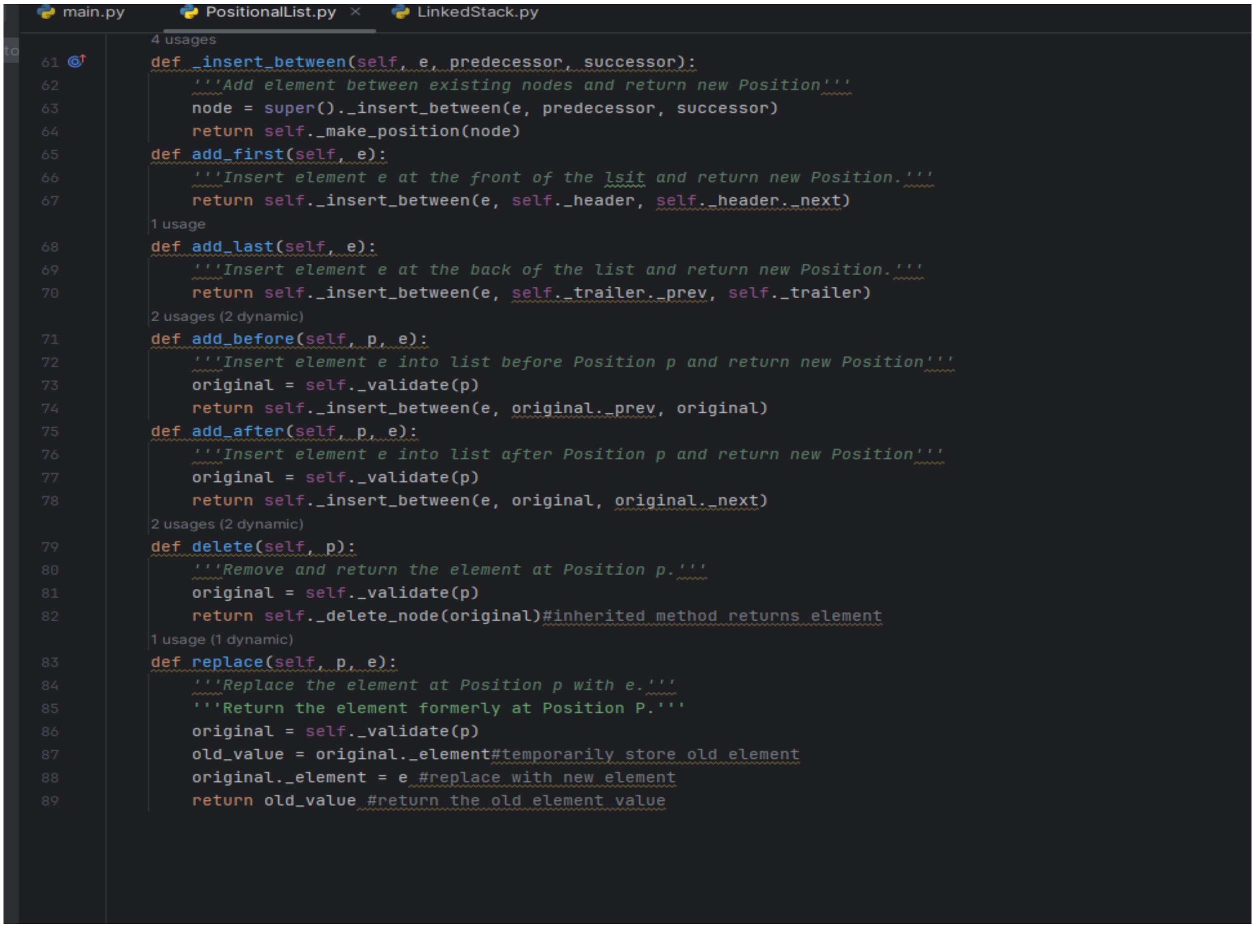
PositionalList.py



```python
from DoublyLinkedBase import _DoublyLinkedBase
2 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:
```



```
#-- accessors
5 usages (4 dynamic)
def first(self):
    '''Return the first Position in the list (or None if list is empty.)'''
    return self._make_position(self._header._next)
2 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)
4 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)
3 usages (2 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)
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)
1 usage
def add_last(self, e):
    '''Insert element e at the back of the list and return new Position.'''
```



```
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)
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)
1 usage
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)
2 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)
2 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
1 usage (1 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
```

LinkedStack.py

```
class LinkedStack:
    """LIFO Stack implementation using a singly linked list for storage."""

    class _Node:
        """Lightweight non-public class for storing a singly linked node."""
        __slots__ = '_element', '_next'

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

    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, element):
        """Add element to the top of the stack."""
        new_node = self._Node(element, self._head)
        self._head = new_node
        self._size += 1

    def top(self):
        """Return but do not remove the element at the top of the stack."""
        if self.is_empty():
            raise Exception('Stack is empty')
        return self._head._element
```

```
def pop(self):
    """Remove and return the element from the top of the stack (LIFO)."""
    if self.is_empty():
        raise Exception("The stack is empty!")
    top_element = self._head._element
    self._head = self._head._next
    self._size -= 1
    return top_element

@staticmethod
def evaluate_postfix(expression):
    """Evaluate a postfix expression using a linked stack."""
    stack = LinkedStack()
    tokens = expression.split()

    for token in tokens:
        if token.isdigit():
            stack.push(int(token))
        else:
            operand2 = stack.pop()
            operand1 = stack.pop()
            result = LinkedStack.perform_operation(operand1, operand2, token)
            stack.push(result)

    return stack.pop()

@staticmethod
def perform_operation(operand1, operand2, operator):
    """Perform arithmetic operations based on the operator."""
    if operator == '+':
        return operand1 + operand2
    elif operator == '-':
        return operand1 - operand2
    elif operator == '*':
        return operand1 * operand2
    elif operator == '/':
        if operand2 == 0:
            raise ZeroDivisionError("Division by zero!")
            return operand1 / operand2
        else:
            raise ValueError(f"Unknown operator: {operator}")


def precedence(operator):
    """Return precedence of operators."""
    if operator in ('+', '-'):
        return 1
    if operator in ('*', '/'):
        return 2
    return 0


def into_postfix(expression):
    """Convert an infix expression to postfix notation."""
    output = []
    operators = LinkedStack()


    tokens = expression.replace(" ", "")

    current_number = ''

    for char in tokens:
        if char.isdigit():
            current_number += char
        else:
            if current_number:
                output.append(current_number)
                current_number = ''
            if char in '+-*/':
                while (not operators.is_empty() and
                       precedence(operators.top()) >= precedence(char)):
                    output.append(operators.pop())
                operators.push(char)
            elif char == '(':
                operators.push(char)
        elif char == ')':
            while not operators.is_empty() and operators.top() != '(':
                output.append(operators.pop())
            operators.pop()

if current_number:
    output.append(current_number)


while not operators.is_empty():
    output.append(operators.pop())

return ' '.join(output)
```