Name

**RAUNAK RAJESH SHAH**

Email

[**shahrrs2004@gmail.com**](mailto:shahrrs2004@gmail.com)

Cohort

**Cohort-29 FSN**

TOPIC

**Data Structure and Algorithms**

College

**Walchand Institute of Technology**

Assignment 2: Stack

**Implementation of Stack Data Structure using Python.**

# **Code:**

# Assignment 2: STACK implementation by RAUNAK SHAH using PYTHON

class Stack:

    def \_\_init\_\_(self):

        self.items = []

    def isEmpty(self):

        return len(self.items) < 1

    def push(self, element):

        self.items.append(element)

    def pop(self):

        if not self.isEmpty():

            return self.items.pop()

        else:

            raise IndexError("Pop method cannot be done when stack is empty. No Element to pop.")

    def peek(self):

        if not self.isEmpty():

            return self.items[-1]

        else:

            raise IndexError("No Element to peek in the stack.")

    def size(self):

        return len(self.items)

    # More Features of Stack

    def is\_balanced(self, expression) -> bool:

        opening\_brackets = "[({"

        closing\_brackets = "])}"

        stack = Stack()

        for char in expression:

            if char in opening\_brackets:

                stack.push(char)

            elif char in closing\_brackets:

                if stack.isEmpty():

                    return False

                top = stack.peek()

                if opening\_brackets.index(top) != closing\_brackets.index(char) or top == None :

                    return False

                stack.pop()

        return stack.isEmpty()

    def evaluate\_infix(self,expression):

        expression = "(" + expression + ")"

        def precedence(operator):

            if operator == '+' or operator == '-':

                return 0

            elif operator == '\*' or operator == '/':

                return 1

            else:

                return 2

        def apply\_operator(operators, values):

            operator = operators.pop()

            right = values.pop()

            left = values.pop()

            if operator == '+':

                values.push(left + right)

            elif operator == '-':

                values.push(left - right)

            elif operator == '\*':

                values.push(left \* right)

            elif operator == '/':

                values.push(left / right)

        operators = Stack()

        values = Stack()

        i = 0

        while i < len(expression):

            if expression[i].isdigit():

                j = i

                while j < len(expression) and (expression[j].isdigit() or expression[j] == '.'):

                    j += 1

                values.push(float(expression[i:j]))

                i = j

            elif expression[i] in "+-\*/":

                while (not operators.isEmpty() and operators.peek() in "+-\*/" and precedence(expression[i]) <= precedence(operators.peek())):

                    apply\_operator(operators, values)

                operators.push(expression[i])

                i += 1

            elif expression[i] == "(":

                operators.push(expression[i])

                i += 1

            elif expression[i] == ")":

                while (not operators.isEmpty() and operators.peek() != '('):

                    apply\_operator(operators, values)

                operators.pop()

                i += 1

            else:

                i += 1

        while not operators.isEmpty():

            apply\_operator(operators, values)

        # return the remaining element in values i.e the final answer

        return values.pop()

new\_stack = Stack()

# Inserting elements

new\_stack.push(10)

new\_stack.push(20)

new\_stack.push(30)

new\_stack.push(40)

new\_stack.push(50)

print(f"Stack: {new\_stack.items}")

print(f"Size: {new\_stack.size()}")

print(f"peek: {new\_stack.peek()}")

print(f"\nRemove element: {new\_stack.pop()}")

print(f"Stack after popping: {new\_stack.items}")

# Testing Exptra features of stack

# Checking for expression balance

print("\nIs expression ([][[]]) balanced: ",new\_stack.is\_balanced("([][[]])"))

print("Is expression ([][[]) balanced: ",new\_stack.is\_balanced("([][[])"))

print("Is expression {([][[]])} balanced: ",new\_stack.is\_balanced("{([[]])}"))

# Evaluating an expression

expression = "((5 + 6) \* (6 - 5) + 1 ) / 3"

print(f"\nexpression: {expression}")

print(f"Result: {new\_stack.evaluate\_infix(expression)}")

# **Output:**

