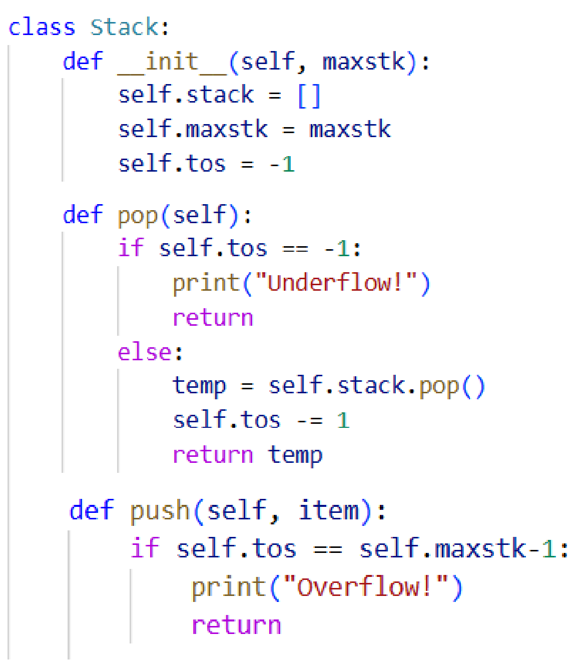
USMAN RASHEED SIDDIQUI CS-24038
NED UNIVERSITY OF ENGINEERING AND TECHNOLOGY DATA STRUCTURES AND ALGORITM

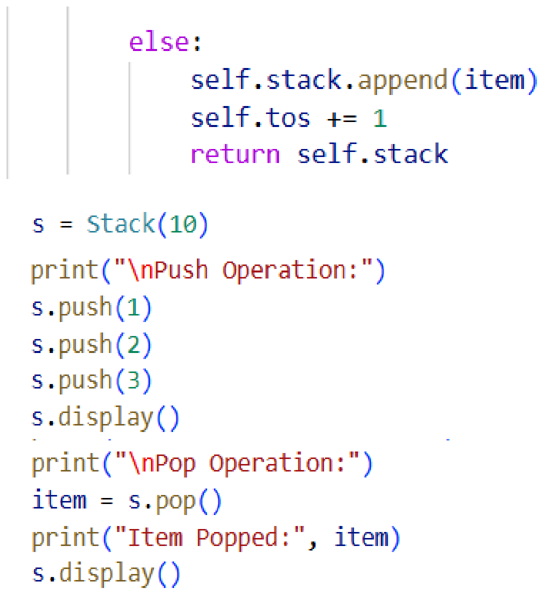
Lab Session 7

Ex A:

CODE

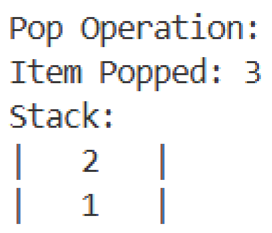


```
class Stack:
    def __init__(self, maxstk):
        self.stack = []
        self.maxstk = maxstk
        self.tos = -1

    def pop(self):
        if self.tos == -1:
            print("Underflow!")
            return
        else:
            temp = self.stack.pop()
            self.tos -= 1
            return temp

    def push(self, item):
        if self.tos == self.maxstk-1:
            print("Overflow!")
            return
```

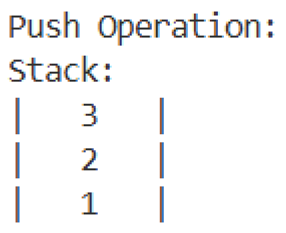


```
        else:
            self.stack.append(item)
            self.tos += 1
            return self.stack

s = Stack(10)
print("\nPush Operation:")
s.push(1)
s.push(2)
s.push(3)
s.display()
print("\nPop Operation:")
item = s.pop()
print("Item Popped:", item)
s.display()
```

OUTPUT



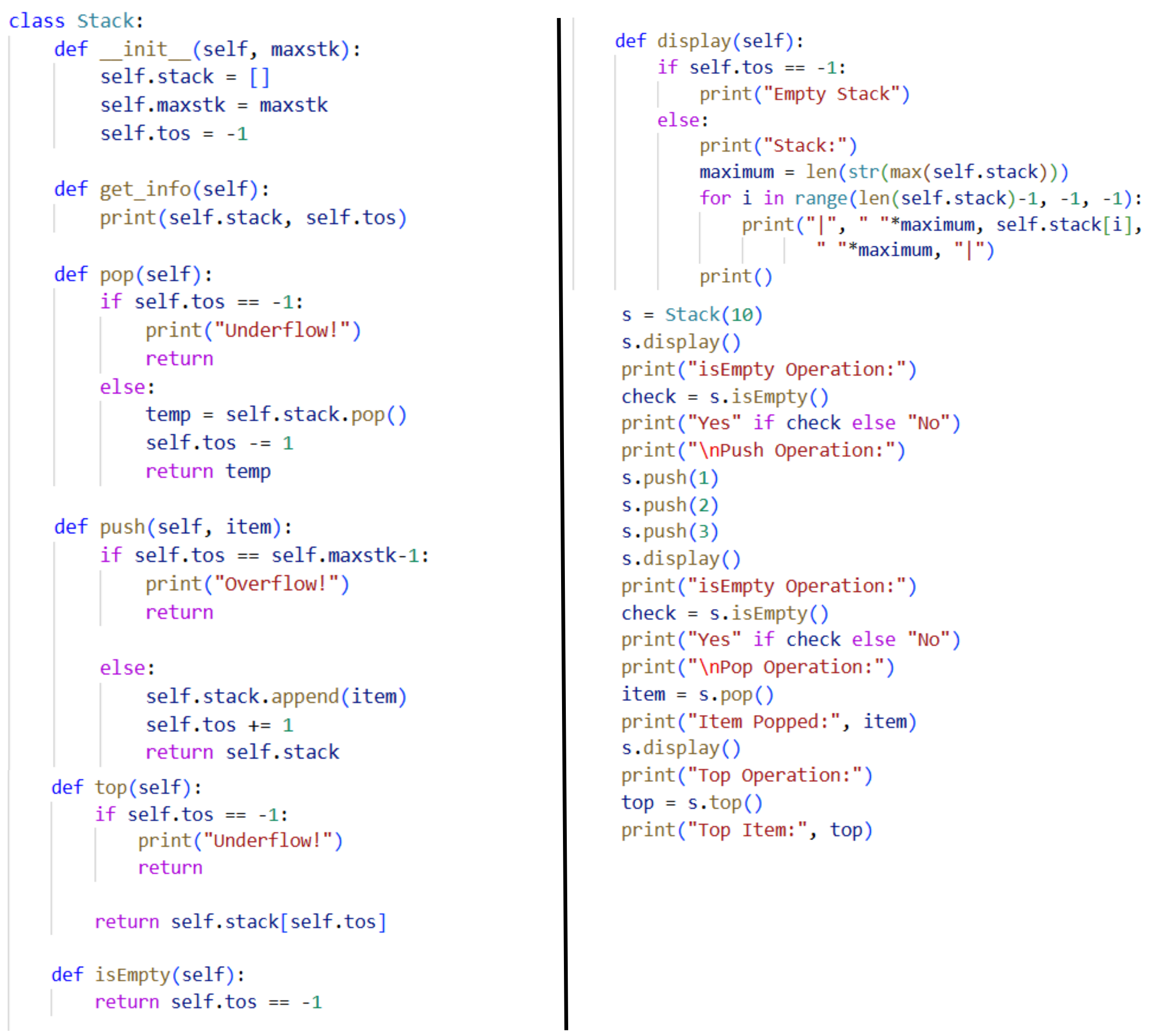
```
Pop Operation:
Item Popped: 3
Stack:
|   2   |
|   1   |
```



```
Push Operation:
Stack:
|   3   |
|   2   |
|   1   |
```

Ex G:

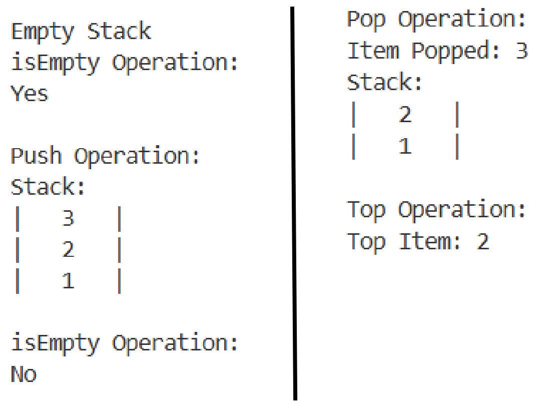
CODE



```
class Stack:
    def __init__(self, maxstk):
        self.stack = []
        self.maxstk = maxstk
        self.tos = -1

    def get_info(self):
        print(self.stack, self.tos)

    def pop(self):
        if self.tos == -1:
            print("Underflow!")
            return
        else:
            temp = self.stack.pop()
            self.tos -= 1
            return temp

    def push(self, item):
        if self.tos == self.maxstk-1:
            print("Overflow!")
            return

        else:
            self.stack.append(item)
            self.tos += 1
            return self.stack
    def top(self):
        if self.tos == -1:
            print("Underflow!")
            return

        return self.stack[self.tos]

    def isEmpty(self):
        return self.tos == -1
```

```
    def display(self):
        if self.tos == -1:
            print("Empty Stack")
        else:
            print("Stack:")
            maximum = len(str(max(self.stack)))
            for i in range(len(self.stack)-1, -1, -1):
                print("|", " "*maximum, self.stack[i],
                      " "*maximum, "|")
            print()

s = Stack(10)
s.display()
print("isEmpty Operation:")
check = s.isEmpty()
print("Yes" if check else "No")
print("\nPush Operation:")
s.push(1)
s.push(2)
s.push(3)
s.display()
print("isEmpty Operation:")
check = s.isEmpty()
print("Yes" if check else "No")
print("\nPop Operation:")
item = s.pop()
print("Item Popped:", item)
s.display()
print("Top Operation:")
top = s.top()
print("Top Item:", top)
```

OUTPUT



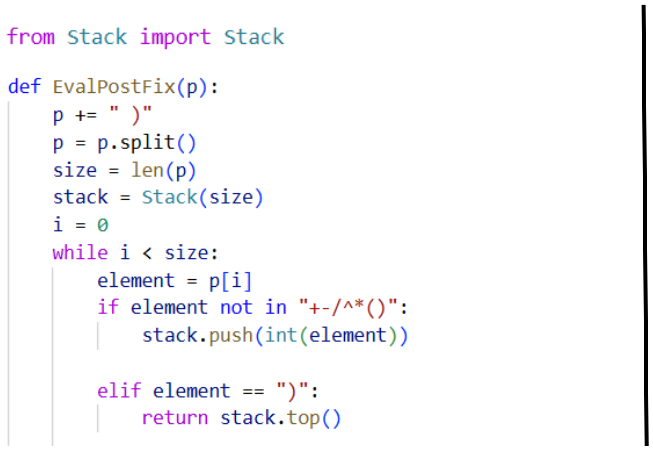
```
Empty Stack
isEmpty Operation:
Yes

Push Operation:
Stack:
|   3   |
|   2   |
|   1   |

isEmpty Operation:
No
```

```
Pop Operation:
Item Popped: 3
Stack:
|   2   |
|   1   |

Top Operation:
Top Item: 2
```

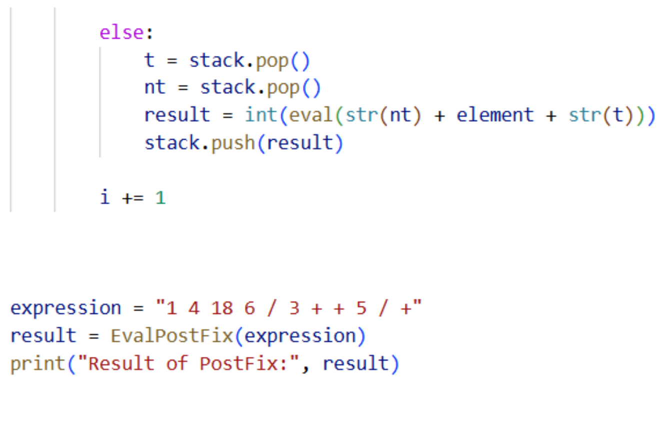
Lab Session 8

Ex A: (Evaluation of a Postfix Expression)

CODE

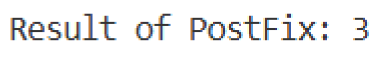


```python
from Stack import Stack

def EvalPostFix(p):
    p += " )"
    p = p.split()
    size = len(p)
    stack = Stack(size)
    i = 0
    while i < size:
        element = p[i]
        if element not in "+-/^*()":
            stack.push(int(element))

        elif element == ")":
            return stack.top()
```



```python
        else:
            t = stack.pop()
            nt = stack.pop()
            result = int(eval(str(nt) + element + str(t)))
            stack.push(result)

        i += 1


expression = "1 4 18 6 / 3 + + 5 / +"
result = EvalPostFix(expression)
print("Result of PostFix:", result)
```

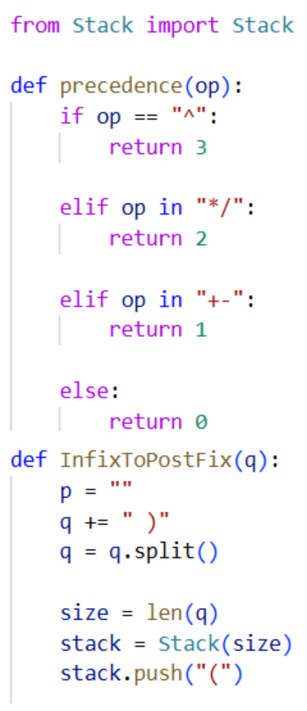
OUTPUT



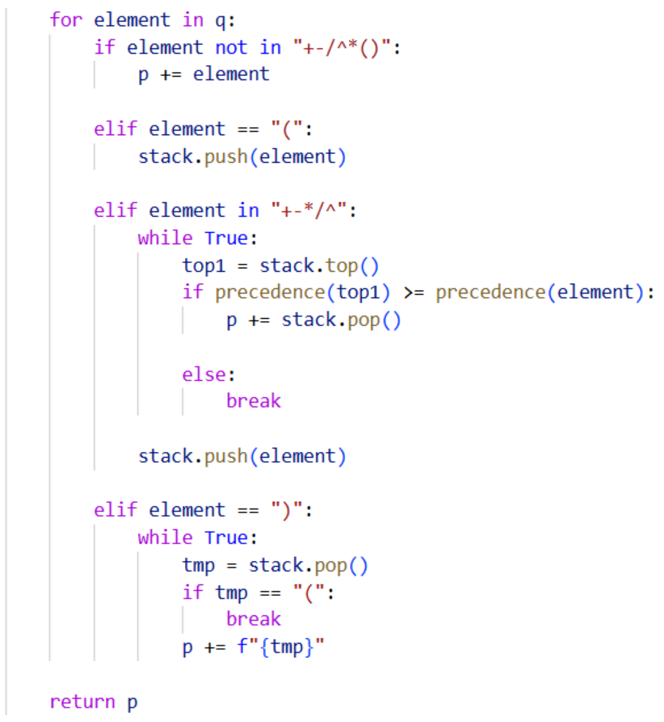
```
Result of PostFix: 3
```

Ex A: (Transforming Infix Expressions into Postfix Expressions)

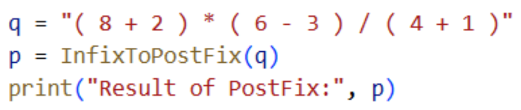
CODE



```python
from Stack import Stack

def precedence(op):
    if op == "^":
        return 3

    elif op in "*/":
        return 2

    elif op in "+-":
        return 1

    else:
        return 0
def InfixToPostFix(q):
    p = ""
    q += " )"
    q = q.split()

    size = len(q)
    stack = Stack(size)
    stack.push("(")
```

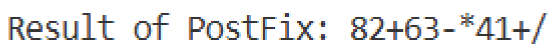


```python
    for element in q:
        if element not in "+-/^*()":
            p += element

        elif element == "(":
            stack.push(element)

        elif element in "+-*/^":
            while True:
                top1 = stack.top()
                if precedence(top1) >= precedence(element):
                    p += stack.pop()

                else:
                    break

            stack.push(element)

        elif element == ")":
            while True:
                tmp = stack.pop()
                if tmp == "(":
                    break
                p += f"{tmp}"

    return p
```



```python
q = "( 8 + 2 ) * ( 6 - 3 ) / ( 4 + 1 )"
p = InfixToPostFix(q)
print("Result of PostFix:", p)
```

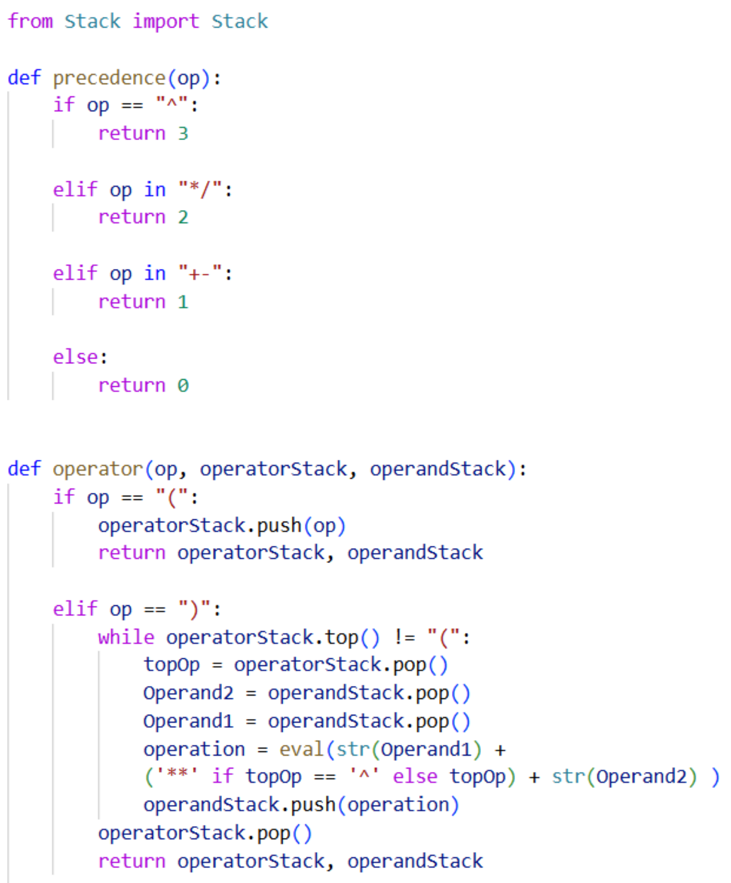
OUTPUT



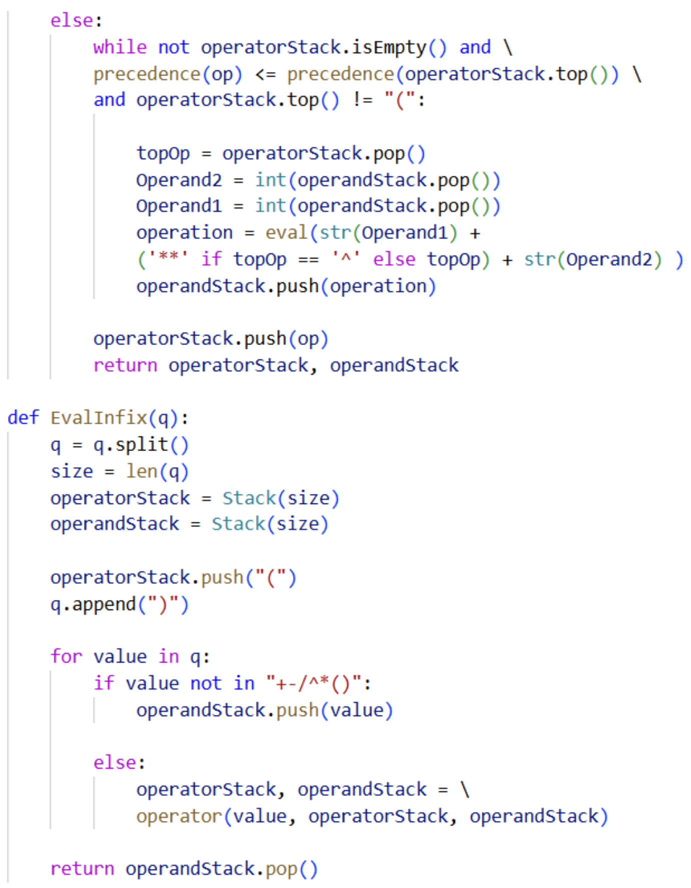
```
Result of PostFix: 82+63-*41+/
```

Ex A: (Evaluate Infix Expressions)

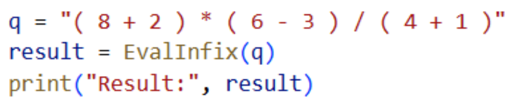
CODE



```python
from Stack import Stack

def precedence(op):
    if op == "^":
        return 3

    elif op in "*/":
        return 2

    elif op in "+-":
        return 1

    else:
        return 0


def operator(op, operatorStack, operandStack):
    if op == "(":
        operatorStack.push(op)
        return operatorStack, operandStack

    elif op == ")":
        while operatorStack.top() != "(":
            topOp = operatorStack.pop()
            Operand2 = operandStack.pop()
            Operand1 = operandStack.pop()
            operation = eval(str(Operand1) +
            ('**' if topOp == '^' else topOp) + str(Operand2) )
            operandStack.push(operation)
        operatorStack.pop()
        return operatorStack, operandStack
```

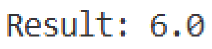


```python
    else:
        while not operatorStack.isEmpty() and \
        precedence(op) <= precedence(operatorStack.top()) \
        and operatorStack.top() != "(":

            topOp = operatorStack.pop()
            Operand2 = int(operandStack.pop())
            Operand1 = int(operandStack.pop())
            operation = eval(str(Operand1) +
            ('**' if topOp == '^' else topOp) + str(Operand2) )
            operandStack.push(operation)

        operatorStack.push(op)
        return operatorStack, operandStack

def EvalInfix(q):
    q = q.split()
    size = len(q)
    operatorStack = Stack(size)
    operandStack = Stack(size)

    operatorStack.push("(")
    q.append(")")

    for value in q:
        if value not in "+-/^*()":
            operandStack.push(value)

        else:
            operatorStack, operandStack = \
            operator(value, operatorStack, operandStack)

    return operandStack.pop()
```



```python
q = "( 8 + 2 ) * ( 6 - 3 ) / ( 4 + 1 )"
result = EvalInfix(q)
print("Result:", result)
```

OUTPUT



```
Result: 6.0
```