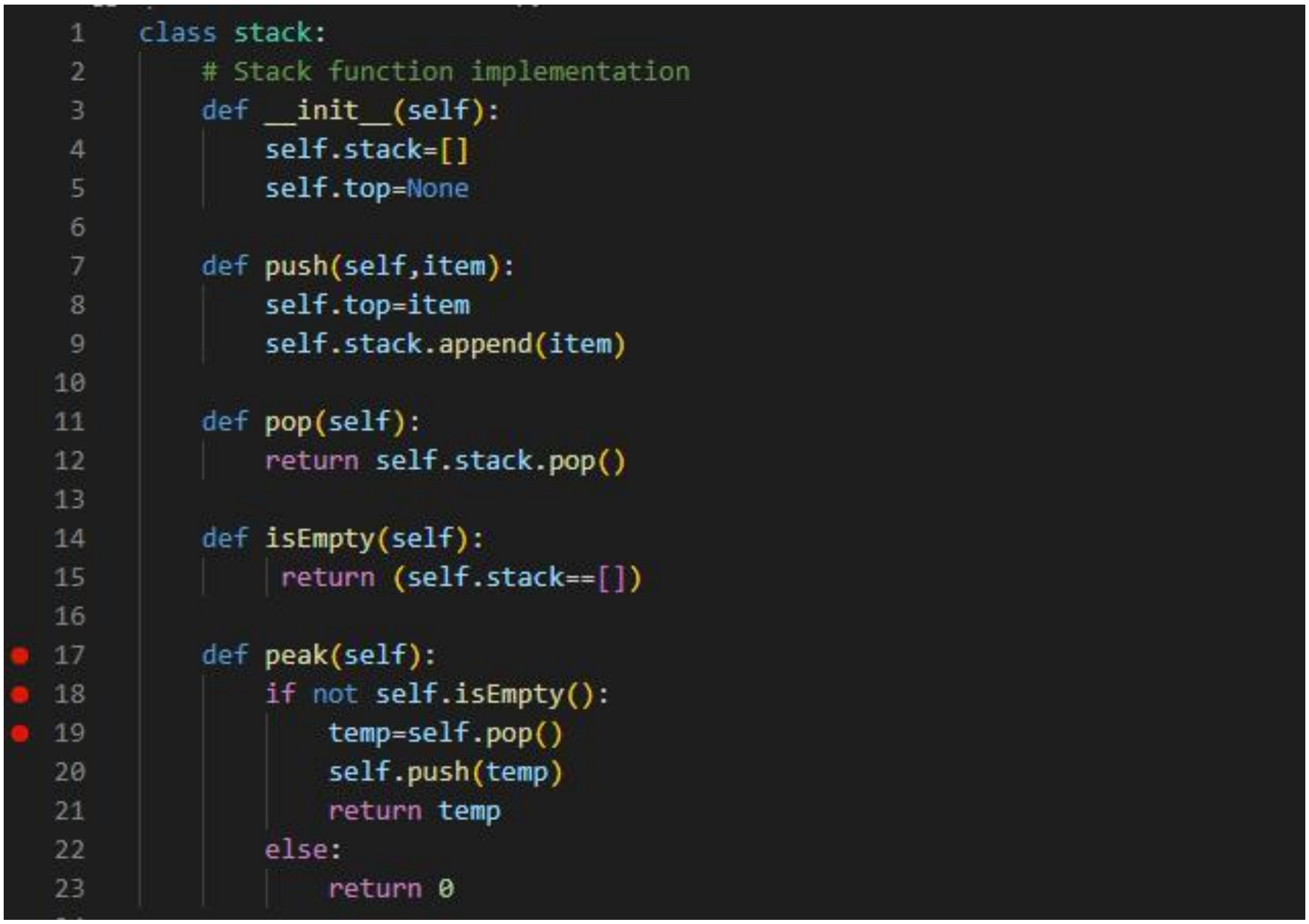
Description of Calculator project

Teacher: Dr. Fatemi

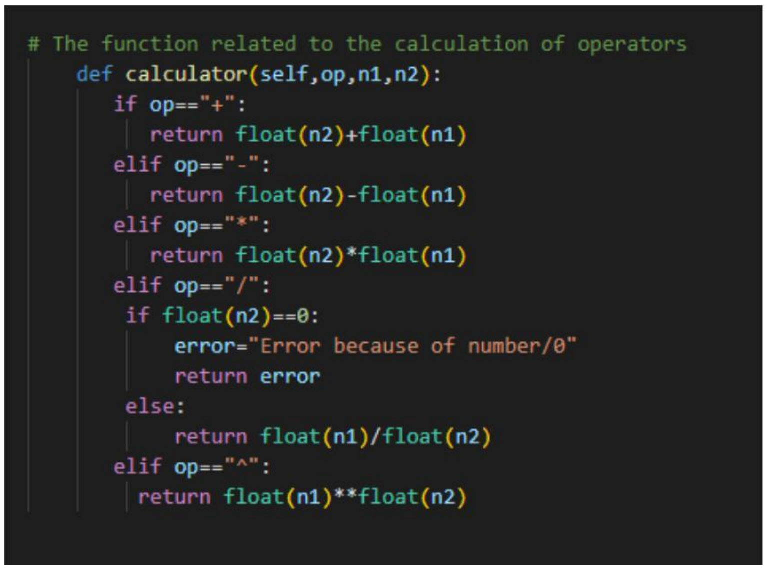
Student: Sara Rauofi

At the beginning of the program, I wrote the class related to the stack to implement it using the list, which includes functions (isEmpty, peak, push, pop).

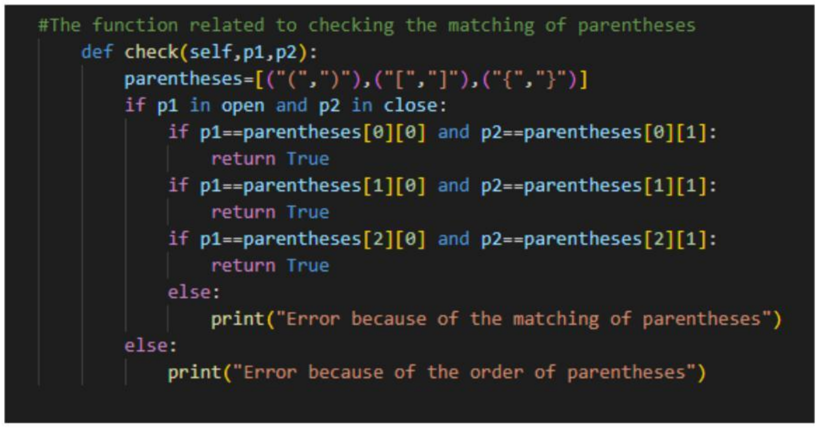


```python
class stack:
    # Stack function implementation
    def __init__(self):
        self.stack=[]
        self.top=None

    def push(self,item):
        self.top=item
        self.stack.append(item)

    def pop(self):
        return self.stack.pop()

    def isEmpty(self):
         return (self.stack==[])

    def peak(self):
        if not self.isEmpty():
            temp=self.pop()
            self.push(temp)
            return temp
        else:
            return 0
```

Then I wrote the expression calculation function (addition, multiplication, division, power, subtraction). And for division, I considered the error of dividing by zero. After this, I wrote a function to check the parentheses to see if they match or not and if their location is correct or not.

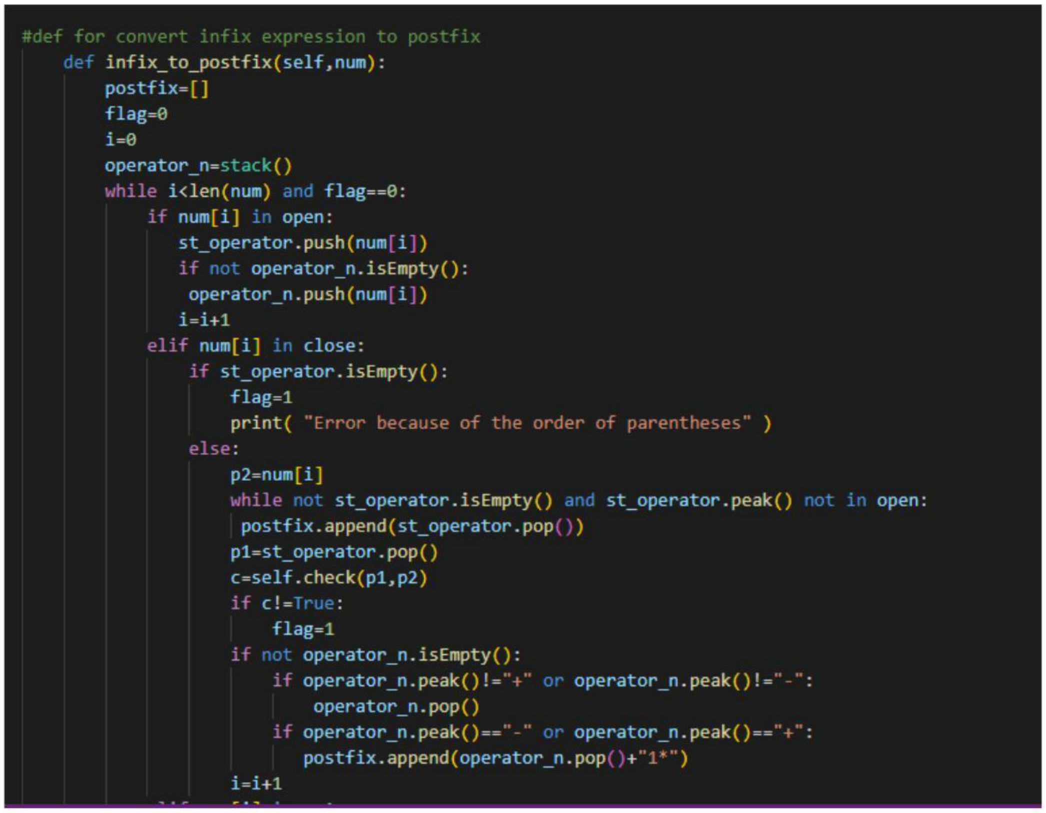


```python
# The function related to the calculation of operators
    def calculator(self,op,n1,n2):
        if op=="+":
            return float(n2)+float(n1)
        elif op=="-":
            return float(n2)-float(n1)
        elif op=="*":
            return float(n2)*float(n1)
        elif op=="/":
         if float(n2)==0:
            error="Error because of number/0"
            return error
         else:
            return float(n1)/float(n2)
        elif op=="^":
          return float(n1)**float(n2)
```

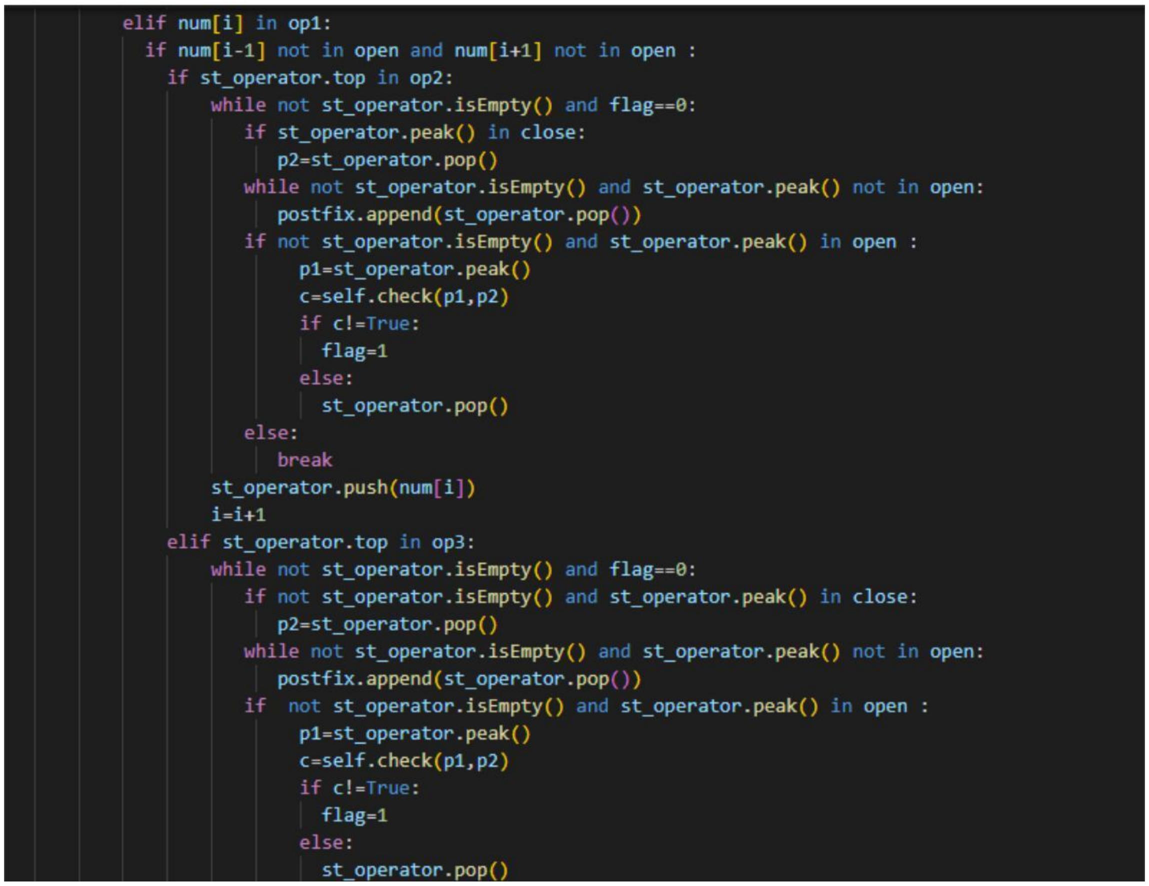


```python
#The function related to checking the matching of parentheses
    def check(self,p1,p2):
        parentheses=[("(",")"),("[","]"),("{","}")]
        if p1 in open and p2 in close:
            if p1==parentheses[0][0] and p2==parentheses[0][1]:
                return True
            if p1==parentheses[1][0] and p2==parentheses[1][1]:
                return True
            if p1==parentheses[2][0] and p2==parentheses[2][1]:
                return True
            else:
                print("Error because of the matching of parentheses")
        else:
            print("Error because of the order of parentheses")
```

After that, I wrote the function to convert infix to postfix. For this conversion, I considered a list to put the converted expression in it. The conversion method is that from the beginning of the expression, if we come across parentheses or operands, we put them in a stack, and if we come across an operator, we put them in the list. Stack members are removed one by one and added to the list whenever we reach the closing parenthesis or according to the order of operands.

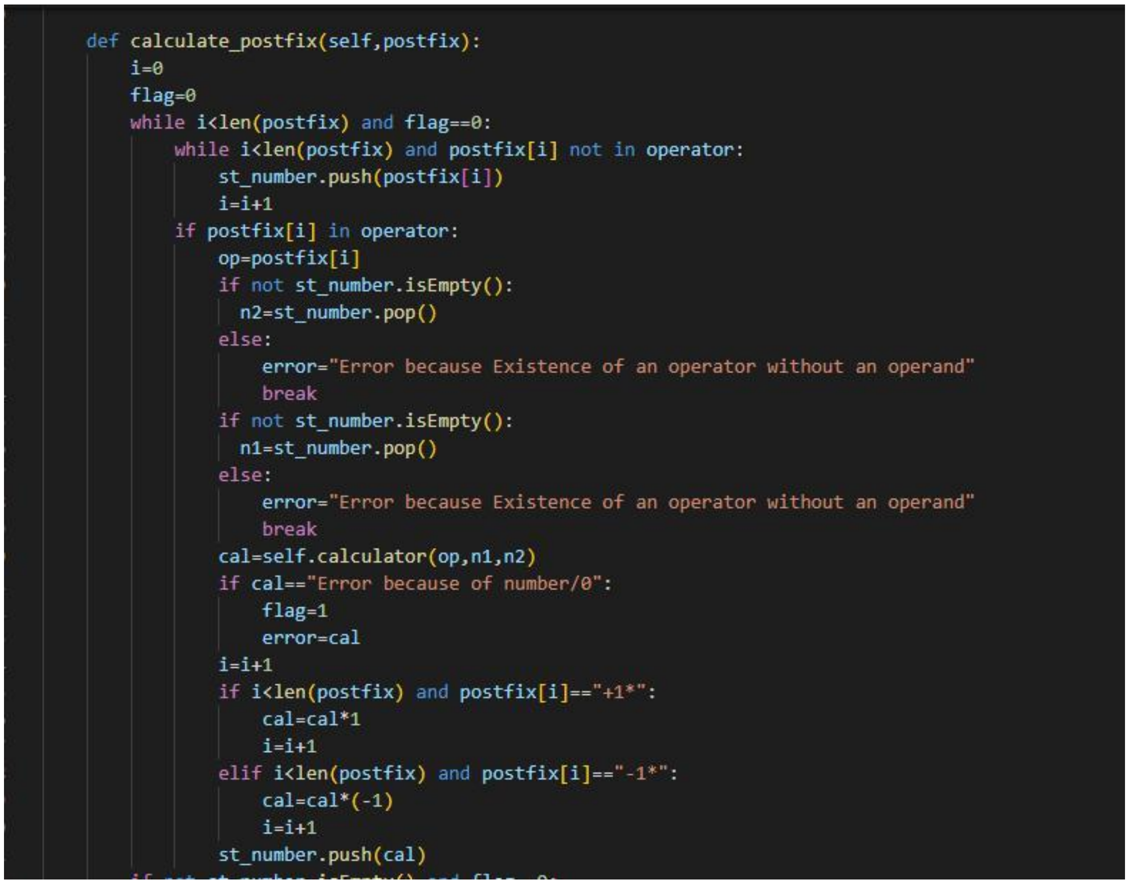


```
#def for convert infix expression to postfix
    def infix_to_postfix(self,num):
        postfix=[]
        flag=0
        i=0
        operator_n=stack()
        while i<len(num) and flag==0:
            if num[i] in open:
               st_operator.push(num[i])
               if not operator_n.isEmpty():
                operator_n.push(num[i])
               i=i+1
            elif num[i] in close:
                if st_operator.isEmpty():
                    flag=1
                    print( "Error because of the order of parentheses" )
                else:
                    p2=num[i]
                    while not st_operator.isEmpty() and st_operator.peak() not in open:
                     postfix.append(st_operator.pop())
                    p1=st_operator.pop()
                    c=self.check(p1,p2)
                    if c!=True:
                        flag=1
                    if not operator_n.isEmpty():
                        if operator_n.peak()!="+" or operator_n.peak()!="-":
                            operator_n.pop()
                        if operator_n.peak()=="-" or operator_n.peak()=="+":
                           postfix.append(operator_n.pop()+"1*")
                    i=i+1
```



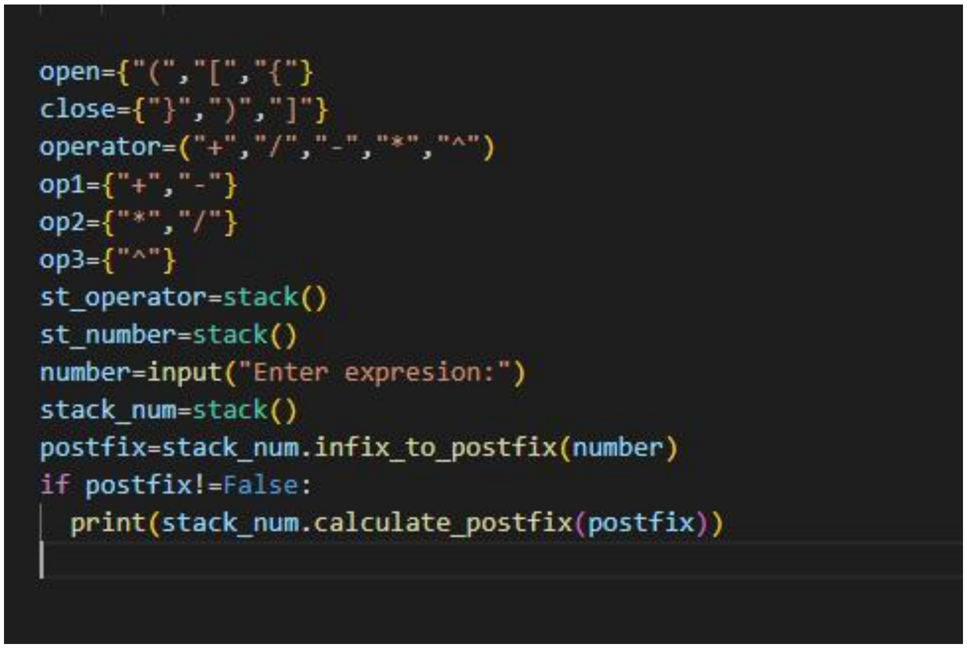
```
            elif num[i] in op1:
              if num[i-1] not in open and num[i+1] not in open :
                if st_operator.top in op2:
                    while not st_operator.isEmpty() and flag==0:
                       if st_operator.peak() in close:
                          p2=st_operator.pop()
                       while not st_operator.isEmpty() and st_operator.peak() not in open:
                          postfix.append(st_operator.pop())
                       if not st_operator.isEmpty() and st_operator.peak() in open :
                            p1=st_operator.peak()
                            c=self.check(p1,p2)
                            if c!=True:
                              flag=1
                            else:
                              st_operator.pop()
                       else:
                          break
                    st_operator.push(num[i])
                    i=i+1
                elif st_operator.top in op3:
                    while not st_operator.isEmpty() and flag==0:
                       if not st_operator.isEmpty() and st_operator.peak() in close:
                          p2=st_operator.pop()
                       while not st_operator.isEmpty() and st_operator.peak() not in open:
                          postfix.append(st_operator.pop())
                       if  not st_operator.isEmpty() and st_operator.peak() in open :
                            p1=st_operator.peak()
                            c=self.check(p1,p2)
                            if c!=True:
                              flag=1
                            else:
                              st_operator.pop()
```

Then I wrote the calculation function of the obtained postfix expression to get the result of our expression. Here, we add the members of the list to the stack one by one from the beginning, and if we reach the operand, I execute the operand on the previous two numbers in the stack, and then put the result back in the stack and...



```
    def calculate_postfix(self,postfix):
        i=0
        flag=0
        while i<len(postfix) and flag==0:
            while i<len(postfix) and postfix[i] not in operator:
                st_number.push(postfix[i])
                i=i+1
            if postfix[i] in operator:
                op=postfix[i]
                if not st_number.isEmpty():
                  n2=st_number.pop()
                else:
                    error="Error because Existence of an operator without an operand"
                    break
                if not st_number.isEmpty():
                  n1=st_number.pop()
                else:
                    error="Error because Existence of an operator without an operand"
                    break
                cal=self.calculator(op,n1,n2)
                if cal=="Error because of number/0":
                    flag=1
                    error=cal
                i=i+1
                if i<len(postfix) and postfix[i]=="+1*":
                    cal=cal*1
                    i=i+1
                elif i<len(postfix) and postfix[i]=="-1*":
                    cal=cal*(-1)
                    i=i+1
                st_number.push(cal)
```

In the end, in the main function or the main part of the program, I received the expression from the user and using the above functions and the stack class, I obtained the result.



```
open={"(","[","{"}
close={"}",")","]"}
operator=("+","/","-","*","^")
op1={"+","-"}
op2={"*","/"}
op3={"^"}
st_operator=stack()
st_number=stack()
number=input("Enter expresion:")
stack_num=stack()
postfix=stack_num.infix_to_postfix(number)
if postfix!=False:
  print(stack_num.calculate_postfix(postfix))
```