Table of Contents

1. Introduction

This report outlines a simple approach to evaluating mathematical expressions using a stack-based system in Python. The stack data structure is fundamental in handling expressions with operators (+, -, *, /) and parentheses, ensuring that operations are executed in the correct order according to their precedence.

The program reads mathematical expressions from an input file, evaluates them using two stacks: one for operands (numbers) and another for operators, and then writes the evaluated results to an output file. This report covers the structure of the stack class, the process of expression evaluation, and the complete handling of file reading and writing.

---

2. Stack Class Overview

The Stack class provides the basic structure for managing operands and operators during expression evaluation. The class allows for pushing, popping, and peeking at elements, as well as checking if the stack is empty.

2.1 Constructor and Initialization

```
class Stack:
    def __init__(self):
        self.items = []
```

The Stack class is initialized with an empty list, self.items. This list will be used to store elements in a last-in, first-out (LIFO) manner. Python's list methods make it easy to simulate stack operations.

2.2 Stack Operations (Push, Pop, Peek, Is Empty)

```
def push(self, item):
    self.items.append(item)
```

- **Push**: The push() method adds a new item to the top of the stack. In this implementation, the append() function is used to add an element to the end of the list.

```
def pop(self):
    if not self.is_empty():
        return self.items.pop()
```

- **Pop**: The pop() method removes and returns the top item from the stack, but only if the stack is not empty (checked with the is_empty() method).

```
def peek(self):
    if not self.is_empty():
        return self.items[-1]
```

- **Peek**: The peek() method returns the top item without removing it, allowing you to inspect the stack’s top element.

```
def is_empty(self):
    return len(self.items) == 0
```

- **Is Empty**: The is_empty() method checks if the stack contains any elements and returns True if it is empty, False otherwise.

3. Expression Evaluation Using Stacks

The evaluate_expression() function handles the core logic of evaluating a mathematical expression using two stacks: one for storing operands (numbers) and one for operators (+, -, *, /).

3.1 Parsing the Expression

```
def evaluate_expression(expression):
    operands = Stack()
    operators = Stack()
    precedence = {'+': 1, '-': 1, '*': 2, '/': 2}
    i = 0
```

- Two stacks are initialized: operands (for numbers) and operators (for operators like + or -).
- A dictionary precedence is defined to assign precedence levels to operators (* and / have higher precedence than + and -).
- The variable i is used to traverse the input expression character by character.

3.2 Applying Operators

```
def apply_operator(operands, operator):
    b = operands.pop()
    a = operands.pop()
    if operator == '+':
        operands.push(a + b)
    elif operator == '-':
        operands.push(a - b)
    elif operator == '*':
        operands.push(a * b)
    elif operator == '/':
        operands.push(a / b)
```

- The apply_operator() function pops two operands from the operands stack, applies the operator to them, and pushes the result back onto the operands stack.
- This ensures that the operations are applied in the correct order.

3.3 Handling Parentheses

```
while operators.peek() != '(':
    apply_operator(operands, operators.pop())
operators.pop()
```

(# Remove the opening parenthesis)

- When a closing parenthesis) is encountered, the program pops operators and applies them to the operands until the opening parenthesis (is reached.
- The opening parenthesis is then removed from the stack.

4. File Processing Explanation

The process_file() function is responsible for reading expressions from the input file, evaluating them, and writing the results to the output file. Let's examine how the file handling works.

4.1 Reading Expressions from the Input File

```
with open(input_file, 'r') as infile, open(output_file, 'w') as outfile:
```

- The open() function is used to open the input file (input_file) in read mode ('r') and the output file (output_file) in write mode ('w').
- The with statement ensures that both files are closed automatically once processing is complete.

4.2 Writing Results to the Output File

```
outfile.write(f"{result}\n")
```

- After evaluating an expression, the result is written to the output file, followed by a newline character (\n), so that each result appears on a separate line in the output file.

4.3 File Processing

Code Block:

```
def process_file(input_file, output_file):
    with open(input_file, 'r') as infile, open(output_file, 'w') as outfile:
        for line in infile:
            if line.strip() == '--------':        # Check for separator lines
                outfile.write('----\n')            # Write a separator in the output
            else:
                result = evaluate_expression(line.strip())        # Evaluate the expression
                outfile.write(f"{result}\n")            # Write the result to the output file
```

- **Line 1**: The process_file() function takes two arguments: the input_file to read from and the output_file to write to.
- **Line 2**: The with statement opens both the input file and the output file.
- **Line 3**: A for loop is used to iterate over each line in the input file.
- **Line 4**: The if statement checks whether the current line is a separator (--------). The strip() method is used to remove any leading or trailing whitespace before comparison.
- **Line 5**: If the line is a separator, a new separator (----) is written to the output file.
- **Line 6**: If the line is not a separator, it is treated as a mathematical expression. The evaluate_expression() function is called to compute the result.
- **Line 7**: The result of the evaluation is written to the output file, followed by a newline.

5. Conclusion

This report presents a simplified solution for evaluating mathematical expressions using stacks in Python. The stack data structure allows for handling complex expressions, ensuring that operators are applied in the correct order and that parentheses are processed properly. The file processing section of the code enables the program to read expressions from an input file, evaluate them, and write the results to an output file. By breaking the problem into smaller, manageable components (stack operations, expression parsing, and file processing), we ensure that the solution is both efficient and easy to understand for beginners.

This method provides a solid foundation for evaluating arithmetic expressions and can easily be extended to support additional operators or functionality, such as handling more complex mathematical functions.

References

- CS50 2018 – LECTURE 4 – Data Structures https://youtu.be/ed2lnJNf7HU?si=_ovo7zvAsZoE54Qc
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.
- Lafore, R. (2002). Data Structures and Algorithms in Java (2nd ed.). Sams Publishing.
- Knuth, D. E. (1997). The Art of Computer Programming: Volume 1 - Fundamental Algorithms (3rd ed.). Addison-Wesley.
- Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D.(2006). Compilers: Principles, Techniques, and Tools (2nd ed.). Pearson.

https://www.youtube.com/watch?v=peYV7VUKFSM