Design Documentation

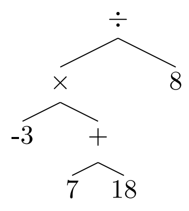
Raman Mathur

November 11, 2019

1 Introduction

For the coding challenge I initially thought to use an Abstract Syntax Tree which can be accessed with the ast module in python. I chose this over a flat structure that would break the expression into list of indices due to the ease of breaking a expression into a tree with the operators as nodes. The design would be as such:

Given the equation $\frac{(7+18)\times -3}{8}$ which would be input as string '((7+8)*-3)/8' the tree-breakdown would be as such:



Going from the leaves the algorithm would work upwards, and travel through the nodes to arrive at an answer.

After looking into the ast module and the various algorithms that would help me such as Dijkstra's shunting yard algorithm I realized that the ast module, particularly the ast.parse() method, would be too good for the task due to the inbuilt parsing capabilities. This would be akin to me using eval() and passing the string to Python command line for an easy but unsafe answer. So I thought it best to work with a flat structure that intelligently slices the string, evaluates, inserts and stitches the string, till it breaks the string down to a single value as this would show the effort I put on my end against the imposed constraints.

2 Approach

For working with the string I thought it best to create a class that has a defined method to deal with parentheses and let the evaluating program know which indices to slice and evaluate and then where to re-insert. This sliced portion has the operations run in the DMAS order, and then stitched back into the string without where the parentheses would have been. An example (red indicates that the part is being disregarded for the moment):

$$(42+24)\times 0.5$$
$$\downarrow$$
$$42+24\ \times 0.5$$
$$\downarrow$$
$$66\times 0.5$$
$$\downarrow$$
$$33$$

3 Code

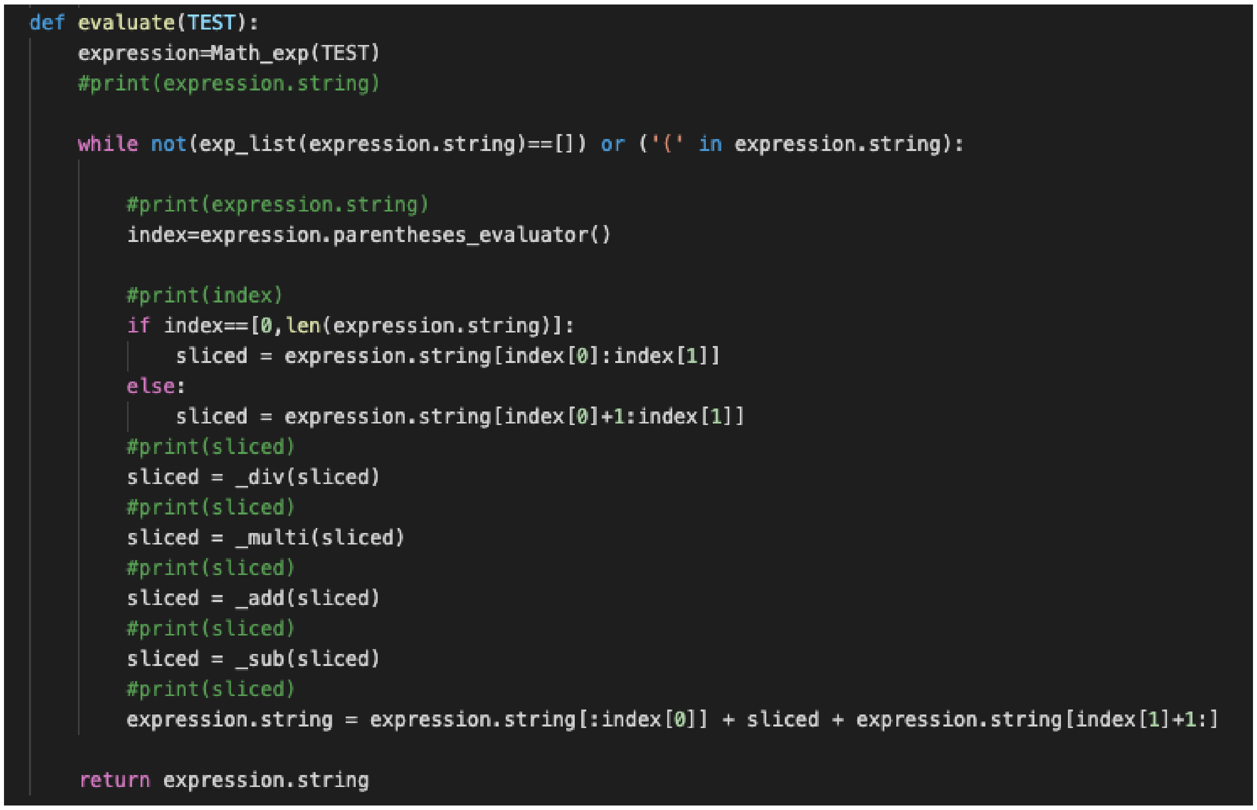
The approach requires multiple parts and functions to work cohesively:

- A robust parentheses checker that would allow for accurate slicing and splicing.
- Functions that take in the sliced string and divide/multiply/add/subtract by finding the operator index and fanning outwards both left and right to intelligently get the operands, operate then splice the value back in. This would be done till all operators are exhausted within the string down the DMAS order. Initially considered making this a Class method but given the slice-splice method thought it best to be a function in itself that works in tandem with the parentheses method.
- An evaluator function that reconciles these functions to enforce the DMAS order, and also loop till a single final value is returned.
- A class that is initialized with the string and deals with whitespace, but also houses the parentheses method for ease of access.
- A function that returns an operator index that also knows to disregard the negative sign.

A big issue that arose while working was floating point arithmetic which is addressed by using Python's Decimal module. It rounds away -by specifications- the finicky precision errors seen in the float data type. Co-prime number division will of course yield but the parser will without exception round to two decimal places for ease of use and aesthetics. This may however lead to error accumulating over long operations, which is the cost for nice-looking numbers.

4 Testability

The testing was done with Python's unittest module. The functions were devised to check for all the required parser behaviour i.e DMAS order, correct order of nested parentheses, proper arithmetic, disregard for white space. These parameters can be altered as wished and the file will run them to check if the evaluator was correct or not.



```
def evaluate(TEST):
    expression=Math_exp(TEST)
    #print(expression.string)

    while not(exp_list(expression.string)==[]) or ('(' in expression.string):

        #print(expression.string)
        index=expression.parentheses_evaluator()

        #print(index)
        if index==[0,len(expression.string)]:
            sliced = expression.string[index[0]:index[1]]
        else:
            sliced = expression.string[index[0]+1:index[1]]
        #print(sliced)
        sliced = _div(sliced)
        #print(sliced)
        sliced = _multi(sliced)
        #print(sliced)
        sliced = _add(sliced)
        #print(sliced)
        sliced = _sub(sliced)
        #print(sliced)
        expression.string = expression.string[:index[0]] + sliced + expression.string[index[1]+1:]

    return expression.string
```

Figure 1: Test unit