Project 2: Evaluating Expressions using stacks
CPE202
Winter 2020

1. Background (https://en.wikipedia.org/wiki/Exponentiation)

This project is built upon the project 1, and we use Stacks you implemented in the project. Make sure your stacks.py work perfectly before starting this project. For this project, you will implement a program to evaluate an infix expression, the kind of expression used in standard arithmetic. The program will do this is two steps, each step implemented as a separate Python function:

1. Convert the infix expression to a postfix expression.
2. Evaluate the postfix expression

Both of these steps make use of a stack in an interesting way. Many language translators (e.g. compiler) do something similar to convert expressions into code that is easy to execute on a computer.

Section 4.9 of the text has a detailed discussion of this material that you should read carefully before building your implementation. You program must use your own implementations of the Stack Abstract Data Type. Specifically, **you need to use either your StackArray class or StackLinked class as your stack.**

Notes:

- Expression will only consist of numbers (integers, reals, positive or negative), parentheses, and the 6 operators separated by spaces.
- The text discusses an easier version of this problem in detail. You should read it carefully before proceeding. In addition to the **operators (+,-,*,/)** shown in the text, your programs should handle the exponentiation operator (^ instead of **) and the unary operator (~ instead of -) for negation. In this assignment, the exponential operator will be denoted **by** ^. For example, 2^3→8 and 3^2→9.
- The syntax is:
  - infix_expression: term (operator term)*
  - term: number | unary_operator term | '(' infix_expression ')'
- For example, infix ~3 ^ 2 will produce [3, 2, '^','~'] in array for postfix, which evaluates into -9: (~3)^2 → [3,'~',2,'^'] evaluates into 9. You can use builtin split().
- The operator precedence is exactly the same as in math. The exponentiation

operator has the second highest precedence (The exponentiation operator has higher precedence than the * or /. For example, 2*3^2 = 2*9 = 18 **not** 6^2=36), and the unary operator ~ has the same precedence as the exponentiation ^, but are evaluated in right-to-left order (-3 ^ 2 = -9, 3 ^ -1 = ⅓). This is because the exponentiation operator associates from right to left while the other binary operators (+,-,*,/) associate left to right. For example: 2^3^2 = 2^(3^2) = 2^9 = 512 **not** (2^3)^2= 8^2=64. Parentheses has the highest precedence: (-3)^2=9.

- Therefore, it is recommended that you push operators to a stack, and that you pop operators from the stack before pushing an operator with lower precedence onto the stack or when the end of the expression has been reached.
- **Every class and function must come with a brief purpose statement in its docstring. In the signature section of the docstring (Args: and Returns:) you should explain the arguments and what is returned by the function or method (I think you have already been doing this).**
- You must provide test cases for all functions.
- Use descriptive names for data structures and helper functions. You must name your files and functions (methods) as specified below.
- You will not get full credit if you use built-in functions beyond the basic built in operations unless they are explicitly stated as being allowed. If you are in doubt ask me or one of the student assistants.
- Note that when your program creates a stack, the syntax should look like name_of_stack = StackLinked(). This means that you only need to import StackLinked class from project 1.
- sample_infix_expressions.txt provided for your convenience.
- You can use Python built-in data construct dict {} to build a dictionary of operators. Read https://docs.python.org/3/tutorial/datastructures.html#dictionaries

2. Functions (90 points: 80 points for correct implementations, 10 points for coding style)

Implement the following three functions in **exp_eval.py**:

1. infix_to_postfix(infix_expr) to convert an infix expression into a postfix expression.**30 points. Optional:** If the expression is invalid, raise Python built-in SyntaxError. You may create a helper function which helps checking the validity of expressions.10 extra points
2. postfix_eval(postfix_expr) to evaluate a postfix expression into a value. Use the postfix_valid function described below to check the validity of the expression. If the expression is invalid, raise Python built-in SyntaxError. The postfix_eval function should raise python built-in ZeroDivisionError if a divisor is 0. **30 points**
3. postfix_valid(postfix_expr) to test for an invalid postfix expression. You may assume that what is passed in is a string that only contains numbers and operators. These are separated into

valid tokens by spaces so you can use split and join as necessary. The focus is on whether the string had the correct number and position of operators and operands (balanced). This function should return true if the expression is valid and False if it is not valid. **30 points**

3. Tests (10 points)

• Create **exp_eval_testcases.py and** write sufficient tests using unittest to ensure full functionality and correctness of your program. Use infix expressions provided in sample_infix_expressions.txt, which should be converted to postfix expressions in sample_postfix_expressions.txt, as your test cases. **You do not need to provide test cases for your stack since you did that for project 1**.

• Make sure that your tests test each branch of your program and any edge conditions. You do not need to test for correct input in the assignment, other than what is specified above.

• You may assume that when infix_to_postfix(infix_expr) is called that infix_expr is a well formatted, correct infix expression containing only numbers and the specified operators and the tokens are space separated. However, the expression may not be valid / balanced (i.e. invalid / unbalanced parentheses and too many operand or too few operands). If the expression is invalid, raise Python built-in SyntaxError. You may use the python string functions .split and .join

• Examples of invalid expression would be “3 5” or “ 3 * +” or “” . The expression will be invalid due to the number or position of the operators or the numbers. **It will not be due to including some other token, say “X”**

• postfix_eval() should raise python built-in ZeroDivisionError if a divisor is 0.

4. Submission

Zip **three** files (**stacks.py from your project 1**, **exp_eval.py** and **exp_eval_testcases.py**) into one zip file, and **name the zip file as <calpoly_username>_project2.zip (replace <calpoly_username> with your cal poly username)**. Submit the zip file to PolyLearn.