Project 2: Evaluating Expressions using stacks

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

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 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 3.9 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 Abstract Data Type Stack. Your programs should work with either your implementation based on a linked data structure and one based on Python's list data type. See the text and Lab 2 for further information on the ADT Stack.

Notes:

- Expression will only consist of numbers (integers, reals, positive or negative) and the five operators separated by spaces. You may assume a capacity of 30 will be sufficient for any expression that your programs will be required to handle.
- 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. In this assignment, the exponential operator will be denoted **by** ^. For example, 2^3→8 and 3^2→9.
- The exponentiation operator has higher precedence than the * or /. For example, 2*3^2 = 2*9 = 18 **not** 6^2=36
- Also, the exponentiation operator associates from right to left. The other operators (+,-,*,/) associate left to right. Think carefully about what this means. For example: 2^3^2 = 2^(3^2) = 2^9 = 512 **not** (2^3)^2= 8^2=64
- **Every class and function must come with a brief purpose statement in its docstring. In separate comments you should explain the arguments and what is returned by the function or method.**
- You must provide test cases 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.

2. Functions

The following bullet points provide a guide to implement some of the data structures and individual functions of your program. Start by downloading templates to be used

- exp_eval.py (contains infix_to_postfix(infix_expr) and postfix_eval(postfix_expr)
- exp_eval_testcases.py

from PolyLearn and use these are starting points for your project.

```
def infix_to_postfix(infix_expr):
"""Converts an infix expression to an equivalent postfix expression"""

"""Input argument:  a string containing an infix expression where tokens are
   space separated.  Tokens are either operators {+ - * / ^} or numbers
   Returns a string containing a postfix expression """
```

Use the split function to convert the input to a list of tokens

```
def postfix_eval(postfix_expr):
"""Evaluates a postfix expression"""

"""Input argument:  a string containing a postfix expression where tokens
   are space separated.  Tokens are either operators {+ - * / ^} or numbers"""
```

3. Tests

- Write sufficient tests using unittest to ensure full functionality and correctness of your program. **Do not provide test cases for you stack since you did that for Lab 2**.
- 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. 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 a ValueError if a divisor is 0.

4. Submission

Submit two files to PolyLearn: **exp_eval.py** and **exp_eval_testcases.py**

5. Modifications

1. Write a separate function postfix_valid(string) to test for an invalid postfix expression. As above you may assume that what is passed in 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. This function should return true if the expression is valid and False if it is not valid.
2. You may now assume that the postfix_eval() function will always be called with a valid expression. The empty string will be considered invalid.
3. There is now a Stack class posted using an array implementation. The graders will use this in testing your program. Make sure that your program imports the file and the StackArray class appropriately.
4. Note that when your program creates a stack the syntax should be **name_of_stack = StackArray(30)**
5. There is now a template file posted for your testcases. Use this and submit your tests in a file with the same name.