Project 2: Evaluating Expressions using stacks

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

For this project, you will implement a program to convert infix to postfix and evaluate a postfix (Reverse Polish or RPN) expression. To make the program more versatile, you'll also provide code to convert infix expressions (the kind used in standard arithmetic) and prefix (Polish or PN) expressions to postfix expressions. In this way, your program will be able to evaluate prefix, infix and postfix expressions.

For this assignment, you will use an implementation of the Abstract Data Type Stack. Make use of a stack_array you implemented. 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. 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 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.

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. You can use your stack_array from previous project. (Note if you didn't get full credit for stack_array, contact me!)

- stack_array.py
- 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. 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.

4. Algorithms

Evaluating a Postfix (RPN) Expression

While RPN will look strange until you are familiar with it, here you can begin to see some of its advantages for programmers. One such advantage of RPN is that it removes the need for parentheses. Infix notation supports operator precedence ($*$ and $/$ have higher precedence than $+$ and $-$) and thus needs parentheses to override this precedence. This makes parsing such expressions much more difficult. RPN has no notion of precedence, the operators are processed in the order they are encountered. This makes evaluating RPN expressions fairly straightforward and is a perfect application for a stack data structure, just follow these steps:

- Process the expression from left-to-right
- When a value is encountered:
  - Push the value onto the stack
- When an operator is encountered:
  - Pop the required number of values from the stack
  - Perform the operation
  - Push the result back onto the stack
- Return the last value remaining on the stack

For example, given the expression **5 1 2 + 4 ^ + 3 -** :

Input	Type	Stack	Notes
5	Value	5	Push 5 onto stack
1	Value	1 5	Push 1 onto stack
2	Value	2 1 5	Push 2 onto stack
+	Operator	3 5	Pop two operands (1, 2), perform operation (1+2=3), and push result onto stack
4	Value	4 3 5	Push 4 onto stack
^	Operator	81 5	Pop two operands (3, 4), perform operation (3^4=81), and push result onto stack
+	Operator	86	Pop two operands (5, 81), perform operation (5+81=86), and push result onto stack
3	Value	3 86	Push 3 onto stack
−	Operator	83	Pop two operands (86, 3), perform operator (86−3=83), and push result onto stack
	Result	83	

Converting Infix Expressions to Postfix (RPN)

You can also use a stack to convert an infix expression to an RPN expression via the Shunting-yard algorithm. The steps are shown below. Note that the algorithm is more complex that what was shown in class, because the project will include a power operator.

- Process the expression from left-to-right
- When you encounter a value:
  - Append the value to the RPN expression
- When you encounter an opening parenthesis:
  - Push it onto the stack
- When you encounter a closing parenthesis:
  - Until the top of stack is an opening parenthesis, pop operators off the stack and append them to the RPN expression
  - Pop the opening parenthesis from the stack (but don't put it into the RPN expression)
- When you encounter an operator, o_1:
  - While there is an operator, o_2, at the top of the stack and either

$o_1$is left-associative and its precedence is *less than or equal to* that of o_2, or
o_1 is right-associative, and has precedence *less than* that of o_2

    - Pop o_2 from the stack and append it to the RPN expression
  - Finally, push o_1 onto the stack
- When you get to the end of the infix expression, pop (and append to the RPN expression) all remaining operators

For example, given the expression **3 + 4 * 2 / (1 - 5) ^ 2 ^ 3**:

operator	precedence	associativity
^	high	Right
*	medium	Left
/	medium	Left
+	low	Left
−	low	Left

Input	Action	RPN	Stack	Notes
3	Append 3 to expression	3		
+	Push + onto stack	3	+	
4	Append 4 to expression	3 4	+	
*	Push * onto stack	3 4	* +	* has higher precedence than +
2	Append 2 to expression	3 4 2	* +	
/	Pop *, push /	3 4 2 *	/ +	/ and * have same precedence / has higher precedence than +
(	Push (to stack	3 4 2 *	(/ +	
1	Append 1 to expression	3 4 2 * 1	(/ +	
-	Push - to stack	3 4 2 * 1	- (/ +	
5	Append 5 to expression	3 4 2 * 1 5	- (/ +	
)	Pop stack	3 4 2 * 1 5 -	/ +	Pop and append operators until opening parenthesis; then pop opening parenthesis
^	Push ^ to stack	3 4 2 * 1 5 -	^ / +	^ has higher precedence than /
2	Append 2 to expression	3 4 2 * 1 5 - 2	^ / +	
^	Push ^ to stack	3 4 2 * 1 5 - 2	^ ^ / +	^ is evaluated right-to-left
3	Append 3 to expression	3 4 2 * 1 5 - 2 3	^ ^ / +	
end	Pop entire stack to output	3 4 2 * 1 5 - 2 3 ^ ^ / +		

Converting Prefix Expressions (PN) to Postfix (RPN)

- Read the Prefix expression in reverse order (from right to left)
- When an operand is encountered, push it onto the stack

- When an operator is encountered:
  - Pop two operands/strings from the stack: op1 = pop(), op2 = pop()
  - Create a string by concatenating the two operands/strings and the operator after them: string = op1 + op2 + operator (remember space separation)
  - Push the resultant string back to the stack
- Repeat the above steps until end of Prefix expression
- The one string remaining on the Stack is the resultant Postfix expression

For example, given the Prefix expression: * - 3 / 2 1 - / 4 5 6

Input	**Action**	**Stack**	**Notes**
6	Push '6' onto stack	'6'	Read from right to left
5	Push '5' onto stack	'5' '6'	
4	Push '4' onto stack	'4' '5' '6'	
/	Pop '4', '5', combine with /, push onto stack	'4 5 /' '6'	Keep tokens space separated
-	Pop '4 5 /', '6', combine with -, push onto stack	'4 5 / 6 -'	
1	Push 1 onto stack	'1' '4 5 / 6 -'	
2	Push 2 onto stack	'2' '1' '4 5 / 6 -'	
/	Pop '2','1', combine with /, push onto stack	'2 1 /' '4 5 / 6 -'	
3	Push 3 onto stack	'3' '2 1 /' '4 5 / 6 -'	
-	Pop '3','2 1 /', combine with -, push onto stack	'3 2 1 / -' '4 5 / 6 -'	
*	Pop '3 2 1 / -','4 5 / 6 -', combine with *, push onto stack	'3 2 1 / - 4 5 / 6 - *'	
end	Pop entire stack to output		Result: '3 2 1 / - 4 5 / 6 - *'

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(input_str) is called that input_str 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
- You may assume that when prefix_to_postfix(input_str) is called that input_str is a well formatted, correct prefix expression containing only numbers, the specified operators, and that the tokens are space separated. You may use the Python 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.
- postfix_eval(input_str) should raise a PostfixFormatException if the input is not well-formed. Specifically, it should raise this exception with the following messages in the following conditions:
    - “Invalid token” if one of the tokens is neither a valid operand nor a valid operator.
    - “Insufficient operands” if the expression does not contain sufficient operands.
    - “Too many operands” if the expression contains too many operands.
    - Note: to raise an exception with a message: rasie PostfixFormatException(“Here is a message”)
-

4. Submission

Submit three files to PolyLearn: **stack_array.py**, **exp_eval.py** and **exp_eval_testcases.py**