# Assignment 2: Evaluating Expressions using stacks

## 1 Background You may find this link to be helpful (http://www.openbookproject.net/books/pythonds/BasicDS/InfixPrefixandPostfixExpressions.html#postfix-evaluation )

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 (program fragments) into code that a computer can execute.

The link above has a detailed discussion of this material that you may find helpful before building your implementation. You program must use your own implementations of the Abstract Data Type Stack. Your programs should work with either of the implementations you did for the Lab.

**Note**:

- For this assignment, in addition to the operators (+,-,*, /), 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 that do not involve I/O (Input/Output).
- However, you do not need to provide test cases for functions that only produce output or consume input, but design (or refactor) your code so that the functions that involve I/O are as small as possible and only handle I/O. **Write data definitions, function signatures and purpose statements to get full credit.**
- Use descriptive names for data structures and helper functions. You **must** name your files, classes, and functions (methods) as specified below.
- You will not get full credit if you use built-in functions unless they are explicitly stated as being allowed.

## 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 from Canvas to be used as starting points for your project.

- **exp_eval.py** (contains **infix_to_postfix(infix_expr)and postfix_eval (postfix_expr). DO NOT CHANGE** the signatures of these functions**!**
- **exp_eval_testcases.py**

**note in the following that numbers includes integers and floats**

```
def infixToPostfix(infixexpr):
"""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 with tokens are space separated """
```

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

***PostfixEval.py***

```
def postfixEval(postfixExpr):
"""Evaluates a postfix expression"""

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

def postfix_valid(postfixExpr):
"""Determines if a string of consisting of valid operands and operators"""
"""separated by spaces is a valid postfix expression"""
   # see test cases
   # Input argument:  a string containing a valid tokens that 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. You do not need to provide test cases for you stack since you did that for Lab 2.
- Make sure that your own tests test each branch of your program and any edge conditions. **You do not need to test for correct input in the assignment**. You may assume that when `infixToPostfix(infixexpr)` is called that infixexp is a well formatted, correct infix expression containing only numbers and the specified operators and the tokens are space separated.

## 4. Submission

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