Project 2

Please read the entire document carefully! There are three parts to this assignment. **Make sure to do all three!**

1 Introduction

For this project, you will implement a program to evaluate a postfix expression. To make the program more versatile, you'll also provide code to convert infix expressions (the normal kind to which you're accustomed) to postfix. In this way, your program will be able to evaluate infix and postfix expressions. Many programming languages do something similar to convert expressions into code that is easy to execute on a computer.

Notes:

- To break up a string into "tokens", consider using the `str.split` method.
- To convert a token (string) into a number (float), consider using `try`/`except` blocks along with the `float` function.
- Your functions will support the mathematical operations of `+`, `-`, `*`, `/`, `//`, and `**`.
- That last one, `**`, is Python's exponentiation operator. It has higher precedence than the other operators (e.g., `2 * 4 ** 3 == 2 * 64 == 128`) and is right-associative (e.g., `2 ** 3 ** 2 == 2 ** (3 ** 2) == 2 ** 9 == 512`).
- All the other operators are left-associative (e.g., `2 * 3 // 4 == (2 * 3) // 4 == 6 // 4 == 1`).
- At *no* point should you *ever* be using the Python builtin function `eval`.

2 Array Stack

The first thing we're going to have to do is implement a stack. In class, we talked about how we could use either an array-based structure, or we could use a link-based structure. While they both do work, for consistency in grading, I'm going to have you all implement an array-based stack.

In a file called `array_stack.py`, you will implement these functions (stubs are included in the starting file):

- `empty_stack` This function takes no arguments and returns an empty stack.
- `push` This function takes a stack and a value as arguments and places the value on the "top" of the stack.

  For this project, because we're only using the array-based approach, we should be mutating the given stack and thus our function won't return anything.
- `pop` This function takes a stack as an argument and removes (and returns) the element at the "top" of the stack. If the stack is empty, then this operation should raise an `IndexError`.

  For this project, because we're mutating the stack directly, we only need return the value being popped.
- `peek` This function takes a stack as an argument and returns (without removing) the value on the "top" of the stack. If the stack is empty, then this operation should raise an `IndexError`.
- `is_empty` This function takes a stack as an argument and returns whether or not the stack is empty.
- `size` This function takes a stack as an argument and returns the number of items in the stack.

You should find that you can reuse a lot of your code from your `ArrayList` implementation, but it should be a fair bit simpler. Similar to `ArrayList`, you have the exact same restrictions about which list operations are not allowed.

These should *all* be $O(1)$ (i.e., constant time) operations.

3 Evaluating a Postfix (RPN) Expression

3.1 Algorithm

In a file called `exp_eval.py`, you will implement this algorithm as a function called `postfix_eval`.

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, we 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 the stack
`1`	Value	`5 1`	Push `1` onto the stack
`2`	Value	`5 1 2`	Push `2` onto the stack
`+`	Operator	`5 3`	Pop two operands (`1` and `2`), perform $1 + 2$, and push the result (`3`)
`4`	Value	`5 3 4`	Push `4` onto the stack
`**`	Operator	`5 81`	Pop two operands (`3` and `4`), perform 3^4, and push the result (`81`)
`+`	Operator	`86`	Pop two operands (`5` and `81`), perform $5 + 81$, and push the result (`86`)
`3`	Value	`86 3`	Push `3` onto the stack
`-`	Operator	`83`	Pop two operands (`86` and `3`), perform $86 - 3$, and push the result (`83`)
	Result	`83`	The value left on the stack is the answer

You may (and should) use the Python string method `str.split` to separate the string into tokens.

3.2 Exceptions

You may (and should) assume that a string is always passed to `postfix_eval`. However, that does not mean that the RPN expression will always be valid. Specifically, your function should raise a `ValueError` with the following messages in the following conditions:

- `"empty input"` if the input string is an empty string
- `"invalid token"` if one of the tokens is neither a valid operand not a valid operator (e.g., `2 a +`)
- `"insufficient operands"` if the expression does not contains sufficient operands (e.g., `2 +`)
- `"too many operands"` if the expression contains too many operands (e.g., `2 3 4 +`)

To raise an exception with a message, you will `raise ValueError("your message here")`.

Additionally, if you would divide by zero, your code should raise a `ZeroDivisionError`.

4 Converting Infix Expressions to Postfix

In a file called `exp_eval.py`, you will implement this algorithm as a function called `infix_to_postfix`.

We can (and you will!) also use a stack to convert an infix expression to an RPN expression via the Shunting-yard algorithm. The steps are:

- 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 the 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 on the top of the stack and either:
    - o_2 has greater precedence than o_1, or
    - they have the same precedence and o_1 is left-associative

    Pop o_2 from the stack into the RPN expression.
  - Then 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.

The following table summarizes the operator precedence, from the highest precedence at the top to the lowest precedence at the bottom. Operators in the same box have the same precedence.

Operators	Notes
`**`	Right associative
`*`, `/`, `//`	Left associative
`+`, `-`	Left associative

For example, given the expression

```
3 + 4 * 2 / ( 1 - 5 ) ** 2 ** 3
```

Input	**Action**	**RPN**	**Stack**
`3`	Append `3` to RPN	`3`	
`+`	Push `+` to stack	`3`	`+`
`4`	Append `4` to RPN	`3 4`	`+`
`*`	Push `*` to stack	`3 4`	`+ *`
`2`	Append `2` to RPN	`3 4 2`	`+ *`
`/`	Pop `*`, push `/`	`3 4 2 *`	`+ /`
`(`	Push `(` to stack	`3 4 2 *`	`+ / (`
`1`	Append `1` to RPN	`3 4 2 * 1`	`+ / (`
`-`	Push `-` to stack	`3 4 2 * 1`	`+ / ( -`
`5`	Append `5` to RPN	`3 4 2 * 1 5`	`+ / ( -`
`)`	Pop until `(`	`3 4 2 * 1 5 -`	`+ /`
`**`	Push `**` to stack	`3 4 2 * 1 5 -`	`+ / **`
`2`	Append `2` to RPN	`3 4 2 * 1 5 - 2`	`+ / **`
`**`	Push `**` to stack	`3 4 2 * 1 5 - 2`	`+ / ** **`
`3`	Append `3` to RPN	`3 4 2 * 1 5 - 2 3`	
	Pop entire stack into RPN	`3 4 2 * 1 5 - 2 3 ** ** / +`	

You may (and should) assume that a well formatted, correct infix expression containing only numbers, the specified operators, and parentheses will be passed to your function. You may also assume that all tokens are separated by exactly one space.

You may (and should) use the Python string methods `str.split` (to split the string into token) and `str.join` (to join the RPN expression when you're done).

5 Testing

For each part, you are *required* to achieve 100% test coverage. This means that every line of code in your functions *must* be executed at some point in at least one of your tests.

As discussed in the syllabus, when I grade your code, you will ordinarily receive feedback regarding any tests that fail, *unless* you do not have 100% test coverage. In the event you do not have 100% test coverage, the only feedback you will receive is that you need to do more testing. I don't want you using my grading script to do your testing in the last day.

6 GitHub Submission

Push your finished code back to GitHub. Refer to Lab 0, as needed, to remember how to push your local code.

The files you need to have submitted are:

- `stack_array.py`
  - Your array-based implementation of a stack.
- `exp_eval.py`
  - Your functions for evaluating RPN and for converting from infix to RPN.
- `exp_eval_tests.py`
  - Your tests for the functions mentioned above. You *must* have 100% coverage.