

Important information about plagiarism:

While you are encouraged to discuss lecture topics with your peers, assignments must be your own work and plagiarism of any kind will not be tolerated. Submitted assignments will be automatically cross-checked with all other submissions in the class. Anyone who plagiarizes (i.e., copies code or allows their code to be copied by others) will receive a zero (0) on the assignment and be referred to the department chair for more severe disciplinary action.

In this assignment, we will implement a function to evaluate an arithmetic expression. We will use the algorithm outlined in the PowerPoint slides in Chapter 6. That is, you will use two stacks to keep track things. The function is named 'evaluate', it takes a string as input parameter, and returns an integer. The input string represents an arithmetic expression, with each character in the string being a token, while the returned value should be an integer representing the value of the expression.

Examples:

- `evaluate('1+2*3')` → 7
- `evaluate('1*2+3')` → 5

Download the four files "csc220a3.py", "csc220a3_tester.py", "csc220a3_testdata.py", and "array_stack.py" from Canvas and save them in the same folder.

The files "csc220a3_tester.py" and "csc220a3_testdata.py" are the tester program and the test data file, respectively. The last file (array_stack.py) is the ArrayStack class from the textbook (with minor modification to make it easier to use). Do not modify these three files.

The first file (csc220a3.py) is the file you will work on. Note that you cannot rename this file and you cannot change the name of the function (otherwise the tester will not be able to pick up your implementation).

The file "csc220a3.py" contains three helper functions and a dummy implementation of the evaluate function that returns a hard-coded (and probably) incorrect value:

```
from array_stack import ArrayStack

def do_operation(value_stack, operator_stack):
    """ Perform a single operation.
        Pop operands from value_stack,
        Pop operator from operator_stack,
        Push result back to value_stack.
        Input: value_stack - the value stack
               operator_stack - the operator stack
        Output: None
    """
    operand2 = value_stack.pop()
    operand1 = value_stack.pop()
    operator = operator_stack.pop()
    if operator == '+':
        value_stack.push(operand1 + operand2)
    elif operator == '-':
```

```

        value_stack.push(operand1 - operand2)
    else: # operator == '*'
        value_stack.push(operand1 * operand2)

def get_precedence(operator):
    """ Get the precedence of an operator.
        Input: operator - the operator
        Output: the precedence of the operator
    """
    operators = '+*-$'
    precedences = [2, 1, 1, 0]
    return precedences[operators.index(operator)]

def repeat_operations(value_stack, operator_stack, reference_operator):
    """ Repeat performing operators.
        Input: value_stack - the value stack
               operator_stack - the operator stack
               reference_operator - the reference operator
        Output: None
    """
    while len(value_stack) > 1 and \
        get_precedence(reference_operator) <= get_precedence(operator_stack.top()):
        do_operation(value_stack, operator_stack)

def evaluate(expression):
    return -1

```

The tester program will use the testing data to test your implementation. It will run a total of 100 test cases. If you make no change to the function above, it should produce the following output:

```

Passed: [] - total 0.
Failed: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40,
41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60,
61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80,
81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100]
- total 100.

0 of 100 test cases passed. Score = 0.00 of 15.00

```

However, if you implement the function correctly, the output should look like this:

```

Passed: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40,
41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60,
61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80,
81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100]
- total 100.
Failed: [] - total 0.

100 of 100 test cases passed. Score = 15.00 of 15.00

```

Notes:

1. You can assume the input string is always a valid arithmetic expression, with all numerical values being one-digit integers (1-9). You do not have to check for the correctness of the input expression.
2. Since all values are one-digit integers, there is no need to tokenize the input string.
3. Python string has an `isdigit` method that returns `True` when the string consists of all digit characters (0-9), and `False` otherwise.
4. The only operators that will appear in the expression are “+”, “-”, and “*”. These operators have the usual meaning as they are usually used in programming languages.
5. Python has a built-in function, “`eval`”, which will do exactly what you are asked to do with the “evaluate” function (and more). For obvious reason, you are **NOT** allowed to use this function in your submission. **You will just get a score of zero (0) for the entire assignment if you use Python’s `eval` function in your implementation.**
6. Your program will be tested with similar but different data. The testing results will earn you up to 15 points. This will be your execution score.
7. You earn up to 5 points for styling your program to adhere to the PEP-8 standard. See later for details.
8. There must be exactly one `import` statement in your program; and that statement must be “`from array_stack import ArrayStack`”. (This statement has already been put in the “`csc220a3.py`” file.) Otherwise, 20% of your execution score will be deducted.
9. You are allowed to have three helper functions (`do_operation`, `repeat_operations`, and `get_precedence` – the first two (with minor name changes to conform to PEP-8 standard) are from the slide, and the last one will return the precedence of an operator as an integer value). These functions have been given and you cannot modify them. There can be only one single function (the `evaluate` function) other than the three mentioned functions defined in the “`csc220a3.py`” file. No other helper function(s) is/are allowed. Please note that you also can neither define a nested function inside the `evaluate` function, nor any `lambda` functions. Otherwise, 20% of your execution score will be deducted.
10. You must use the three provided helper functions in your code to qualify for full credits. That is, you should not reinvent the wheel.
11. You must have meaningful documentation (in form of Python comments) to explain what is going on in your implementation. Otherwise, 20% of your execution score will be deducted.
12. Your function cannot print anything to the screen – otherwise, 20% of your execution score will be deducted.
13. You must create exactly two stacks and invoke its methods at least three times in your implementation. Otherwise, 90% of your final score will be deducted.
14. Your implementation should be compatible to Python 3.x.
15. Debugging tips: if your code contains syntax errors, the tester will fail to load with no apparent reason. This will generate a result of failing all test cases. Therefore, you should try to just load your code into Python to make sure it does not have any syntax error preventing the tester to load.
16. The tester contains a function “`run_testcase`”, this function takes a testcase number and run that single testcase. It will produce a more verbose output to help you debug your program.

```
>>> run_testcase(50)
TC 50 failed - expecting 927, got -1 instead.
```

Please note that this function is not executed by default when you load the tester. So, you may need to somehow activate Python in interactive mode to use this function. If you are using Python IDLE,

you are automatically put in interactive mode after loading (and executing) the module. However, this is not the default if you are invoking and running Python from the command line. To allow you to use the function if you load the tester from the command line, you must use the interactive flag when invoking Python:

```
> python -i csc220a3Tester.py
```

After you are done, you can use the `exit()` function to abort the Python session.

About Programming Styles

Although writing correct and fast code is the main goal of programming, equally important is writing code that is easy to read and understand. To achieve this second goal, programmers around the world have established conventions and standards. For Python programs, these conventions and standards are documented in PEP (Python Enhancement Proposal) 8. You can read about PEP-8 at <https://www.python.org/dev/peps/pep-0008>.


PEP-8 defines the conventions and standards that are agreed-upon by most Python programmers around the world. By adhering to PEP-8, you can ensure that your programs are (most likely) well-structured and easily understood by others.

However, checking whether a program adheres to PEP-8 can be difficult. It would be convenient if we had a program that could check for us and point out anything that does not conform to PEP-8. Luckily, we do have such a program.

`pylint` is an open-source program that does just that. By default, it checks whether your program conforms to PEP-8. `pylint` has been installed on most computers in campus laboratories and classrooms. You can also install it on your own computer using the following command (you must have Python installed on your computer to install `pylint`):

```
python -m pip install pylint
```

You can run `pylint` from a Windows® command prompt:

1. Open a Windows command prompt:  (this is the Windows® logo at the lower left corner of the screen) | Windows System | Command Prompt
2. Navigate to the folder that contains your program: `cd <\path\to\your\program>`
 - If you do not know the path to your program, you can find it using File Explorer. Once you find it, click on the address bar and it will show the path.
3. Run the command: `pylint <your_program.py>`

For each instance of code that does not conform with the PEP-8 guidelines, `pylint` will display a message telling you what the problem is. It is, of course, your responsibility to fix the problem, and to re-run the check. The message will look something like this:

```
program.py:39:14: C0326: No space allowed before bracket
```

This means that in the file `program.py`, on line `39`, at character `14`, there is an incompilance (`C0326` is an internal identifier for `pylint`): PEP-8 says there should not be space character(s) before a bracket.

At the end of the output, you will see a score on how compliant your program is. The score starts from 10.0 and each incompilance will result in a score deduction. Therefore, you will never have a score greater than 10, but it is possible to have a negative score.

Note that `pylint` assumes that your Python program is a valid Python program (that is, it is free of syntax errors). If any syntax error is detected during the process, it will NOT output a score.

You can earn up to 5 points for conforming to the PEP-8 guidelines. This will be shown as the style score. Your style score will be calculated as follows (assuming `pylint_score` is the score provided by `pylint`):

```
style_score = 0 if pylint_score < 0 else pylint_score * 0.5
```

If `pylint` did not output a score (due to a syntax error), your style score will also be zero (0).

Submission

Submit only the file “`csc220a3.py`” to Canvas. Due date is Sunday March 24.