Task 1 (10%): File IO (Easy)

For Task 1, you will read in, store and return binary arithmetic expression(s) from a text file. You will implement the `read_expressions_easy` function inside the `calculator.py` file as follows. The `read_expressions_easy` function must:

- input the directory of a text file `file_name` (e.g., `expressions.txt`) where each line of the text file represents a binary arithmetic expression (e.g., `(* 2 (+ 8 7))`), and
- return the list of the binary arithmetic expressions `expr_lists` in the specific format that is provided below.

The format of the returned list `expr_lists` must be as follows. Each element `e` of the returned list `expr_lists` must:

- be a list where each element `e` represents a line of the input file (i.e., in the same order), and
- contain three elements `e_1`, `e_2` and `e_3` such that `e = [e_1, e_2, e_3]` where the first element `e_1` must be an arithmetic operator (i.e., `'+'`, `'-'`, `'/'` or `'*'`), and the last two elements `e_2` and `e_3` must be either ii) a non-negative number (i.e., of type `int` or `float`) and/or ii) a list with the same format as the element `e`.

For example, the binary arithmetic expression `(* 2 (+ 8 7))` would be stored as the list `['*', 2, ['+', 8, 7]]` according to the format specified above.

> **Hint:** You are allowed to use the `eval` function for Task 1.
>
> You must not `import` and/or use any external modules for Task 1.

Task 2 (15%): Expression Evaluation (Iterative)

For Task 2, you will use iterations (i.e., for and/or while loops) to compute the result of a binary arithmetic expression. You will implement the `evaluate_expression_iter` function inside the `calculator.py` file as follows. The `evaluate_expression_iter` function must:

- input a list `expr_list` where `expr_list` represents a binary arithmetic expression that is stored in the format that is previously described in Task 1 (e.g., `['*', 2, ['+', 8, 7]]`), and
- return the number (i.e., of type `float`) that is the result of evaluating the input binary arithmetic expression (e.g., `30.0` for the binary arithmetic expression `['*', 2, ['+', 8, 7]]`).

The result of the input binary arithmetic expression must be computed without using recursion (i.e., only using **loops**):

- based on the structure of the nested parentheses (i.e., the computation of the innermost expression(s) take priority), and
- by applying the first element (i.e., the operator `'+'`, `'-'`, `'/'` or `'*'`) between the second and the third elements of the input list `expr_list` recursively.

For example, the binary arithmetic expression `['*', 2, ['+', 8, 7]]` is equivalent to `['*', 2, 15]` which is equivalent to `30.0`. **Please note that for Task 2, you can still get partial marks if your implementation only works for some fixed maximum depth of the input list `expr_list`.**

Turns out, the use of the `eval`, `exec` or `compile` functions has important security implications!

You must not use or create an alias for the `eval`, `exec` or `compile` functions in your solution for Task 2.

You must not use recursion in your solution for Task 2.

You must not `import` and/or use any external modules for Task 2.

Task 3 (15%): Expression Evaluation (Recursive)

For Task 3, you will re-implement the `evaluate_expression_iter` function using **recursion** (i.e., without using iteration such as for and/or while loops) in the `evaluate_expression_rec` function inside the `calculator.py`. You must clearly label both i) the base case(s) and ii) the recursive relation(s) with appropriate descriptive comment(s) for full marks.

> Turns out, the use of the `eval`, `exec` or `compile` functions has important security implications!
>
> You must not use or create an alias for the `eval`, `exec` or `compile` functions in your solution for Task 3.
>
> You must not use iteration such as `for` or `while` loops in your solution for Task 3.
>
> You must not `import` and/or use any external modules for Task 3.

Task 4 (30%): File IO (Hard)

For Task 4, you will re-implement the `read_expressions_easy` function using **recursion** without using the `eval` function, given the use of the `eval` function might have some serious security consequences! You must clearly label both i) the base case(s) and ii) the recursive relation(s) with appropriate descriptive comment(s) for full marks.

> **Hint:** You should be able to copy and paste your code from Task 1, and simply replace the `eval` function with your own recursive implementation of the function.
>
> You must not use or create an alias for the `eval`, `exec` or `compile` functions in your solution for Task 4.
>
> You must not `import` and/or use any external modules for Task 4.

Task 5 (30%): Unit Test

For Task 5, you will use unit testing to verify the correctness of your program. You will implement the `test_file`, `test_equivalence` and `test_correctness` functions as a part of the `CalculatorTestCase` class inside the `test_calculator.py` file, using assertions and the `unittest` module, as follows.

- Verify the format of the text file that contains the binary arithmetic expression(s) (i.e., using `test_expressions.txt`).
- Verify the equivalence of the file IO functions in Tasks 1 and 4 (i.e., using `read_expressions_easy` and `read_expressions_hard`), and Tasks 2 and 3 (i.e., using `evaluate_expression_rec` and `evaluate_expression_iter`)
- Verify the correctness of the results computed by the `evaluate_expression_rec` and `evaluate_expression_iter` functions (i.e., using `test_expressions.txt` and `test_results.txt`).

Across the `test_file`, `test_equivalence` and `test_correctness` functions, you will need to verify **ten unique necessary conditions of correctness** (i.e., previously equivalently referred to as **properties**) for full marks. Each unique necessary condition that is checked should be clearly labeled with i) a descriptive comment, and ii) an appropriate and unique `Assertion Error` message.

> **Reminder:** You might need to make multiple assertions to check a single necessary condition.
>
> Do not forget to copy and paste your previous implementations of the necessary functions from Tasks 1-4 into the `calculator.py` file.
>
> Aside from `unittest` and `calculator`, you must not `import` and/or use any external modules for Task 5.