Infix Calculator

CSCI 112, Lab 3

File names: Names of files, functions, and variables, when specified, must be EXACTLY as specified. This includes simple mistakes such as capitalization.

Individual work: All work must be your own. Do not share code with anyone other than the instructor and teaching assistants. This includes looking over shoulders at screens with the code open. You may discuss ideas, algorithms, approaches, *etc.* with other students but NEVER actual code. Do not use code written by anyone else, in the class or from the internet.

Documentation: Each file should begin with a docstring that includes your name, the class number and name, the lab number, and a short description of the lab, as well as documentation pertinent to that particular file.

Calculator: Finish the following two problems from the text:

1. Implement a direct infix evaluator that combines the functionality of infix-to-postfix conversion and the postfix evaluation algorithm. Your evaluator should process infix tokens from left to right and use two stacks, one for operators and one for operands, to perform the evaluation.

   Follow the textbook's style. Try to use as much of the code from the textbook as possible. You should be able to simply modify the `infixToPostfix` function using the `doMath` function. Every time the function would output an operator, apply this operator to the top two items on the stack. Think about the order of operands and the order of stacking!

2. Turn your direct infix evaluator from the previous problem into a calculator. This will simply print a prompt, read the input, evaluate the arithmetic expression, print the result, and loop. Here's mine in action:

   ```
   >>> repl()
   Geoff's Amazing Calculator! >>> 2 + 2
   4
   Geoff's Amazing Calculator! >>> 3 + 4 * 5
   23
   Geoff's Amazing Calculator! >>> 3 * 4 + 5
   17
   Geoff's Amazing Calculator! >>> quit
   >>>
   ```

   `repl` stands for "read, evaluate, print, loop"

Tokenizing: Instead of using single-character numbers, as in the text, use my tokenizing function found in `tokens.py`. This allows the use of floats, too. It uses `split()` to separate tokens, so all tokens must be separated by spaces, including operators and parentheses. For example:

```
>>> tokenize('( 44 + -33.3 ) * 2 - 4')
['(', 44, '+', -33.3, ')', '*', 2, '-', 4]
```

Operators: You must support addition, subtraction, multiplication and division. Multiplication and division have higher precedence than addition and subtraction. Otherwise everything is left associative.

Turn in: A file named `calculator.py`, with a function called `evaluate` with the following behavior, which does infix evaluation, for example:

```
evaluate('2 + 2')    =>    4
```

and a function called `repl` which starts a read-eval-print loop.

A unit test module called `calc_test.py`, that tests the major functionality of your calculator functions.

Zip these functions, together with `tokens.py` and any other modules you built, in a folder called `csci112lab03yourname` zip and submit to canvas.

Optional additions:

- Write a better tokenizer so that spaces are not required where they are not needed, for example, `(2+2)`
- Add error checking to the tokenizer, so that the user gets meaningful feedback from expressions like these: `3 + )`, `4 + * 5`, and `( 3 ( 4 + 5 ))`, and the calculator continues to work, rather than stopping with an error.
- Add some unary operators, such as `sin` and `cos` and `exp`
- Add the `**` operator. Note that this has higher precedence than all the others, and is also right associative. You will need to think through how to handle the operator stack for this one, and make the necessary additions to the algorithm outlined in §4.9.2.