CSCI 111, Lab 5
RPN Calculator

Due date: Midnight, Tuesday, October 11, on Canvas. No late work accepted.

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.

RPN: Reverse Polish Notation is a way of expressing algebraic expressions without using parentheses. Instead of coming between operands, the operator comes after the operands.

4 + 5	becomes	4 5 +
(4 + 5) * 3	becomes	4 5 + 3 *
4 + (5 * 3)	becomes	4 5 3 * +
sin(123)**2 + cos(123)**2	becomes	123 cos 2 ** 123 sin 2 ** +

There are a lot of examples online. You should make sure you understand how it works before attempting this lab.

RPN calculators: Because no parentheses are required, the interface to a calculator was much simpler. HP calculators in the early years, and some even today, used RPN.

Start a module `rpn.py` which will need to import `math`:

```
1 import math
```

We will put the following functions into this module.

Parse numbers: Before we can process a line of RPN, we need to parse it. This involves breaking it into tokens (easy, tokens are separated by whitespace, so use `split`), and then convert the numeric strings into actual numbers. For instance, the string `'2.3e2'` has to be translated into the number `230.0`.

Parse integers: We break this down into several simple parts. Integers are easy: they consist of an optional `'+'` or `'-'`, followed by any number of digits in `'0123456789'`. Each digit represents a quantity ten times bigger than the one to its right, so we can process an integer as follows:

1. Set variable `sign` to `1`
2. If the first character is `'+'`, remove it.
3. If the first character is `'-'`, remove it and set `sign` to `-1`
4. Set variable `accumulator` to `0`
5. For each of the remaining digits, `d`:

   (a) Multiply `accumulator` by `10`
   (b) Add `ord(d) - ord('0')` to `accumulator`
6. Return `sign * accumulator`

Note that the empty string should parse as the integer `0`. This will come in handy.

Write a function `parse_integer` that does this:

```
>>> parse_integer('321')
321
>>> parse_integer('-555')
-555
>>> parse_integer('')
0
```

Parse floats: Floats are more difficult because they have more parts. A typical float, `+123.456e-78` for example, looks like:

$$\underbrace{+123}_{whole}\underbrace{.}_{dot}\underbrace{456}_{fraction}\underbrace{e}_{e}\underbrace{-78}_{exponent}$$

So, really it consists of three integers, `whole`, `fraction`, and `exponent`. The `e` and the `exponent` are optional.

In python either the `whole` or the `fraction` is optional, but you can't omit both! We will let both of them be optional for simplicity's sake.

So, now we have a simple algorithm for integers:

1. Set the variable `exponent` to `0`
2. If `'e'` is in the string:
   (a) Split on `'e'`
   (b) Parse the second part as an integer
   (c) Set the variable `exponent` to this integer
   (d) Set the string to the first part
3. If `'.'` is in string:
   (a) Split the string on `'.'`
   (b) Parse the first part as an integer
   (c) Assign that to `whole`
   (d) Assign the length of the second part to `n`
   (e) Parse the second part as an integer
   (f) Assign that to `fraction`
   (g) Divide `fraction` by $10^{\texttt{n}}$
   (h) If `whole` is negative, multiply `fraction` by `-1`.
4. If `'.'` is not in the string:

(a) Parse the string as an integer
(b) Set `whole` to this integer
(c) Set `fraction` to `0`

5. Return `(whole + fraction) * (10 ** exponent)`

Write a function `parse_float` that does this:

```
>>> parse_float('1e1')
10.0
>>> parse_float('-123e-123')
-1.2300000000000001e-121
>>> parse_float('345.678e4')
3456780.0
>>> parse_float('-33.44e-2')
-0.3344
>>> parse_float('.33e2')
33.0
>>> parse_float('.')
0.0
```

Parsing numbers: A string in our calculator could be either an integer or a float. Write a function `parse_number` that will take a string of either kind and return either an integer or a float. Examples:

```
>>> parse_number('1e1')
10.0
>>> parse_number('10')
10
```

We will assume that all input is either a valid integer or a valid float. How can you tell them apart? Hint: floats must have either `'.'` or `'e'` in them.

Note: our numbers are slightly different from Python's numbers. For example, what does Python do with `'.'`? How about `'0003'`? Our parser will have different answers.

Note: once again, we will assume the input is perfect. We will not check for errors. So if we try to parse a number out of the string `'1.0xxx99'` the program will crash. (What will be the error?)

Keywords: There will be certain strings that are keywords in our RPN calculator language. They are:

```
['pop', 'clr', '+', '-', '*', '/', '**', 'sin', 'cos', 'sqrt']
```

Write a boolean function `is_keyword` that determines whether or not a string is a keyword.

Tokenize: We need to take a line of input and convert it to tokens. The tokens of our language are integers, floats, and keywords. The first step is easy, since these are all separated by whitespace in our RPN calculator. Keywords are represented by strings, but the strings for numbers should be converted to numbers of the right type. Write a function `tokenize` that takes a string and returns a list of tokens. For example:

```
>>> tokenize('3 99.9 + 1e5 *')
[3, 99.9, '+', 100000.0, '*']
```

Note: We will assume that all input is perfect. You don't have to check for things that are not keywords or numbers.

The stack: Our interpreter will use a global variable called `The_Stack`. This will behave like a **stack** data structure, a particular kind of list where you can *only* remove one item from the end, or instert one item on the end. In other words, the **only** operations allowed on a stack `s` are: `s.pop()` and `s.append(item)`. Other than initializing it to an empty list, and printing it, you are not allowed to do anythgin else to a stack.

The calculator: An RPN calculator works as follows:

1. Read a token.
2. If the token is a number, push it onto the stack.
3. If the token is an operator:
   (a) If it is `'+'`:
      i. Pop two items from the stack
      ii. Add them up
      iii. Push the sum onto the stack
   (b) If it is `'-'` or `'*'` or ...,
      i. behave similarly to `'+'` (be careful about argument order for `'-'` *etc.*)
   (c) If it is `'sin'` or `'sqrt'` or ...
      i. behave similarly to `'+'` but with only one item from the stack.
   (d) If it is `'pop'`:
      i. Pop the tack
   (e) If it is `'clr'`:
      i. Remove all items from the stack
4. Loop to step 1

Write a function `rpnline` that will take a line of text as a string and a stack as arguments. It then tokenizes the string, and proceeds to handle all the tokens in the manner above.

Before handling each token, have the procedure print the stack and the remaining tokens. It really helps understanding and debugging.

Once again, we are not concerned with errors. If someone enters the string `'99 +'` starting with an empty stack, then the program will crash. (What would be the error?)

When out of tokens, the function prints the stack one more time before terminating.

Example:

```
>>> The_Stack = []
>>> rpnline('3 4 1e1 + *', The_Stack)
Stack: [] Tokens: [3, 4, 10.0, '+', '*']
Stack: [3] Tokens: [4, 10.0, '+', '*']
Stack: [3, 4] Tokens: [10.0, '+', '*']
Stack: [3, 4, 10.0] Tokens: ['+', '*']
Stack: [3, 14.0] Tokens: ['*']
[42.0]
```

REPL: You can now build your own REPL, like IDLE's, but for your RPN calculator:

```
def repl(stack):
    while True:
        line = input('RPN>>> ')
        if line == 'exit':
            break
        rpnline(line,stack)
```

You can run it like this:

```
>>> repl(list())
RPN>>> 4 5 *
Stack: [] Tokens: [4, 5, '*']
Stack: [4] Tokens: [5, '*']
Stack: [4, 5] Tokens: ['*']
[20]
RPN>>> 2 2 2 2 ** ** **
Stack: [20] Tokens: [2, 2, 2, 2, '**', '**', '**']
Stack: [20, 2] Tokens: [2, 2, 2, '**', '**', '**']
Stack: [20, 2, 2] Tokens: [2, 2, '**', '**', '**']
Stack: [20, 2, 2, 2] Tokens: [2, '**', '**', '**']
Stack: [20, 2, 2, 2, 2] Tokens: ['**', '**', '**']
Stack: [20, 2, 2, 4] Tokens: ['**', '**']
Stack: [20, 2, 16] Tokens: ['**']
[20, 65536]
RPN>>> exit
```

Notice I didn't clear the stack between operations. That's OK; you might want to make several computations, leave all the results on the stack, and then do further computations with the results.

File processing: You can also put RPN programs in a file and run them all like this:

```
def runfile(fname,stack):
    fin = open(fname)
    for line in fin:
        line = line.strip()
        if line == 'exit':
            break
        print('INPUT:', line)
        rpnline(line,stack)
```

Turn in: Turn in the single module `rpn.py`. I will run it on a new input file during grading, so be sure yours can handle all proper input, using any of the operators in the list of keywords.

Sample output: Here is the output from my program on a sample input file, similar to the one I will use for grading:

```
INPUT: 009 9 *
Stack: [] Tokens: [9, 9, '*']
Stack: [9] Tokens: [9, '*']
Stack: [9, 9] Tokens: ['*']
[81]
INPUT: clr
Stack: [81] Tokens: ['clr']
[]
INPUT: 5 4 3 - -
Stack: [] Tokens: [5, 4, 3, '-', '-']
Stack: [5] Tokens: [4, 3, '-', '-']
Stack: [5, 4] Tokens: [3, '-', '-']
Stack: [5, 4, 3] Tokens: ['-', '-']
Stack: [5, 1] Tokens: ['-']
[4]
INPUT: clr
Stack: [4] Tokens: ['clr']
[]
INPUT: 5 4 - 3 -
Stack: [] Tokens: [5, 4, '-', 3, '-']
Stack: [5] Tokens: [4, '-', 3, '-']
Stack: [5, 4] Tokens: ['-', 3, '-']
Stack: [1] Tokens: [3, '-']
Stack: [1, 3] Tokens: ['-']
[-2]
INPUT: clr
Stack: [-2] Tokens: ['clr']
[]
INPUT: 3 2 ** 4 2 ** + sqrt
Stack: [] Tokens: [3, 2, '**', 4, 2, '**', '+', 'sqrt']
Stack: [3] Tokens: [2, '**', 4, 2, '**', '+', 'sqrt']
Stack: [3, 2] Tokens: ['**', 4, 2, '**', '+', 'sqrt']
Stack: [9] Tokens: [4, 2, '**', '+', 'sqrt']
Stack: [9, 4] Tokens: [2, '**', '+', 'sqrt']
Stack: [9, 4, 2] Tokens: ['**', '+', 'sqrt']
Stack: [9, 16] Tokens: ['+', 'sqrt']
Stack: [25] Tokens: ['sqrt']
[5.0]
INPUT: clr
Stack: [5.0] Tokens: ['clr']
[]
INPUT: 123 sin 2 ** 123 cos 2 ** +
Stack: [] Tokens: [123, 'sin', 2, '**', 123, 'cos', 2, '**', '+']
Stack: [123] Tokens: ['sin', 2, '**', 123, 'cos', 2, '**', '+']
Stack: [-0.45990349068959124] Tokens: [2, '**', 123, 'cos', 2, '**', '+']
Stack: [-0.45990349068959124, 2] Tokens: ['**', 123, 'cos', 2, '**', '+']
```

```
Stack: [0.21151122074847095] Tokens: [123, 'cos', 2, '**', '+']
Stack: [0.21151122074847095, 123] Tokens: ['cos', 2, '**', '+']
Stack: [0.21151122074847095, -0.8879689066918555] Tokens: [2, '**', '+']
Stack: [0.21151122074847095, -0.8879689066918555, 2] Tokens: ['**', '+']
Stack: [0.21151122074847095, 0.7884887792515292] Tokens: ['+']
[1.0]
INPUT: clr
Stack: [1.0] Tokens: ['clr']
[]
INPUT: 3.14159265 2 / sin
Stack: [] Tokens: [3.1415926499999998, 2, '/', 'sin']
Stack: [3.1415926499999998] Tokens: [2, '/', 'sin']
Stack: [3.1415926499999998, 2] Tokens: ['/', 'sin']
Stack: [1.5707963249999999] Tokens: ['sin']
[1.0]
INPUT: clr
Stack: [1.0] Tokens: ['clr']
[]
INPUT: 3.14159265 2 / cos
Stack: [] Tokens: [3.1415926499999998, 2, '/', 'cos']
Stack: [3.1415926499999998] Tokens: [2, '/', 'cos']
Stack: [3.1415926499999998, 2] Tokens: ['/', 'cos']
Stack: [1.5707963249999999] Tokens: ['cos']
[1.7948967369654108e-09]
INPUT: clr
Stack: [1.7948967369654108e-09] Tokens: ['clr']
[]
INPUT: 1e10 1e-10 *
Stack: [] Tokens: [10000000000.0, 1e-10, '*']
Stack: [10000000000.0] Tokens: [1e-10, '*']
Stack: [10000000000.0, 1e-10] Tokens: ['*']
[1.0]
INPUT: clr
Stack: [1.0] Tokens: ['clr']
[]
INPUT: -2.2e-3 2.2e-3 +
Stack: [] Tokens: [-0.0022, 0.0022, '+']
Stack: [-0.0022] Tokens: [0.0022, '+']
Stack: [-0.0022, 0.0022] Tokens: ['+']
[0.0]
```