CS417 Lab #15

Getting Started

Begin the lab by downloading these starting files:

- `postfix.py`
- `stack.py`
- `console.py`

Description

You are given an implementation of a stack. You are also given skeleton code, for a postfix calculator. Your job is to complete the postfix calculator.

Postfix Evaluation

You are used to working with infix expressions, where operators occur **between** operands. For example:

```
1 * 2 + 3.14 / 9
```

In a postfix expression, the operator comes **after** the operands. For example:

```
1 2 * 3.14 9 / +
```

Postfix expressions are useful, because it's very easy to evaluate one, using a stack of values. Here is the algorithm:

```
for each token in the expression:
    if the token is an operand:
        push the operand
    else (it's an operator):
        pop the right operand
        pop the left operand
        apply the operator
        push the result
```

The `stack` module

You have a module `stack.py`, which defines a class called `Stack` (note the capitalization). This class implements these methods:

- `__init__` constructs an empty stack
- `push(x)` pushes `x` on top of the stack

- `pop()` pops (removes) the top value from stack, and returns that value.
- `top()` returns the value at the top of the stack, but does not remove it (stack is unchanged).
- `empty()` returns True/False if the stack is/isn't empty.
- `__len__` returns the number of values in the stack. Invoked by the python `len()` function.
- `__str__` returns a printable string version of the stack's contents. The top of the stack is right-most.

Open `stack.py` and look at the code. Look at the `main()` function, which shows how it's used:

- `s = Stack()` creates a stack object
- `s.push(10)` pushes the value 10
- `x = s.pop()` pops the top value, which goes into `x`

Notice that the `pop()` method may raise an exception, if the stack is empty.

Your Tasks

1. Open the module `postfix.py`, and run it. Type this sample input:

   ```
   1 2 +
   ```

   You **MUST** leave spaces between the tokens; otherwise the line can't be easily split into tokens.

   You will get an error, because you didn't import the `stack` module. Add this line to the top of your file:

   ```
   from stack import Stack
   ```

   By using `from`, we don't have to mention the module's name. Instead of writing `s = stack.Stack()`, we simply write `s = Stack()`.

   Run `postfix.py` again, and verify that the module was imported.

---

2. The function `eval_postfix` calls `is_operator` to identify the token. Implement `is_operator(token)`. If `token` occurs in `"+-*/="`, return `True`. Otherwise return `False` (use the `in` operator).

---

3. Take care of operators. If your token is an operator, pop the stack twice: once into `right_operand`, and once into `left_operand`:

   ```
   right_operand = s.pop()
   left_operand = s.pop()
   ```

   Then, write a four-way `if-elif-elif-elif` block of code that adds, subtracts, multiplies, or divides the two operands, into a `result`. Push the result:

   ```
   s.push(result)
   ```

4. Now, handle operands. If `token` is not an operator, it's an operand. Push the token.

5. If the expression is valid, then, after all the tokens have been processed, there should be a single value on the stack. It's the expression's value.

   Pop the stack into a result, and return that result.

```
value = s.pop()
return value
```

6. Run the program, and enter this expression:

```
    2 3 *
```

   You should get an error. What happened? You're multiplying two strings! We forgot to turn the operands into numbers!

   Go back to Task 4, and convert the token into a `float`, and *then* push it. Run the program again, and you should see `6.0`

7. (Error handling) Try this expression:

```
    1 2 + -
```

   Your program will die, with an empty-stack exception. That's because there are too many operators. The program should not die; it should report the error.

   The problem is with Task 3. The stack may be empty. So, add an `if s.empty():` before each `s.pop()`. If the stack is empty, `return "Too many operators"`. Else, `pop` normally.

8. (Error handling) Now, try this input:

```
    1 2 3 +
```

   Your program shouldn't die, but clearly this is a bad expression. It has too many operands. This error is detected at the end (Task 5). Before you pop the result, check the length of the stack using `if len(s) > 1:`. If the stack has more than 1 value, `return "Too many operands"`

9. [10% Bonus] Implement the `"="` assignment operator. Such an operator will arise if you use variables, such as in this infix expression:

```
    a = 1 + 2
```

   which is converted into postfix thus:

```
a 1 2 + =
```

Adding this feature requires **many** changes:

- You will need a symbol table to store the values of all the variables. The symbol table should be created in the `main` function:

```
variables = dict()
```

and should be passed to `eval_postfix`, which now expects **two** arguments.

- When you get a `"="` operator, its action should be to modify a variable:

```
variables[left_operand] = right_operand
```

- When you get an operand, it may be a number, or it may be a variable name. You need to check this. Try converting it into a `float`, and catching an exception:

```
try:
    value = float(token)
except ValueError:
    value = token  # not a number, assume it's a name
s.push(value)
```

- You may get expressions like this one:

```
a b 1 + =
```

which comes from the infix expression `a = b + 1`. In this expression, the stack may hold the string `"b"`, not the value of the variable `b`. To handle this, you will have to add code that checks `left_operand` and `right_operand`, and retrieves their value, like this:

```
if type(left_operand) != float:
    left_operand = variables[left_operand]
```

Turning in your work

To submit your work, go to mycourses.unh.edu, find cs417, and the lab, and upload `postfix.py`. Submit whatever you have completed, at the end of the lab session. You can submit again until midnight, with no lateness penalty.