CSC148 Summer 2018: Lab 3

Introduction

The goals of this lab are:

- To get you familiar with how stacks work
- To get you familiar with how queues work
- To give you practice solving problems using stacks (using add/remove/is_empty)
- To give you practice solving problems using queues (using add/remove/is_empty)

Don't hesitate to make use of other resources for this lab, including the course notes, your TAs, instructor, or other students.

General Lab Notes

1. Make sure you have lab_pyta.txt downloaded and placed in the directory (or directories) where you'll be working.
2. To use PythonTA, include the following code (if you already have a main block, just add the body to the end of it):

```
if __name__ == '__main__':
    import python_ta
    python_ta.check_all(config="lab_pyta.txt")
```

Your lab_pyta.txt should be in the same folder as the .py files you're running. PythonTA will raise errors regarding style, specifying the lines you need to fix. You should get familiar with what the errors mean, and how to fix them: this will be important for your exercises and assignments.

Getting Started

This lab will have you using Stacks and Queues. Download the adts.py from lecture or write your own Stack and Queue classes. As a reminder: **Do not make a file named queue.py or stack.py.** If you do so, PythonTA will break, and parts of your PyCharm might as well.

Warm-up Problem

When given a Stack or a Queue, you should be able to trace through and know which items get added or removed. To make sure you understand this: Suppose we take the following steps for a Queue. Keep track of the state of the stack at each point, as well as what gets returned when something is removed from it:

1. Add "E"
2. Add "S"
3. Remove an item
4. Add "U"
5. Add "E"
6. Add "A"
7. Add "T"

8. Remove an item
9. Remove an item
10. Add "U"
11. Add "K"
12. Add "C"
13. Remove an item
14. Remove an item
15. Add "Q"
16. Keep removing items until it's empty

After you're gone through this for a Queue, try the same operations but with a Stack.

Writing the Algorithm from Lecture (list_container)

Write a function called list_container that takes a list and a Container (either a Stack or a Queue) and does the following.

1. Add all of the items from the list into the Container.
2. Do the following until the Container is empty:
    a. Remove an item from the Container.
    b. If that item is not a list, print it out.
    c. If it is a list, add all of its items into the Container.

This is the algorithm from lecture. Your implementation must only use the add, remove, and is_empty methods, and should not access the private attributes of our Stack or Queue. We should be able to change the implementation of the Stack and Queue and have the algorithm still work correctly.

Call your function on the example list from lecture: [1, [2, 3, [4, 5], 6], 7, [[8], 9]]

Make your own examples too, to make sure it works correctly. When using a Stack, your function should always print out the elements in reverse order, ignoring nesting. When using a Queue, it should return the elements in "level order".

Returning a List of Levels using a Queue (list_nestedness)

Make a copy of the function above and rename it list_nestedness(), but make it take only a Queue. In this part, you will be returning a list of lists, with each list containing all of the items in each 'level' of nestedness. For example, passing in the list [1, [2, 3, [4, 5], 6], 7, [[8], 9]] should give us [[1, 7], [2, 3, 6, 9], [4, 5, 8]]. That is, the list at index i should contain elements nested within i sublists.

The steps to this should be the same, but you'll want to add a few extra lines (which have been bolded.)

1. Add all of the items from the list into the Queue.
    a. Assume that none of the items being added are called "END".
2. **Add the item "END" to the Queue.**
3. **Create an empty list called "level"**
4. **Create an empty list called "all_levels"**

5. Do the following until the Queue is empty:
    a. Remove an item from the Queue.
    b. **If that item is "END" append "level" to "all_levels" and set level to an empty list.**
        i. **If the Queue is not empty, add "END" to the Queue.**
    c. If that item is not a list **or "END", add it to the list "level".**
    d. If it is a list, add all of its items into the Queue.

Stacks to Queues

In this part, you'll be using 2 stacks in order to implement a Queue. First: Create a class called StackQueue, with 2 stacks as attributes (we'll call one 'add_to' and the other 'remove_from').

When we add to a StackQueue, we add the item in to the add_to attribute.

When we remove from a StackQueue, we want the item at the bottom of add_to. To do this:

1. Remove an item from add_to and add that item to remove_from.
2. Repeat this until add_to is empty.

The item we want to return is at the top of remove_from. So remove that item from remove_from and store it in another variable.

Now re-add everything from remove_from back to add_to, so that add_to is in the right order.

Remember to also implement the is_empty() method.

Try to trace through a few examples to convince yourself that this works correctly. You could try to pass this StackQueue into the list_container() function and list_nestedness() function to make sure the StackQueue works properly.

For additional practice, consider the following problems:

- If we wanted to make our remove() method faster, what would we have to change?
    - Try to implement add() such that remove() only need to remove once from one of the Stacks.
- Can you implement a Stack using Queues? If so, how?

Remember: In none of these examples should you ever access the private attributes of a Stack or Queue that you're using (aside from the private attributes in the class that you're writing, anyways). To make sure you're not doing this:

- Modify the Stack and Queue is adts.py so they work in the opposite way. For example, make the Stack add to the front and remove from the front, and make the Queue add to the front and remove from the end. Everything you made in this lab should still work properly.