Lab 4: Abstract Data Type

1) Stack review

Open *mystack.py* and first review the given stack implementation and the *size* function we discussed in lecture.

Complete the following tasks.

Note that you should write these as top-level functions, not stack methods.

While you may use a temporary stack (as we did in lecture for *size*),do not use any other Python compound data structures, like lists.

1. Write a function that takes a stack of integers and removes all of the items which are greater than 5. The other items in the stack, and their relative order, should remain unchanged.

2. Write a function that takes a stack and returns a new stack that contains each item in the old stack twice in a row. We'll leave it up to you to decide what order to put the copies into in the new stack.

   Note that because the docstring doesn't say that the old stack will be mutated, the old stack should remain unchanged when the function returns.

```
"""CSC148 Lab 4: Abstract Data Types

=== CSC148 Winter 2020 ===
Department of Computer Science,
University of Toronto

=== Module Description ===
In this module, you will write two different functions that operate on
a Stack.
Pay attention to whether or not the stack should be modified.
"""
from typing import Any, List


```

```
14  ##############################################################
15  # Task 1: Practice with stacks
16  ##############################################################
17  class Stack:
18      """A last-in-first-out (LIFO) stack of items.
19
20      Stores data in a last-in, first-out order. When removing an item
    from the
21      stack, the most recently-added item is the one that is removed.
22      """
23      # === Private Attributes ===
24      # _items:
25      #     The items stored in this stack. The end of the list
    represents
26      #     the top of the stack.
27      _items: List
28
29      def __init__(self) -> None:
30          """Initialize a new empty stack."""
31          self._items = []
32
33      def is_empty(self) -> bool:
34          """Return whether this stack contains no items.
35
36          >>> s = Stack()
37          >>> s.is_empty()
38          True
39          >>> s.push('hello')
40          >>> s.is_empty()
41          False
42          """
43          return self._items == []
44
45      def push(self, item: Any) -> None:
46          """Add a new element to the top of this stack."""
47          self._items.append(item)
48
49      def pop(self) -> Any:
50          """Remove and return the element at the top of this stack.
51
52          Raise an EmptyStackError if this stack is empty.
53
54          >>> s = Stack()
55          >>> s.push('hello')
56          >>> s.push('goodbye')
57          >>> s.pop()
58          'goodbye'
59          """
60          if self.is_empty():
61              raise EmptyStackError
62          else:
63              return self._items.pop()
64
65
```

```
    class EmptyStackError(Exception):
        """Exception raised when an error occurs."""
        pass


    def size(s: Stack) -> int:
        """Return the number of items in s.

        >>> s = Stack()
        >>> size(s)
        0
        >>> s.push('hi')
        >>> s.push('more')
        >>> s.push('stuff')
        >>> size(s)
        3
        """
        side_stack = Stack()
        count = 0
        # Pop everything off <s> and onto <side_stack>, counting as we go.
        while not s.is_empty():
            side_stack.push(s.pop())
            count += 1
        # Now pop everything off <side_stack> and back onto <s>.
        while not side_stack.is_empty():
            s.push(side_stack.pop())
        # <s> is restored to its state at the start of the function call.
        # We consider that it was not mutated.
        return count


    # TODO: implement this function!
    def remove_big(s: Stack) -> None:
        """Remove the items in <stack> that are greater than 5.

        Do not change the relative order of the other items.

        >>> s = Stack()
        >>> s.push(1)
        >>> s.push(29)
        >>> s.push(8)
        >>> s.push(4)
        >>> remove_big(s)
        >>> s.pop()
        4
        >>> s.pop()
        1
        >>> s.is_empty()
        True
        """
        pass


    # TODO: implement this function!
```

```
    def double_stack(s: Stack) -> Stack:
        """Return a new stack that contains two copies of every item in <
    stack>.

        We'll leave it up to you to decide what order to put the copies
    into in
        the new stack.

        >>> s = Stack()
        >>> s.push(1)
        >>> s.push(29)
        >>> new_stack = double_stack(s)
        >>> s.pop()  # s should be unchanged.
        29
        >>> s.pop()
        1
        >>> s.is_empty()
        True
        >>> new_items = []
        >>> new_items.append(new_stack.pop())
        >>> new_items.append(new_stack.pop())
        >>> new_items.append(new_stack.pop())
        >>> new_items.append(new_stack.pop())
        >>> sorted(new_items)
        [1, 1, 29, 29]
        """
        pass


    if __name__ == '__main__':
        import doctest
        doctest.testmod()
```

Listing 1: mystack.py

2) Queues

You learned in the readings this week that a queue is another simple ADT that works in the opposite way to a stack: when asked to remove an item, a queue always removes the item that was added first, making it a "first-in, first-out" (FIFO) data structure.

Any time we'd like to model a lineup (of people at the grocery store, for example), queues are a natural choice.

Here are the operations supported by the Queue ADT:

- *__init__*: initialize a new empty queue
- *is_empty()*: return whether the queue is empty.
- *enqueue(item)*: add item to the back of the queue.

- *dequeue()*: remove and return the front item, if there is one, or None if the queue is empty.

Your first task is to implement the *Queue* class found in *myqueue.py*. We strongly recommend you review the stack implementation from lecture to remember how we used a list to implement the *Stack* class.

After you've finished, complete the functions *product* and *product_star* in *myqueue.py*, which compute the product of all numbers in a queue (but have an important difference).

Notice that these are defined outside the class (they are not indented). That makes them ordinary top-level functions, not methods.

```
"""CSC148 Lab 4: Abstract Data Types

=== CSC148 Winter 2020 ===
Department of Computer Science,
University of Toronto

=== Module Description ===
In this module, you will develop an implementation of the Queue ADT.
It will be helpful to review the stack implementation from lecture.

After you've implemented the Queue, you'll write two different functions that
operate on a queue, paying attention to whether or not the queue should be
modified.
"""
from typing import Any, List, Optional


# TODO: implement this class! Note that you'll need at least one private
# attribute to store items.
class Queue:
    """A first-in-first-out (FIFO) queue of items.

    Stores data in a first-in, first-out order. When removing an item from the
    queue, the least recently-added item (i.e. the oldest item in the Queue)
    is the one that is removed.
    """
    def __init__(self) -> None:
        """Initialize a new empty queue."""
        pass

    def is_empty(self) -> bool:
        """Return whether this queue contains no items.

        >>> q = Queue()
```

```
            >>> q.is_empty()
            True
            >>> q.enqueue('hello')
            >>> q.is_empty()
            False
            """
            pass

        def enqueue(self, item: Any) -> None:
            """Add <item> to the back of this queue.
            """
            pass

        def dequeue(self) -> Optional[Any]:
            """Remove and return the item at the front of this queue.

            Return None if this Queue is empty.
            (We illustrate a different mechanism for handling an erroneous
    case.)

            >>> q = Queue()
            >>> q.enqueue('hello')
            >>> q.enqueue('goodbye')
            >>> q.dequeue()
            'hello'
            """
            pass


    def product(integer_queue: Queue) -> int:
        """Return the product of integers in the queue.

        Remove all items from the queue.

        Precondition: integer_queue contains only integers.

        >>> q = Queue()
        >>> q.enqueue(2)
        >>> q.enqueue(4)
        >>> q.enqueue(6)
        >>> product(q)
        48
        >>> q.is_empty()
        True
        """
        pass


    def product_star(integer_queue: Queue) -> int:
        """Return the product of integers in the queue.

        Precondition: integer_queue contains only integers.

        >>> primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

```
        >>> prime_line = Queue()
        >>> for prime in primes:
        ...     prime_line.enqueue(prime)
        ...
        >>> product_star(prime_line)
        6469693230
        >>> prime_line.is_empty()
        False
        """
        pass


    if __name__ == '__main__':
        import doctest
        doctest.testmod()
```

Listing 2: myqueue.py

3) Running timing experiments

Now that you have a complete working implementation of Queue, we'll introduce a new technique for evaluating your code: running timing experiments to see how quickly it runs.

This is a very simple form of software profiling, which is the act of measuring the amount of resources (like time) a program takes while its running.

In fact, PyCharm has a built-in profiler, but that's more advanced than what we need for this course, and so you'll use a simpler profiling library that comes with Python.

1. Your first task is to open *timequeue.py* and follow the instructions contained within it to complete the timing experiment.

2. After you've run your experiment, you should notice that your two queue operations *enqueue* and *dequeue* behave quite differently.

   While one seems to take the same amount of time no matter how many items are in the queue, the other takes longer and longer as the number of items are in the queue.

   Compare your notes with other groups. Which end of a Python list seems to be the "slow" end? Do you have a guess as to why this might be the case? (If you don't: don't worry! You'll learn about this in later weeks.)

   If you still have time, continue on to the additional exercises!

```
"""CSC148 Lab 4: Abstract Data Types

=== CSC148 Winter 2020 ===
Department of Computer Science,
University of Toronto

=== Module Description ===
This module runs timing experiments to determine how the time taken
to enqueue or dequeue grows as the queue size grows.
"""
from timeit import timeit
from typing import List, Tuple

from myqueue import Queue


############################################################
# Task 3: Running timing experiments
############################################################
# TODO: implement this function
def _setup_queues(qsize: int, n: int) -> List[Queue]:
    """Return a list of <n> queues, each of the given size."""
    # Experiment preparation: make a list containing <n> queues,
    # each of size <qsize>.
    # You can "cheat" here and set your queue's _items attribute directly
    # to a list of the appropriate size by writing something like
    #
    #     queue._items = list(range(qsize))
    #
    # to save a bit of time in setting up the experiment.
    pass


def time_queue() -> None:
    """Run timing experiments for Queue.enqueue and Queue.dequeue."""
    # The queue sizes to try.
    # You can change these values if you'd like.
    queue_sizes = [10000, 20000, 40000, 80000, 160000]

    # The number of times to call a single enqueue or dequeue operation.
    trials = 200

    # This loop runs the timing experiment. It has three main steps:
    for queue_size in queue_sizes:
        #   1. Initialize the sample queues.
        queues = _setup_queues(queue_size, trials)

        #   2. For each one, calling the function "timeit", takes three
        #      arguments:
        #        - a *string* representation of a piece of code to run
        #        - the number of times to run it (just 1 for us)
```

```
52              #         - globals is a technical argument that you DON'T need
    to
53              #           care about
54              time = 0
55              for queue in queues:
56                  time += timeit('queue.enqueue(1)', number=1, globals=
    locals())
57
58              #   3. Report the total time taken to do an enqueue on each
    queue.
59              print(f'enqueue: Queue size {queue_size:>7}, time {time}')
60
61          # TODO: using the above loop as an analogy, write a second timing
62          # experiment here that runs dequeue on the given queues, and
    reports the
63          # time taken. Note that you can reuse most of the same code.
64          for queue_size in queue_sizes:
65              pass
66
67
68      # TODO: implement this function
69      def time_queue_lists() -> Tuple[List[int], List[float], List[float]]:
70          """Run timing experiments for Queue.enqueue and Queue.dequeue.
71
72          Return lists storing the results of the experiments.  See the lab
73          handout for further details.
74          """
75          pass
76
77
78      if __name__ == '__main__':
79          time_queue()
```

Listing 3: timequeue.py

4) Additional exercises

Graphing your results

Let's make our timing experiment a bit more visually appealing. Rather than print out a bunch of numbers, it would be much more satisfying to take these values and plot them on a graph to display.

To do this, you'll need to do a few things:

1. Implement *time_queue_lists*, a modified version of your timing experiment function that returns a tuple containing three lists:

    - A list of queue sizes it tried
    - A list of the corresponding times to run enqueue for each queue size

- A list of the corresponding times to run dequeue for each queue size

Note that each of your lists should have the same length.

2. To actually plot the data, we'll use the Python library *matplotlib*, which is an extremely powerful and popular library for plotting all sorts of data.

   If you're on a Teaching Lab machine, you already have this library installed.

   If you're on your own machine, you should have already installed this library by following the CSC148 Software Guide. (Look for the section on installing Python libraries.)

   Add the statement *import matplotlib.pyplot as plt* to the top of *timequeue.py*, and make sure you can still run your file without error.

3. To get a basic 2-D plot of your timing data, work your way through the first part of this guide (Links to an external site.). (Ignore all of the references to "numpy", which is another Python library we aren't using in this course. Also ignore the other sections after the first one; the whole tutorial is pretty long!)

   You can use an x-axis range of 0-200000 and a y-axis range of 0-0.02 (feel free to adjust the y-axis depending on how long the experiments take to run on your computer).

4. If you still have time, explore! There's lots of customization you can do with *matplotlib* to make your graphs really pretty.

Undo and redo