COSC122 (2017) Lab 2

Stacks and Queues

Goals

This lab will provide you with some practice with *Stacks* and *Queues*. In this lab you will:

- complete the implementations for simple stack and queue classes.
- evaluate the speed of stack, queue and deque operations.
- learn how to say deque without confusing everyone.
- apply stacks, queues and deques to the evaluation and transformation of simple equations.

You should be familiar with the material in Chapter 3 of the textbook, ie, Basic Data Structures.[1]

Python doesn't provide specialised stack or queue implementations, but its `list` can be utilised as either of them. The `stack.py` and `queue122.py` modules contain the class definitions for a `Stack` and `Queue` respectively. They both use a Python `list` internally to keep track of stack/queue items and they both provide proper interfaces for working with a stacks and a queues.

You first job is to fill in the code for the relevant methods in each module. That is, they're missing implementations for the most important methods: `push`/`pop`/`peek` and `enqueue`/`dequeue`! Before you start you should unzip **all** the lab files to a suitably named folder—`lab2` would be a good name for such a folder, then from there you can open the relevant files as you need them.

Implementing a Stack and a Queue

Complete the missing implementations for both the `Stack` and `Queue` classes in the `stacks` and `queue122` modules using the `self._items` lists created in their constructors.[2]

Remember that a stack is 'last-in–first-out' (LIFO), and a Queue is 'first-in–first-out' (FIFO). The stack implementation should push and pop items from the end of the list; with the queue implementation it doesn't matter which end of the list you decide to add items to as long as `dequeue` returns the right value from the other end of the list.

When you've completed your implementations you can manually experiment with the classes, for example:

```
from stack import Stack
>>> s = Stack()
>>> s.push(1)
>>> s.push(2)
>>> s.push(3)
>>> len(s)
3
>>> s.pop()
3
>>> s.peek()
2
```

```
>>> from queue122 import Queue
>>> q = Queue()
>>> q.enqueue(1)
>>> q.enqueue(2)
>>> q.enqueue(3)
>>> len(q)
3
>>> q.dequeue()
1
```

[1] http://interactivepython.org/courselib/static/COSC122/index.html#basic-data-structures

[2] Check out `help(list)` in the shell if you can't remember which methods are helpful.

Doctests are handy

You can also test the modules with the doctests for each of the classes (by running the file in *Wing*). Initially you should get a failure for all the tests for the functions that aren't implemented. As you implement functions correctly you should get fewer and fewer errors until finally you reach a blissful state of nothingness (by default doctests will give no output if everything is as expected). The doctests in each module will run automatically (you can stop them running by commenting out the `doctest.testmod()` line at the end of the module).

> ***Complete questions 1 to 2 in Lab Quiz 2.***

Timing Stack and Queue Operations

You should now open `struct_timer.py` and use it to run some time trails—you will see we have given you some help to get started.

For each of the main stack and queue operations (ie, `push`, `pop`, `enqueue` and `dequeue`) carry out the following five steps (writing your code into the `struct_timer.py` file):

1. Create a stack/queue (whichever is relevant for the operation you are testing) with 1,000,000 items.
2. Perform the given operation 100 times (eg, do 100 pushes).
3. Time step 2 using `get_time()`.
4. Find the average time for a single operation (over this stack/queue size range).
5. Repeat the steps 2, 3 and 4 for an initial stack/queue size of 10,000,000 items.

Which operations look like O(1) and which look like O(n)?

> ***Complete questions 3 to 7 in Lab Quiz 2.***

The Deque Data Structure

A Deque is a queue that can have items added or removed from either the front or rear of the queue. If you have read the textbook then you will know that deque is pronounced as 'deck'. A deque can be thought of as a deck of cards - you can add a card to the bottom or top of the deck and you can remove a card from the top or bottom (all depending on how dodgy your card playing is). *Some people pronounce deque as 'dee-cue' but this is obviously going to get confused with dequeue operations!!!*

Most of the methods for a Deque class have been implemented in the `deque` module. You will need to complete the `dequeue_front`, and `dequeue_rear` methods.

Once you have completed the class you can run the doctests doctests for a quick check, but we recommend you manually experiment with a Deque or two, for example:

```
>>> from deque import Deque
>>> d = Deque()
>>> d.enqueue_front(1)
>>> d.enqueue_front(2)
>>> d.enqueue_rear(3)
>>> d.dequeue_front()
2
>>> d.dequeue_rear()
3
>>> len(d)
1
```

> ***Complete question 8 in Lab Quiz 2.***

Timing Deque Operations

Find the time taken by `enqueue_front`, `enqueue_rear`, `dequeue_front`, and `dequeue_rear` methods by repeating steps one to five (from the Stack/Queue testing section).

> ***Complete questions 9 and 10 in Lab Quiz 2.***

Application: Mathematical Expressions

From lectures, you should be familiar with arithmetic expressions written in three different notations: *infix* (3 + 4), *prefix* (+ 3 4), and *postfix* (3 4 +). The `expressions.py` module contains one completed method: `calculate` and a number of incomplete methods that you will eventually write the code for. The docstrings and doctests for each method describe their operation; refer to pages 95–103 of your textbook for a deeper explanation of the algorithms.[3]

Evaluating Postfix Expressions

Postfix is also known as Reverse Polish Notation (RPN) and was invented in the 1920s by Jan Łukasiewicz. See (http://en.wikipedia.org/wiki/Reverse_Polish_notation) for more background.

Evaluating Postfix expressions can be done by using a stack. We give a few simple examples below and suggest using a pen and paper to work through the use of a stack to evaluate the expressions, this will give you a better feel for the algorithm before you actually write any Python code. For example, the first expression should evaluate to `20`.

Evaluate each of the following Postfix expressions by hand:

`2 3 + 4 *`	`2 3 4 * +`	`2 3 4 + *`	`2 3 * 4 + 2 /`

Why is a space/comma in between all operands necessary when writing equations in Postfix notation?

Now complete the `evaluate_postfix` method (in `expressions.py`) to save you all this laborious hand-cranked evaluation . As described in the doctests, this method receives a string like `2 3 4 * +` and will evaluate it from left-to-right using a `Stack`, returning the result.

> ***Complete question 11 in Lab Quiz 2.***

Converting Infix to Postfix Expressions

Converting from Infix to Postfix can be done by using a Stack. Simple expressions can be done in a relatively intuitive way but we suggest you try using pen and paper to hand crank the conversions so that you can get a feel for the algorithm. Your lecture handout includes the pseudo-code for this algorithm and there is quick video demonstration available in the section for this topic on the quiz server.

Convert the following Infix expressions to Postfix notation by hand before you write the code:

`4+3/2`	`4*3+2`	`4*(3+2)`	`(3/2)+4`

The constant definition below (a dictionary) is included (where relevant) to allow you to easily lookup the precedence order of a given operator, eg, (`OP_PREC["+"]`) will return `2` and (`OP_PREC["*"] > OP_PREC["+"]`) will return `True`.

```
OP_PREC={"(":1, "+":2, "-":2, "*":3, "/":3, ")":4}
```

Now complete the `infix_to_postfix` method. As described in the doctests, this method takes a string like `2 + 3 * 4` and converts it to a post-fix expression, returning `2 3 4 * +` in this case. Test your implementation with the doctests and your own experimentation.

> ***Complete questions 12 to 13 in Lab Quiz 2.***

[3] http://interactivepython.org/courselib/static/COSC122/BasicDS/stacks.html#infix-prefix-and-postfix-expressions

Extras

> ***Complete question 14 Lab Quiz 2.***

- Add support for exponentiation, indicated by a caret symbol (^), to the postfix evaluator, infix-to-postfix converter, and infix evaluator. The first operand is raised to the power of the second, and exponentiation has a greater precedence than that of multiplication.

The following extras exercises are aimed at students who find themselves very comfortable with the material in this lab and want to challenge themselves. Answering the questions may use ideas that don't feature in this lab and/or course.

- Implement a prefix_to_postfix function using a Stack. Check out the following link for some psuedo-code that will help: `http://fahadshaon.files.wordpress.com/2008/01/prefix-to-postfix-stack-algo.pdf`. Implementing this function will help you answer the prefix to postfix conversion question in the lab quiz (ie, the extension question). For extra fun you should think about how your function can tell if there are too many operators—and raise an exception in this case. You don't need to worry about the case where there are too many operands and this will cause the stack to raise an exception when it trys to pop while empty.

- Complete the `evaluate_infix` method. That is, a method to evaluate infix expressions directly (don't just `return evaluate_postfix(infix_to_postfix(infix))`!).

  The general strategy will be similar to the way `infix_to_postfix` works, but evaluating operators as it encounters them, rather than converting them to a postfix string. Think about where you might change the algorithm if you wanted to evaluate the expression rather than translate. To complete this method, you will want to maintain two stacks: one for operators, and one for operands. Whenever you encounter an operand, push it onto its stack; whenever you encounter an operator, process it as the infix converter does—making sure that whenever you pop an operator, you evaluate it as you would for a postfix operator.

- Implement an evaluator for expressions in `prefix` notation.

  This is slightly trickier than evaluating postfix notation because you have to have to read ahead for each operator you encounter and only evaluate it if you have a complete set of operator and operands. An approach you might want to take would be to enqueue all the items at the rear of an deque, then dequeue from the front and apply the following process until only one item remains in the deque–this will be the answer. Dequeue items from the front of the deque and try to build expressions until you get a token that means you can't get the three tokens you need to use calculate (ie, you will need operator operand1 operand2 so if you get the sequence operator operand operator you will need to enqueue the first operator and the operand to the rear of the deque and then enqueue the last operator at the front of the deque). If you manage to get an operator operand1 operand2 sequence then you can send it to the calculate function and then enqueue the result at the back of the deque.