Week 4 - Linked Lists

In the previous week, we saw that the standard array-based Python implemention of lists has some drawbacks: inserting and deleting requires shifting of many elements in the array. This week, we're going to study another implemention of lists: the **linked list**.

Note that in contrast to last week, when we looked at the `Stack` and `Queue` ADTs, this week we are focusing on an alternate implementation of an *existing* ADT: the familiar list. Thus our goal is to create a new class that behaves exactly the same as existing lists, changing only what goes on behind the scenes.

On the last exercise, you explored the `PeopleChain` class. If we abstract away the notion of people holding onto each other, we see that this class is precisely a list, where every element of the list stores a "link" to the next. Intuitively, this makes it easier to remove and delete elements, because the elements don't need to be stored in memory in order; insertion and deletion involves only changing a few links.

Here's a basic implementation of a linked list: note the similarities to `PeopleChain`.

```
class Node:

    def __init__(self, item):
        self.item = item
        self.next = None  # Initially pointing to nothing


class LinkedList:

    def __init__(self, items):
        """
        Create Node objects linked together in the order provided in items.
        Set the first node of the chain as the first item in items.
        """

        if len(items) == 0:  # No items, and an empty list!
            self.first = None
        else:
            self.first = Node(items[0])
            current_node = self.first
            for item in items[1:]:
                current_node.next = Node(item)
                current_node = current_node.next
```

How does this constructor work? Let's step through it carefully.

Traversing a Linked List

Let's now look at traversing a linked list. We're basically going to have one technique and apply it to a bunch of different methods. This may seem repetitive, but this is one of the most technically challenging and important parts of the course, so spend the time to

master it!

We have already seen the most important part in the constructor: a *temporary variable* that iterates through each node of the list. Let's remind you about how this might look for a regular list (the following code should be easy enough to understand, but please keep in mind that there are more "Pythonic" ways of iterating through a list).

```
i = 0
while i < len(my_list):
    # Do something with my_list[i], e.g.:
    print(my_list[i])
    # Increment i
    i = i + 1
```

The code consists of four parts:

1. Initializing the temp index
2. Checking if we've reached the end of the list
3. Doing something with the current element `my_list[i]`
4. Incrementing the index

The Linked List has all of these analogous actions, the main difference being that the temporary variable in question now refers to a particular `Node` object rather than an index. Understanding how this works is the first step to understanding linked lists. Here is a code snippet that prints out every `Node`'s `item` attribute in a linked list.

```
curr = my_linked_list.first
while curr is not None:
    print(curr.item)
    curr = curr.next
```

With this in mind, let's complete the following methods for the `LinkedList` class. Note that we're *overriding* built-in `Object` methods to make convenient use of Python syntax.

```
def __len__(self):
    """ (LinkedList) -> int

    Return the number of elements in self.
    """

    pass

def __getitem__(self, index):
    """ (LinkedList, int) -> str

    Return the item at position index in self.
    """

    pass
```

David Liu (liudavid at cs dot toronto dot edu)
Come find me in BA4260
Site design by Deyu Wang (dyw999 at gmail dot com)