CSCI 241 Data Structures
Project 2: Literally Loving Linked Lists LOL

In this project, you will implement a sentineled doubly-linked list. Recall that a linked list is composed of Node objects that are linked together. This means that we will need to create two classes in this implementation. One class will represent the Nodes of data and how they are linked together. The other class will represent the actual Linked List, defining methods for adding and removing elements, creating a string representation of the object, and obtaining its length.

We have discussed a variety of methods for inserting and removing values in a linked list. This project will use index-based addressing. Recall from our studies of arrays that index zero identifies the location of the first datum. This approach also means that the maximum valid index is one less than the length of the sequence. We will replicate that indexing paradigm here. Note that index zero identifies the first Node object that contains data, and not the header. Neither the header nor the trailer has an index.

Your implementation should support the following methods, keeping in mind that the words index, head, and tail are used descriptively only and should not appear as attributes of either class:

`append_element(self, val)` This method should increase the size of the list by one, adding the specified value in the new tail position. This is the only way to add a value as the tail.

`insert_element_at(self, val, index)` If the provided index identifies a valid zero-based position within the list, then insert the specified value at that position, increasing the length by one. This method can be used to insert at the head of a non-empty list, but cannot append to a list. The provided index must be within the current bounds of the list. If the index is not valid, raise an `IndexError` exception.

`remove_element_at(self, index)` If the provided index identifies a valid zero-based position within the list, then remove and return the value stored in a Node at that position. If the index is not valid, raise an `IndexError` exception.

`get_element_at(self, index)` If the provided index identifies a valid zero-based position within the list, then obtain the value from the Node at that position and return it. Do not unlink the Node object. If the index is not valid, raise an `IndexError` exception.

`rotate_left(self)` Without constructing any new node objects and without returning anything, rotate the list left so that each node moves one position earlier than it was and the original head becomes the new tail. The length of the list should not change. If the list is empty, this method has no effect.

`__len__(self)` Return the number of values currently stored in the list.

`__str__(self)` Return a string representation of the values currently stored in the list. An empty list should appear as `[ ]` (note the single space). A list with one integer object should appear as `[ 5 ]` (note the spaces inside the brackets). A list with two integer objects should appear as `[ 5, 7 ]`, and so on. Pay close attention to the format of this string, and remember that strings can be concatenated with the + operator. To convert other objects to strings, use `str(other_object)`. As long as the class for the other object implements the `__str__()` method, this approach will work.

`__iter__(self)` `See "Iterators" below.`

`__next__(self)` `See "Iterators" below.`

Exceptions

In lecture, we have silently ignored bad method calls (such as requesting the value of an index that is equal to or greater than the length of the list) by detecting that condition at the beginning of the method and returning. In practice, it is better to inform the programmer that her request was invalid, and allow her to handle the problem. The mechanisms for achieving this are called exceptions and try blocks. When you detect an error condition, instead of returning, raise the appropriate exception using the syntax

```
raise ExceptionName
```

When the programmer calls a method that could potentially generate an exception, she does so in what we call a try block. Suppose she calls a method that could raise a `ValueError`. Her code to invoke that method would have to look like this:

```
my_object = Awesome_Class()
try:
  asplode = random.randint(1,10)
  my_object.dangerous_call(asplode)
  print("whew... made it.")
except ValueError:
  print("**>_KABOOM_<**")
print("on the other side.")
```

Perhaps the `dangerous_call(num)` method raises a `ValueError` if the value of `num` is 5, and raises no exception otherwise. Because `asplode` is equally likely to be one of ten values (one through ten, inclusive) in the example above, she will get with 90% probability

```
whew... made it.
on the other side.
```

or with 10% probability (when `asplode` is the one of ten possible values, the value 5, that is problematic)

```
**>_KABOOM_<**
on the other side.
```

Each method in your Linked List class that takes an index as a parameter should raise an `IndexError` (this type is built-in to Python) if the provided index is out of bounds. For our implementation, indices that are either too large or negative should be considered out-of-bounds.

Inner Classes

One thing that we have mentioned briefly during lecture that is relevant to this project is the concept of inner classes. We already know that the Linked List implementation will employ objects of a Node class, so these two classes will be working together. An important point, though, is that the end user of the Linked List doesn't actually see Nodes. Think back to arrays for a moment; when you use an array, you don't actually see the cells that store the data. You see and interact with the data themselves. The same will be true for Linked Lists. The person using your list implementation doesn't actually care about Nodes. All she cares about is that her data are stored in the list. Because of this, it is not necessary (or even desirable) for the Node class to be exposed to the world. It should only be used internally by the Linked List implementation methods. When a user inserts data into the list, she provides the data as an object of whatever type she is storing. If she is dealing with integers, she will insert the number 5, not a Node containing the number 5. The use of a Node object to encapsulate the value 5 is completely internal to the Linked List and is never exposed.

To help provide this encapsulation, your solution should implement the Node class itself as a private member of the Linked List class. By marking the class private (with leading underscores) and implementing it inside of the Linked List class, we indicate that it should only be used internally to Linked Lists. The concept is similar to private attributes, but instead of being declared as `self.__attr_name` inside of the constructor, the inner class is defined at the same level as the methods. Note the discussion later in this specification about transitivity of privacy.

Iterators

Using the method `get_element_at(index)`, we could visit every node in the list by looping through all of the valid indices. The problem with that approach is that instead of linear time performance, we have quadratic time. Notice that the `get_element_at(index)` method is linear. It must do a current-walk to reach position `index`, which is at the tail position in the worst case. Retrieving the first element will take 1 step; retrieving the second element will take 2 steps. You should recognize this pattern from our analysis of insertion sort. The sum of n consecutive integers beginning at 1 is bounded by n^2. Considering how frequently we use loops to visit every value in a sequence, quadratic performance is not desirable.

To keep the time linear as expected, we employ a programming structure called an iterator. You have used iterators many times. Consider the following code segment:

```
arr = [5,2,-4,1]
for val in arr:
  print(str(val))
```

The loop *iterates* through the values in the array. When Python encounters the loop, it initializes an index for the array. On every entrance of the loop, it assigns `val` the value contained at that index, then increments the index. When the index reaches the length of the array, the iteration terminates.

You can replicate this behavior for any class you write by implementing two special methods: `__iter__(self)` and `__next__(self)`. Analogous to the code segment above is the following version that uses a linked list object instead of an array:

```
ll = Linked_List()
ll.append_element(5)
ll.append_element(2)
ll.append_element(-4)
ll.append_element(1)
for val in ll:
  print(str(val))
```

Right before the for loop, the object `ll` should contain the elements 5, 2, -4, and 1. When Python encounters the for loop, it invokes the `__iter__()` method on the `ll` object (after the keyword **`in`**). This is Python's way of telling the `ll` object to prepare to cycle through its elements. In your `__iter__()` method, you should initialize a current pointer in preparation for walking the list. Each time Python enters the indented **for** block, it assigns `val` whatever is returned by a hidden call to `__next__()`. In your `__next__()` method, you should decide whether there is another value to return. If so, advance to the node whose value should be returned and return that value. If not, raise a `StopIteration` exeption. Python will automatically handle the exception as a signal to stop calling your `__next__()` method. This terminates the **for** loop.

Below is the skeleton implementation that you will complete. The Python file attached to this assignment contains comments describing each method.

```
class Linked_List:

  class __Node:

    def __init__(self, val):

  def __init__(self):
```

```
  def __len__(self):

  def __iter__(self):

  def __next__(self):

  def append_element(self, val):

  def insert_element_at(self, val, index):

  def remove_element_at(self, index):

  def get_element_at(self, index):

  def rotate_left(self):

  def __str__(self):

if __name__ == '__main__':
```

Most importantly, notice that the Node class is defined within the Linked List class and is private to that class. This means that only the methods inside of the Linked List implementation have access to Nodes; they are not exposed to the user. It also means that to create a new node inside of an insert method, the syntax is

```
new_node = Linked_List.__Node(my_val)
```

Then, `new_node` is a Node object that can be linked in at the appropriate location. In most object-oriented languages, outer classes have access to the private members of inner classes. This is not true in Python, so we must make the Node attributes public. Alternatively, we could add accessor and mutator methods to the Node class, but that would add significant overhead given the frequency of node references. Even though we make the Node attributes public, the nodes themselves can only be referenced from the Linked List methods, because the very concept of what a Node is is private to the Linked List class.

In the main section of your Python file, provide some test cases to ensure that your Linked List implementation functions correctly. Though this is not an exhaustive list, some things to consider are:

- Does appending to the list add an element at the new tail position and increment the size by one?
- Does inserting an item at a valid index increase the size by one and correctly modify the list's structure?
- Does inserting an item at an invalid index leave the list completely unchanged?
- Does removing an item at a valid index decrease the size by one and correctly modify the list's structure?
- Does removing an item at an invalid index leave the list completely unchanged?

- Does length always return the number of values stored in the list (not including sentinel nodes)?
- Is the string representation of your list correct for a variety of lengths?
- Does a for loop like

  ```
  for val in my_list
  ```

  visit every value?

Writeup

Your writeup for this project should be significant. It must contain a performance analysis of every method in your Linked List class and your solution to the Josephus problem, a detailed explanation of how you approached testing your Linked List, and thorough comparison of our approach and the textbook's model. Performance analysis should be well-justified; explain why the implementations have the stated performance. Testing should be explained not just as how it was done, but why it was done and how that improves the reliability of the structure. When you compare and contrast the textbook's approach with ours, consider ease of implementation, performance, and any other implications of the design choices you can determine.

Submission Expectations

1. **`Linked_List.py`**: A file containing your completed Linked List class implementation. Though you are free to add additional methods as you deem necessary, you must not change the names (including spelling) or parameter lists of any methods in the skeleton file. The main section at the bottom of this file must contain your testing code.
2. **`Writeup.pdf`**: A prose document that details why you performed the test cases that you did and explains why those tests are complete. You must also describe the performance characteristics of each of the methods in the `Linked_List` class along with your Josephus solution. Finally offer a deliberate discussion on the similarities, differences, and implications of this approach versus the textbook's approach.
3. **`Josephus.py`**: Your completed Josephus application using the provided skeleton file.