Instructions: In this homework, you will create a class named `LinkedList` to implement a linked list data structure that can be iterated over with a for loop. You'll need to create your own classes, raise exceptions, overload operators and built-in functions, and implement an iterator, and write a generator. Name your file `hw3.py` and submit on CCLE.

Test code:
The provided test code lists point values associated with various pieces of functionality. For credit, the output of your code should match the included "Example output" and should work equally well for similar test commands.

Tasks:

- : Create a class `Node`.
  - The initializer should accept one argument - the data to be stored in the node. Store the data in an instance variable `data` and the reference to the next node in an instance variable `next`. What should `next` be initialized to, considering there are no other nodes in the list?
    Hint: search online for the Python equivalent to the "null" of C++ and Java.
  - Write magic methods such that the built-in `str` and `repr` functions can be used to return a string representation of the node, e.g. the following code:

    ```
    n = Node(10)
    print(str(n))
    print(repr(n))
    print(n)
    ```

    should print `10` three times.
    Hint: remember that `str` and `repr` already work on whatever data is stored in the node. Leverage this to make them work on your `Node` objects.
- Create a class `LinkedList`.
  - The constructor should accept one argument: the data to be stored in the first node. Create a new node to store that data, and set two instance variables of the `LinkedList` – `first` and `last` – to that node. A third instance variable `n` should track the number of elements.
  - Write a method `append` that accepts one argument: the data to be stored in a new node at the end of the list. Store that data in a new node, update the `next` field of the current `last` node of the Linked List, and update the `last` node. Remember to increment `n`.
  - Write methods `__iter__` and `__next__` so that the data structure can be iterated over with a `for` loop. Do not use a generator (yet). When you are done, the code:

    ```
    a = LinkedList(0)
    a.append(1)
    a.append(2)
    for n in a:
        print(n)
    ```

should print the numbers 0 through 2 to the console.
Hint: `__next__()` is going to need to both return the value stored in a node and keep track of which node will come next. When the end of the list is reached (how do you know when that happens?) it needs to raise a `StopIteration` exception (review 8.4).

- If you try to run the same for loop (`for n in a: print(n)`) again on the same `LinkedList` object, I'll bet nothing will print out (unless you have great foresight). Figure out why and fix it so that you can loop over the same `LinkedList` object as many times as you wish. Don't use a generator to fix the problem (yet).

- If you run the following code:

```
for n in a:
    if n == 2:
        break
    else:
        print(n)
```

  once, it will print out the numbers 0 and 1. If you run it a second time (without changing the object referred to by a), it probably won't work. Fix this using a *generator*, which can replace some of the code you've already written.

  Hint: this is why there needs to be an `__iter__` method and collections don't just define their own `__next__` method. By the way, writing this generator is very easy once you understand how generators work. It took me five lines, and afterward I could comment out the original `__next__` method.

- Implement magic methods such that built-in functions `len`, `str`, and `repr` will work. The string representations should be like `[0->1->2->]` (yes, the end arrow is OK).

- Implement magic methods such that you can set and get (single) nodes in your list via bracket notation, e.g. `print(a[0]), a[1] = 10`. If the index is out of bounds, raise the appropriate exception.

- Implement the appropriate magic method so that the + operator concatenates the second operand to a *copy* of your list.

  That is, running `a + 1` should not modify `a`;

  `a = a + 1` should effectively append `1` to the list (by replacing the original).