CS2121B - Assignment 2 - 2020

Linked Lists: Implementing a Functional Design

Out: January 29th, 2020

Due date: February 12th, 2020 at 11:00 pm via OWL

Note Well:

You must take note of the following:

- You are given two (2) .py files that you are to complete. For your work with these files take note of the following:
  - Your function names, along with the parameters they take **must** remain the same; **you cannot alter these**. If you alter these, you will lose significant marks and risk getting a mark of zero.
  - A file called `test.py` will be used by the TA to mark your assignment – I have provided you with the file we will actually use to test your code. You are free to use this on your own code.
  - Please **do not** prompt the user for input; all functions will be called by just calling the functions.
- Some of the descriptions are deliberately left somewhat vague. This gives you practice interpreting design instructions, but things will become clear as you get further on with your implementation.
- If there is something you are not sure how to do, you should first Google it or look it up in the Python documentation online: `https://docs.python.org/3/index.html`

Purpose

The purpose of this assignment is to continue to familiarize yourself with writing your own classes, using them, and learning new types of collections. In particular, in this assignment you will familiarize yourself with linked structures and gain experience of how to interpret functional requirements (specification of the behaviour of software) and how to develop your implementation to meet them.

Summary

You will be writing a list type class that is based on an underlying linked list rather than an array. This class will be called `OurLinkedList`.

Provided:

You are provided with a testing script (`test.py`) and two incomplete python scripts (`Node.py` and `OurLinkedList.py`) that you are to finish.

Node class

The `Node` class specifies the content and behaviour of a single node in a linked list. This class will have two (5) attributes: `data` and `next`. You will need to complete three (3) functions:

1. `__init__(self, data, next)`
   - This is the constructor, which will set the two attributes of the class.
   - Implement the constructor so that we can pass some `data` and a possible `next` node to the constructor. This will probably require you to use some kind of default values, e.g., if no `next` node is supplied.
2. `__str__(self)`
   - This function will return the string version of the data.
3. `__eq__(self, other)`
   - This function will tell us if two nodes are the same, which will happen if their data is the same. **Do not** check that the next nodes are the same.
   - Also, note that a `Node` obbject is never going to be equal to something that is not a `Node`.

There is nothing else to the `Node` class.

OurLinkedList class

The `Node` class is very easy to implement, but this class will be more challenging. The `OurLinkedList` class will manage an actual linked list and provide the functionality for it that we want. Many of the functions we are implementing here are also defined for the Python `List` class; you may wish to familiarize yourself with the behaviour of the `List` versions as you implement this class. Don't forget to take proper account of special cases, especially empty lists and lists of size 1.

This class has three (3) attributes: `head`, `tail`, and `currNode`. These define the first, last and current nodes of the list. You will need to complete 20 functions:

1. `__init__(self, head)`
   - The constructor will define the three attributes of the class.
   - The constructor should be set up to allow it to take a reference to a head node (of an arbitrarily large pre-existing list) to create an `OurLinkedList` object.
   - But the constructor should also work if no node is supplied, creating an empty list by default.
2. `__str__(self)`
   - Return a string representation of the entire contents of the list.
   - Use commas to separate the data values (you can leave a trailing comma if you like).

- Enclose the contents of the list in square brackets (`[` and `]`).

3. `__len__(self)`
   - Return the length of the list.
4. `__contains__(self, element)`
   - Return `True` if the element is in the list and `False` otherwise.
5. `__eq__(self, other)`
   - Check whether two lists have identical contents in exactly the same order.
6. `__getitem__(self, index)`
   - This function allows us to index the array using an integer (e.g., `myList[4]` would return the data value of the 5th item in the list.
   - Don't worry about allowing slicing.
   - If the `index` is less than 0 or equal to or greater than the length of the list, then raise an `IndexError`: `IndexError('list index is out of range')`
7. `__setitem__(self, index, value)`
   - This function allows us to change the value of a node given an integer index.
   - Note that this does not add a new node, but changes the value of an existing node.
   - If the `index` is less than 0 or equal to or greater than the length of the list, then raise an `IndexError`: `IndexError('list index is out of range')`
8. `__iter__(self)`
   - Complete the code to return an iterator for the list.
9. `__next__(self)`
   - Complete the code to iterate through the list.
10. `getNode(self, index)`
    - This function should be like the `__getitem__` function, but should return a reference (pointer) to the node at the given index.
    - This is not a Python special ('dunder') method.
11. `clear(self)`
    - This function clears out the contents of the list.

12. `shallowCopy(self)`
    - This function creates a shallow copy of the list.
13. `deepCopy(self)`
    - This function creates a deep copy of the list.
14. `index(self, value)`
    - This function will take a data value and return the index of the first node in the list containing that value.
    - The iterator may be helpful for implementing this function.
    - If the value is not in the list, then raise a `ValueError`: `ValueError(str(name) + ' is not in the list')`
15. `append(self, value)`
    - This function adds the supplied value to the end of the list.
16. `prepend(self, value)`
    - This function adds the supplied value to the beginning of the list.
17. `insert(self, index, value)`
    - This function inserts a new node containing the supplied value into the list at the supplied index.
    - After the insert operation the value will be found at the specified index.
    - Make sure **not to overwrite anything**.
    - `getNode` may be helpful here, as may `append` and `prepend`.
18. `pop(self, index)`
    - This function removes the node at the supplied index *and* returns the value of the removed node.
    - If the list is empty then return an `IndexError`: `IndexError('pop from empty list')`.
    - If the index specified is equal to or greater than the length of the list, return an `IndexError`: `IndexError('pop index out of range')`.
    - `getNode` may be useful.
19. `remove(self, value)`
    - Remove the first instance of the specified value from the list.
    - `pop` and `index` may be useful.
20. `reverse(self)`
    - This function reverses the contents of the list.
    - The function should reverse the list without copying any nodes.
    - This will probably be challenging to implement.

Submitting the Assignment

You will submit the file via OWL, under the assignments tab. **Only submit your complete python files** (`Node.py` and `OurLinkedList.py`) via OWL. Please follow the instructions carefully and do what is asked, but not more, to ensure that you do not unnecessarily lose marks. Make sure not to modify the function names.