Linked Lists

In this assignment you will implement a `LinkedList` class. Under the hood, it will be using the built-in collection list of Python, but its API is different.

Objectives

1. Linked lists
2. Magic methods
3. Exceptions
4. Iterators

Part 1: Initial implementation

Initially you have to define a class `LinkedList` including initializer that enables to create empty instances of `LinkedList`. In particular, you have to define the `__init__` method of `LinkedList` that defines an instance variable that references an empty Python list.

Now implement the method `addLast` that enables us to add a new item to the end of the list. For this lab, your implementations of `addLast` method (and most other methods) should not use any list operations besides `append`, reading at an index and writing to an index. That's right, no `len`, no `in`, no anything but `self.values.append(item)`, and `list[index] = item` or `item = list[index]`. This will give you a feel for what it was like to code in the good ol' days before Object-oriented languages were around.

To be able to print instances of `LinkedList` you need to implement `__str__` method which returns a string that represents the content of a given object. The returned string should follow the following format: `[elements separated by semicolons]`

In addition to this, to keep track of the length of the linked list, `LinkedList` will have a class variable `size`, initialized to `0`, which is incremented and decremented appropriately.

At the end of part 1, you should be able to do the following:

```
lst = LinkedList()
print(lst)          # prints []
print(lst.size)     # prints 0

lst.addLast(1)
lst.addLast(2)
```

```
print(lst.size)     # prints 2

lst.addLast(3)
print(lst)          # this should print "[1;2;3]"
```

Make sure your code works correctly no matter what kind of items we store in the LinkedList -- numbers, strings, tuples, etc!

Part 2: Filling out the methods

Now define `addFirst`, which works like `addLast` but adds the new item at the beginning of the list. Similarly, define two methods `removeFirst` and `removeLast` that remove the first item and last item respectively. The methods return the removed item. In case the list is empty, the methods return `None`. Remember to update `size` appropriately!

```
lst = LinkedList()
lst.addLast(5)
lst.addFirst(3)
lst.removeFirst()    # returns 3
print(lst.size)      # prints 1

lst.removeLast()     # returns 5
lst.removeLast()     # returns None
print(lst.size)      # prints 0
```

We also have `addAt`, which inserts a new item at a given index. If the index is `0`, it will insert at the beginning, which is the same as `addFirst`. It does **NOT** overwrite any values already in the list, but "moves over" the ones that are after the new item. `addAt` returns a boolean, which is normally `True`, but is set to `False` if the index is invalid.

```
lst = LinkedList()
lst.addFirst(17)
lst.addFirst(2)       # list is now [2;17]
lst.addAt (1,3)       # returns True
print(lst)            # prints [2;3;17]

lst.addAt(0,5)        # returns True, and the list is now [5;2;3;17]
lst.addAt(6,11)       # returns False, and the list is unchanged
print(lst.size)       # prints 4
```

Similarly, there is also `removeAt`, which removes an item at a given index. If the index is `0`, it will remove from the beginning, which is the same as `removeFirst`. `removeAt` returns the removed item, or `None` if the index was invalid.

```
lst = LinkedList()

lst.addFirst(33)
lst.addFirst(32)
lst.addFirst(11)     # list is now [11;22;33]

lst.removeAt(1)     # returns 22; list is now [11;33]
lst.removeAt(2)     # returns None; list is unchanged
```

Note: For all these methods, you still can't use any other list operations besides `append`, reading at an index and writing to an index!

Part 3: Magic methods and Exceptions

Add a public method called `__contains__` to the `LinkedList` class. This method allows us to use `in` and `not in` to test membership. The method should take a parameter item and should return `True` or `False` depending on whether or not the item is in the list. The method signature is as follows.

```
def __contains__(self, item):
    pass
```

When implemented, the following example code should work.

```
L = LinkedList()

L.addfirst(2)
L.addfirst(1)

print(1 in L)         # prints True
print(3 in L)         # prints False
print(2 not in L)     # prints False
```

Note: to implement `__contains__`, you still can't use any other list operations besides reading at an index and writing to an index!

We also want to provide access to the LinkedList elements using the `[]` operator. To do so you have to implement two methods: `__getitem__` and `__setitem__`. Following are their signatures:

```
def __getitem__(self, idx):
    pass
def __setitem__(self, idx, newitem):
    pass
```

Where idx is the location of the item in the `LinkedList` that you want the method to access, and `newitem` is the new item that will replace the current item at location `idx`.

In Part 2, we used `None` and `False` as "special" return values that indicated bad input indices. Since we never call magic methods directly, we cannot check directly what they return. So when trying to access an element that does not exist, : `__getitem__` and `__setitem__` will **raise an exception** rather than return `None` or `False`:

```
raise Exception("Invalid index " + index)
```

Here is how it would work:

```
lst = LinkedList()
lst.addFirst(17)      # this will work OK
lst[0] = 13          # this replaces 17 by 13
lst[1] = 5           # this will create ERROR message "Invalid index 1"
```

Part 4: Iteration

You might have noticed that instances of `LinkedList` cannot be traversed using the `for` loop as we have been doing for built-in collections (list, tuple, string, etc). The purpose of this section is to have the `LinkedList` class capable of doing so. In particular we want the following code snippet to work:

```
lst = LinkedList()
lst.addFirst(8)
lst.addFirst(5)
lst.addFirst(3)

for item in lst:
    print(item)     # will print 3, then 5, then 8
```

To be able to use `LinkedList` objects in a `for` loop, as we did in this snippet, they have to be **iterable**. To accomplish this, the `LinkedList` class has to implement a magic method called `__iter__` which will get called automatically whenever we enter the `for` loop. Here is its signature:

```
LinkedList.__iter__(self)
```

`__iter__` should return a new object of type `LinkedListIter` (you will have to implement this class separately), which is an iterator. Iterators are objects that know how to iterate over the elements of its iterable object (in this case, `LinkedList`). Iterators have to implement the `__next__` magic method, which gets called once every time we go through the for loop.

```
LinkedListIter.__next__(self)   # returns the next item in the LinkedList
```

`__next__` must raise the `StopIteration` exception once it's out of items. The `for` loop construct listens for the `StopIteration` exception specifically, so as to know when to stop iterating.

Note that once you've implemented the `LinkedListIter` class and the `__iter__` method of the `LinkedList` class properly, the following code snippet should also print out the items of your list:

```
it = lst.__iter__()      # creates a new iterator for the LinkedList
try:
  while True:
    x = it.__next__()    # gets the next value from the iterator,
                         # starting with the first one
    print(x)
except StopIteration:
    None
```

In the `while` loop, we get to see these methods being called explicitly, unlike the `for` loop. In fact, you may want to start part 4 of the lab by getting this one to work, before you test the `for` loop.