Python Intermediate Programming

Unit 3:Python Data Structures

CHAPTER 7: NODES AND LINKED LISTS

DR. ERIC CHOU

IEEE SENIOR MEMBER

Objectives

- ICS-46 is concerned with studying the lower-level data structures that are used in Python to represent `lists/tuples`, `dicts`, `sets/frozensets`, and other not-built-into Python named data-types: `stacks`, `queues`, `priority-queues`, `equivalence classes`, and `graphs`. There are two primary components to all these data-types: arrays and self-referential data-structures. Linked lists are the simplest kind of self-referential data-structures; trees (we will study binary trees) are more complex self-referential data-structures.

Objectives

- Languages like Java/C++ don't build-in most of Python's useful data-types, but instead provide them in standard libraries, which are a bit more awkward to use than these data-types in Python. These data-type libraries are built on arrays and self-referential structures. This week is a peek at self-referential structures (the last week will be a peek at more of Java, and will address some of these same issues in a larger context).

Linked Node

LECTURE 1

Linked Node

- Here is the trivial class that serves as the basis for all our linked list code (and the tree class covered later this week isn't much different). LN stands for List Node: a linked list is a sequence of zero or more list nodes, one explicitly referring to the next (via an attribute that is a next reference).

```
class LN:
    def __init__(self:"LN", value:object, next:"LN"=None):
        self.value = value
        self.next  = next
```

- We write "LN" in the annotations above, because when defining LN we cannot use LN for an annotation (because it hasn't been completely defined yet).

Linked Node

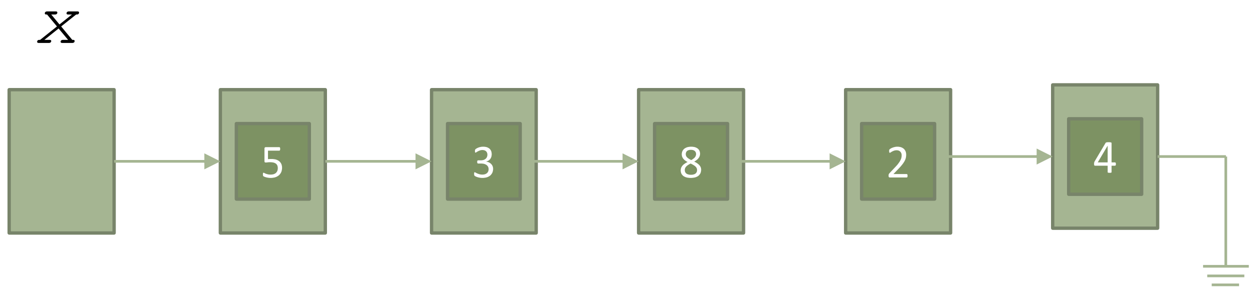
- Basically, the class allows us to create objects with two attribute names: value refers to some object (of any class) but next should either refer to an object constructed from the LN class or refer to the special value None (its default value in __init__ above).
- In this way we describe LN as a self-referential class: each of its objects refers to another one of its objects (although None will serve to stop this recursive definition from being infinite: it will be the base case in our recursive functions that process linked lists): None represents a linked list with no/0 nodes. Generally, a linked list is a sequential list of values, with their order being important.

Linked Node

- So, a linked list is like a standard Python list (implemented by arrays, which you'll learn tons about in ICS-45J/ICS-45C and ICS-46, but are hidden in Python). Here, and much more in these other courses, we will learn many details concerning the objects that implement linked lists and how we can use linked list to implement the kinds of operations we perform on standard Python lists.
- In ICS-46 we will focus the performance tradeoffs (speed/space) for array vs. linked structures for representing lists and other data types.
- We already know a lot about names referring to objects that have namespaces, but objects referring to objects (of the same class) that refer to objects (of the same class) that refer to objects (of the same class) ... will be something new to learn about and explore.

Linked Node

- We will start with pictures, because pictures are essential to understanding and manipulating these data structures. In lecture, I will show some detailed diagrams of linked lists built with LN objects; then I will remove much redundant information to show more concise and easy to draw pictures. Please copy these down, as they are not in the notes.
- Here is an abbreviated picture: name x refers to an LN object whose value is 5 and whose next is a reference to another LN object whose value 3 and whose next is a reference to another LN object whose value 8 and whose next is .....
- The value None is represented symbolically by the symbol /.



Note that the tails of the arrows (references) are put INSIDE a box representing a place where a name's value is stored. The heads of the arrows refer to an entire LN object, not any particular attribute name/value in it.

Linked Node

- In the following, whenever we see a .name it means "follow the arrow to the object it refers to (all arrows refer to objects) and select the name attribute (in LN objects, all attributes store data). Read the following carefully; everything we do later with linked lists is built on understanding the meaning of .name (something we've been doing with class objects for a while, even if just doing something like writing `print(self.name)` or `self.name = value`.
  1. `x` stores a reference to the first LN object
  2. `x.value` stores a reference to the int object 5
  3. `x.next` stores a reference to the second LN object
  4. `x.next.value` stores a reference to the int object 3 in this second LN object
  5. `x.next.next` stores a reference to the third LN object
  6. x.next.next.value stores a reference to the int object 8 in this third LN object

Linked Node

- Don't memorize this information; understand what .name means and carefully be able to analyze each of these expressions, and any others using .next and .value.
- Typically, we will look at classes for a list/tree data structure as representing just data and no methods. So, we will examine functions defined outside of LN, not methods defined inside the LN class (although most of these functions can be easily written as methods). We will discuss both iterative and recursive versions of most functions.
- See the download that contains all these functions and a simple driver that you can use to test them.

Functions that query/access linked lists

LECTURE 2

Functions that query/access linked lists

- One of the main operations we perform on linked lists (as we do with lists) is to iterate over them, processing all their values. The following function computes the sum of all the values in a linked list ll.

```
def sum_ll(ll):
    sum = 0
    while ll != None:
        sum += ll.value
        ll  =  ll.next
    return sum
```

- Lots of code that traverses (iterates over) linked lists looks similar. In class we will cover (hand simulate) how this code processes the linked list above, with the call sum_ll(x) and see exactly how it is that we visit each node in the linked list and stop processing it at the end.

Functions that query/access linked lists

- There is no special iterator for LN objects (unless we create one, as we will below); LN is just like any other Python class that we write.
- We can also define linked lists recursively and use such a definition to help us write functions that recursively process linked lists.

  (1) None is the smallest linked list: it contains no nodes

  (2) A list node whose next refers to a linked list is also linked list
- So, None is a linked list (of 0 values); a list node whose next is None is a linked list (of 1 value); a list node whose next is a list node whose next is None is a linked list (of 2 values); etc.

Functions that query/access linked lists

- So, we can recursively process a linked list by processing its first LN and then recursively processing the (one smaller) linked list it refers to; recursion ends at None (which is the base case: the smallest linked list). We can recursively compute the sum of linked list by

```
def sum_ll_r(ll):
    if ll == None:
        return 0
    else:
        return ll.value + sum_ll_r(ll.next)
```

Functions that query/access linked lists

Back to the three rules we studied to prove a recursive functions correct:

1) It recognizes and computes the correct result for the smallest (no LN) linked list: it returns 0 which is the sum of no nodes.
2) Each recursive call is on a smaller linked list, which is closer to the base case: The recursive call is on ll.next, which is a linked list with one fewer nodes.
3) Assuming sum_ll_r(ll.next) computes the sum of all values after the node representing the start of the linked list to be processed, this function returns the sum of all the nodes in this linked list: if we add the value of this first node to the sum of the values in all the following nodes in the linked list, then we have computed the sum of all the nodes in the linked list.

Functions that query/access linked lists

- An even simpler traversal of linked lists computes their length. Here are the iterative and recursive functions.

```
def len_ll(ll):
    count = 0
    while ll != None:
        count += 1
        ll = ll.next
    return count
def len_ll_r(ll):
    if ll == None:
        return 0
    else:
        return 1 + len_ll_r(ll.next)
```

Functions that query/access linked lists

- These are simpler than the sum_ll functions: rather than adding the value of each list node, these add 1 to a count for each list node, ultimately computing the number of list nodes in the entire linked list: its length.
- Next let’s look at computing a string representation for a list. There is no standard for how linked lists are represented as strings. We could convert them to look like a normal list: [...] but instead we will use the following form '5->3->8->2->4->None'. Here are the iterative and recursive functions to produce such strings.

Functions that query/access linked lists

- In the iterative method, for each node in the list we append its value followed by '->' into a string, and append just the value 'None' at the end, before returning.

```
def str_ll(ll):
    answer = ''
    while ll != None:
        answer += str(ll.value) + '->'
        ll = ll.next
    return answer + 'None'
```

Functions that query/access linked lists

- In the recursive version, we return 'None' as the base-case, appending the value and '->' in front of the result returned on each recursive call.

```
def str_ll_r(ll):
    if ll == None:
        return 'None'
    else:
        return str(ll.value) + '->' + str_ll_r(ll.next)
```

- In all these examples, the iterative and recursive code have been approximately the same complexity. Let's now look at two other functions: one that converts a standard Python list into a linked list, and one that copies a linked list, observing that the recursive versions are a bit simpler to write and understand.
- BUT, you should hand simulate the iterative methods to understand how/why they work too.

First:

two functions to convert a standard Python list into a linked lists

- In list_to_ll we must treat an empty list specially, returning None: otherwise (for a non-empty list) we can access the first value: l[0]. We make two names (front and rear) to refer to the LN constructed with that value (this LN has next=None).
- We will not change front and eventually return its value (returning a reference to the front of all the list nodes in our list). We add each new value at the end of the list of nodes by extending the node rear refers to: changing its next from None to an actual list node (whose next is None), and then re-adjusting rear to refer to this new end-of-the-list node, extending it as many times as necessary.

```
def list_to_ll(l):
    if l == []:
        return None
    front = rear = LN(l[0])
    for v in l[1:]:
        rear.next = LN(v)
        rear = rear.next
    return front
```

Functions that query/access linked lists

- The recursive version of this function is simpler, and looks pretty much like all the recursive functions that we have seen for linked lists. One interesting feature of note: the recursive call is the second argument to LN's constructor.
- It calls this recursive function and passes a reference to the copied list to the constructor.

```
def list_to_ll_r(l):
    if l == []:
        return None
    else:
        return LN( l[0], list_to_ll_r(l[1:]) )
```

Functions that query/access linked lists

Here is the proof this function is correct

1) It recognizes and computes the correct result for the smallest (empty) list: it returns None, which is an empty linked list.
2) Each recursive call is on a smaller list, which is closer to the base case: The recursive call is on l[1:], the standard one-smaller list.
3) Assuming list_to_ll(l[1:]) returns a linked list with all the values in the l[1:], this function returns a linked list of all the values in the parameter list: it returns a reference to a new list node with the first value in the list (l[0]) and whose .next refers to a linked list with all the values in l[1:].

Functions that query/access linked lists

- To find a value in a linked list (returning a reference to the node that contains that value), we write an iterative method and two recursive variants. Each returns None if the value is not found in the linked list.
- Iteratively, we use ll to traverse the list, looking for avalue: we either find it or "run off the end of the list" and return None

```
def find_ll(ll, avalue):
    while ll != None:
        if ll.value == avalue:
            return ll
        ll = ll.next
    return None
```

Functions that query/access linked lists

- We can also write this more simply as follows (see code below), combining the two conditions for returning a value; when the loop terminates, the test

```
   ll != None and ll.value != avalue
```

is False when the while loop ends, so either ll == None or ll.value == avalue; in both cases returning ll is correct. Note that the short-circuit evaluation of the and operator (and the order of the conjuncts) is critical: we should not follow the reference in ll (with ll.value) until we are sure that ll does not refer to the None object; if it does ll.value would raise an exception

```
def find_ll(ll, avalue):
    while ll!=None and ll.value!=avalue: #short-circuit is critical
        ll = ll.next
    return ll  # ll may be None, or refer to the LN storing avalue
```

Functions that query/access linked lists

- DeMorgan's law in boolean algebra is very important in programming. It says that not (A and B) == not (A) or not (B); also, not (A or B) == not (A) and not (B).
- Each part is negated and the connector flips: and -> or; or -> and.
- So the loop above terminates when its test is False: not (test) is True.

```
not (ll != None and ll.value != avalue)
```

is equivalent to terminating when

```
not (ll != None) or  not(ll.value != avalue)
```

removing the double negative we get

```
ll == None or  ll.value == avalue
```

- We could have also written the continuation test for this loop as

```
not (ll == None or ll.value == avalue)
```

Functions that query/access linked lists

- For the recursive functions, the first uses the simplest base case/non-base case form. If the linked list isn't empty

```
def find_ll_r(ll, avalue):
    if ll == None:
        return None
    else:
        if ll.value == avalue:
            return ll
        else:
            return find_ll_r(ll.next, avalue)
```

Functions that query/access linked lists

- We could replace this entire body by one complicated conditional expression:

```
return (None if ll==None \
                else ll if ll.value==avalue \
                        else find_ll_r(ll.next,avalues)
```

- But this version is very hard to read, and not in the standard recursive form that we have been using.
- As a slight variant (and similar to what we did in the while loop version) we can test both ll == None or ll.value == avalue and in both cases return ll (returning either None of a reference to a list node). Note that if ll == None is True, short-circuit evaluation of "or" means that the expression ll.value == avalue will not need to be evaluated: good thing, too, because accessing ll.value when ll is None would raise an exception.

Functions that query/access linked lists

```
def find_ll_r2(ll, avalue):
    if ll==None or ll.value==avalue: #short-circuit or is critical
        return ll
    else:
        return find_ll_r(ll.next, avalue)
```

- Note that this function is tail recursive and could automatically be written iteratively (as the code above shows).

Functions that query/access linked lists

- We have already examined code that returned the linked list equivalent of a standard Python list. Here is similar code that copies a linked list: constructs new nodes with the same values, arranged in the same order, for a linked list. In the iterative version we again use front/rear to remember the front of the list and extend the rear for each values we traverse in ll.

```
def copy_ll(ll):
    if ll == None:
        return None
    front = rear = LN(ll.value)
    while ll.next != None:
        ll = ll.next
        rear.next = LN(ll.value)
        rear = rear.next
    return front
```

Functions that query/access linked lists

- As we expect, the recursive version is more elegant, and similar to the other recursive code that processes linked lists. It is similar to the code we wrote to translate a Python list into a liked list. Again, here the recursive calls is the second argument to LN's constructor.

```
def copy_ll_r(ll):
    if ll == None:
        return None
    else:
        return LN(ll.value, copy_ll_r(ll.next))
```

Functions that query/access linked lists

- Finally, languages like Java/C++ don't easily support generators. But because Python does, we can easily write a generator that produces all the values in a linked list.

```
def iterator(ll):
    while ll != None:
        yield ll.value
        ll = ll.next
```

Functions that query/access linked lists

- With this code we could print every value in a linked list by writing

```
for v in iterator(ll):
    print(v)
```

- In fact, we could put a variant of this code in the __iter__ method in the LN class as follows:

```
def __iter__(self):
  current = self
  while current != None:
      yield current.value
      current = current.next
```

Functions that query/access linked lists

- With this method, we could write just

```
    for v in ll:
        print(v)
```

although this code (unlike the generator above) will not work when ll refers to None, because there is no __iter__ method in NoneType. But if ll refers to an LN object, the __iter__ code above will iterate through its values.

Functions that command/mutate linked lists

LECTURE 3

Functions that command/mutate linked lists

- All the functions above queried/accessed/created but did not mutate linked lists: no changes were made to .value or .next of an LN object.
- If x refers to the first LN in a linked list, we can add a new value at the front of the linked list by the simple code:

```
    x = LN(newvalue,x)
```

- Now x refers to a new list node, whose value is newvalue, and whose next refers to the LN that x used to refer to: all the nodes in the original linked list.
- Draw a picture with x = None originally or x referring to the linked list above.

Functions that command/mutate linked lists

- We can write the following iterative/recursive functions to append a value at the end of the linked list. In both cases the list is mutated: the last list node has its next changed to refer to a new list node containing the new value (and whose .next is None). But, to handle the case where x is initially empty (stores None), the iterative/recursive functions must return a reference to the front of the list (maybe x itself, or if x stored None, a reference to a one-node linked list storing newvalue). We call these functions like

```
    x = append_ll(x,newvalue)
```

and

```
    x = append_ll_r(x,newvalue)
```

Functions that command/mutate linked lists

- As with list_to_ll and copy, the iterative version needs to remember the front while using ll to traverse down the list, to find the last list node to extend.

```
def append_ll(ll,value):
    if ll == None:
        return LN(value)
    front = ll
    while ll.next != None: # while ll does not refer to the last node...
        ll = ll.next     # terminates when ll.next == None
    ll.next = LN(value) #(know at end: ll.next==None)append value at end
    return front        # return reference to original front of ll
```

Functions that command/mutate linked lists

- The recursive method is again simpler to write.

```
def append_ll_r(ll,value):
    if ll == None:
        return LN(value)
    else:
        ll.next = append_ll_r(ll.next,value)
        return ll
```

Functions that command/mutate linked lists

Here is the proof this function is correct

1. It recognizes and computes the correct result for the smallest (empty) linked list: it returns a reference to a linked list with one node (storing value)
2. Each recursive call is on a smaller linked list, which is closer to the base case of None: the recursive call is on ll.next.
3. Assuming append_ll_r(ll.next,value) returns a reference to a linked list that is one longer than ll.next containing all its list nodes followed by value in the last list node, this function returns a linked list that is one longer than ll containing all its list nodes followed by value in the last list node (by storing in ll.next a reference to the extended linked list and returning the original reference to ll).

Functions that command/mutate linked lists

- ICS-46 studies the execution times of various code applied to data structures. We will do a bit of this analysis in ICS-33, in week 8. Lists in Python allow us to add a value at the end very quickly, but adding a value at the front takes more time: Python must first move the value at index 0 into index 1; the value at index 1 to index 2; ...For linked lists, adding a value at the front is very quick, while adding a value at the rear requires traversing every value in the list (to find the end).
- Depending on how often we perform these two operations, it might be faster to use a list or a linked list.

Functions that command/mutate linked lists

- Here are two simple functions (not iterative or recursive) to mutate a list by adding/removing a value directly after the one referred to by their argument.
- Both functions return None implicitly.

```
def add_after_ll(ll,value):
    # raises an exception if ll is None
    ll.next = LN(value,ll.next)

def remove_after_ll(ll):
    # raises exception of ll (no list) or ll.next
    #(no value to remove) is None
    ll.next = ll.next.next
```

Functions that command/mutate linked lists

- Note that to remove the first value in a linked list, we write

```
    x = x.next
```

- Finally, we could write an append method in the LN class itself, which scanned to the end of the linked list containing the LN, and appending the value there.
- As with generator/__iter__, this method won't wor for an empty linked list, but its class is NoneType, not LN.