CSC148 winter 2016
stack application, linked lists, iteration, mutation — week 4

Danny Heap
heap@cs.toronto.edu
BA4270 (behind elevators)
http://www.cdf.toronto.edu/~heap/148/W14/
416-978-5899

January 31, 2016
Outline

balanced parentheses

linked lists

mutation
In some situations it is important that opening and closing parentheses, brackets, braces match.

’(1 + \[7 - \{8 / 3\}\])’ — good

’(1 + [7 - \{8 / 3\}])’ — bad

Remember, the computer only “sees” one character at a time.
define balanced parentheses:

- a string with no parentheses is balanced
- a string that begins with a left parenthesis ", ends with a right parenthesis ", and in between has balanced parentheses is balanced. Same for brackets ""[...]"" and braces ""..."
- the concatenation of two strings with balanced parentheses is also balanced
why linked lists?

regular Python lists are flexible and useful, but overkill in some situations — they allocate large blocks of contiguous memory, which becomes increasingly difficult as memory is in use. linked list nodes reserve just enough memory for the object value they want to refer to, a reference to it, and a reference to the next node in the list.
linked lists, two concepts

There are *two useful, but different, ways* of thinking of linked list nodes

1. as lists made up of an item (value) and a sub-list (rest)

2. as objects (nodes) with a value and a reference to other similar objects

![Diagram of linked list]

For now, will take the second point-of-view, and design a separate “wrapper” to represent a linked list as a whole.
class LinkedListNode:
    """
    Node to be used in linked list
    """

    def __init__(self, value, next_=None):
        """
        Create LinkedListNode self with data value and successor next_.
        """
        self.value, self.next_ = value, next_
a wrapper class for list

The list class keeps track of information about the entire list — such as its front, back, and size.

class LinkedList:
    ""
    Collection of LinkedListNodes

    === Attributes ==
    @param: LinkedListNode front: first node of this LinkedList
    @param LinkedListNode back: last node of this LinkedList
    @param int size: number of nodes in this LinkedList
        a non-negative integer
    ""

    def __init__(self):
        ""
        Create an empty linked list.

        @param LinkedList self: this LinkedList
        @rtype: None
        ""

        self.front, self.back, self.size = None, None, 0
division of labour

Some of the work of special methods is done by the nodes:

- __str__

- __eq__

Once these are done for nodes, it’s easy to do them for the entire list.
walking a list

Make a reference to (at least one) node, and move it along the list:

```python
cur_node = self.front
while <some condition here...>:  
    # do something here...
    cur_node = cur_node.nxt
```
Check (possibly) every node

cur_node = self.front
while <some condition here...>:
    # do something here...
    cur_node = cur_node.nxt
_getitem_

Should enable things like

```python
>>> print(lnk[0])
5
```

... or even

```python
>>> print(lnk[0:3])
5 4 3 ->|
```
append

We’ll need to change...

- last node
- former last node
- back
- size
- possibly front

draw pictures!
We need to find the second last node. Walk two references along the list.

```
prev_node, cur_node = None, lnk.front
# walk along until cur_node is lnk.back
while <some condition>:
    prev_node = cur_node
    cur_node = cur_node.nxt
```