CS134 Lecture 30:
Special Methods & Linked Lists
Announcements & Logistics

- **HW 9** due tonight @ 10 pm on GLOW
- Short: 6 questions for practice on OOP concepts
- **Lab 9 Boggle**: two-week lab now in progress
  - **Part 2** due May 1/2 (handout posted)
  - Part 2 also has a prelab!
    - Asks you to draw out the Boggle game logic
    - Draw it on a sheet of paper and bring the diagram to lab
    - Make sure it is legible and clear!

Do You Have Any Questions?
Last Time

• Learn how to implement several **special methods** which let us utilize built-in operators in Python for user-defined types

• Discussed general concept of a linked list
  • Recursively defined data structure
  • Elements stored in objects that also store the “next object's" location
Today’s Plan

• Write our own implementation of LinkedList

• Implement functionality (write code) for special methods:
  • __init__
  • __str__
  • __len__
  • __getitem__
  • __contains__

• Discuss at high level (without code) other functionality we may want
But First…

Let’s get a better intuition for how a linked list behaves…

Volunteers?
Our Own Class LinkedList

- Attributes:
  - `_value, _rest`

- **Recursive class**:
  - `_rest` points to another instance of the *same class*
  - Any instance of a class that is created by using another instance of the class is a *recursive class*

![Diagram](image)
Initializing Our LinkedList

class LinkedList:
    """Implements our own recursive list data structure"""

    def __init__(self, value=None, rest=None):
        self._value = value
        self._rest = rest

rest is another instance of our LinkedList class

How do we create an empty list?
Recursive Implementation: __str__

- Let's think about how to implement a string representation of our list
- What is the base case?
  - What if our list has only one item
    - Just return `str` (value) (so if value is int, this return `str(5)` e.g.
  - How do we check if list only has one item in it?
    - `_rest` is `None`
Recursive Implementation: __str__

- Let's think about how to implement a string representation of our list
- What is the base case?
  - What if our list has only one item
    - Just return \texttt{str} (value) (so if value is int, this return \texttt{str(5)} e.g.
    - How do we check if list only has one item in it?
Recursive Implementation: __str__

# str() function calls __str__() method

def __str__(self):
    if self._rest is None:
        return str(self._value)

Python: "is None" vs "== None":
PEP 8 (Style Guide for Python Code) says:
"Comparisons to singletons like None should always be done with 'is' or 'is not', never the equality operators."
Recursive Implementation: `__str__`

```python
# str() function calls __str__() method
def __str__(self):
    if self._rest is None:
        return str(self._value)
    else:
        return str(self._value) + ', ' + str(self._rest)
```

This is recursion since `str` calls `__str__`. The base case is when `self._rest` is `None`.

Diagram:

```
<table>
<thead>
<tr>
<th>_value</th>
<th>_value</th>
<th>_value</th>
<th>_rest</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>3</td>
<td>11</td>
<td>None</td>
</tr>
</tbody>
</table>
```
What if we want to enclose the elements in square brackets [ ]?

**Idea:** Use a helper method that does the same thing as `__str__()` on the previous slide, and then enclose its return in ' [ ]'

```python
def __get_string(self):
    '''Helper method for str of contents'''
    if self._rest is None:
        return str(self._value)
    else:
        return str(self._value) + ', ' + self._rest.__get_string()

def __str__(self):
    return "[" + self.__get_string() + "]"
```
Empty Lists?

- What happens when we call print on an empty LinkedList?
- Do we want a different behavior? How do we change our code?

def __get_string(self):
    # handle empty list
    if self._value is None and self._rest is None:
        return ''  # empty list notation

    elif self._rest is None:  # value is not None
        return str(self._value)

    else:  # neither is None
        return str(self._value) + ', ' + self._rest.__get_string()

def __str__(self):
    return "[" + self.__get_string() + "]"
Special Method: **\_\_len\_\_**

- **\_\_len\_\_(self)**
  - Called when we use the built-in function `len()` in Python on an object `obj` of the class: `len(obj)`
  - We can call `len()` function on any object whose class has the `\_\_len\_\_()` special method implemented
  - We want to implement this special method so it tells us the number of elements in our linked list, e.g. 3 elements in the list below

```
5 -> 3 -> 11 -> None
```
Implementing Recursively

• As our `LinkedList` class is defined recursively, let's implement the `__len__` method recursively
  • Method will return an int (num of elements)
• What is the base case(s)?
• What about the recursive case?
  • Count self (so, +1), and then call `len()` on the rest of the list!

---

```
5
None
3
11
```
Recursive Implementation: `__len__`

```python
# len() function calls __len__() method
def __len__(self):
    # base case: handle empty list first
    if self._value is None and self._rest is None:
        return 0

    # list of length 1
    elif self._rest is None:
        return 1

    # recursive case (larger than 1)
    else:
        # same as return 1 + self._rest.__len__()
        return 1 + len(self._rest)
```

---

```
_value  
 5
_rest

_value  
 3
_rest

_value  
 11
_rest

None
```
Other Special Methods
In Operator: `__contains__`

- `__contains__`(self, val)
  - When we say `if elem in seq` in Python:
    - Python calls the `__contains__` special method on `seq`
    - That is, `seq.__contains__(elem)`
  - If we want the `in` operator to work for the objects of our class, we can do so by implementing the `__contains__` special method
  - Basic idea:
    - “Walk” along list checking values
    - If we find the value we’re looking for, return True
    - If we make it to the end of the list without finding it, return False
    - We’ll do this recursively!
**in** Operator: **__contains__**

- **__contains__(self, val)**
  - When we say `if elem in seq` in Python:
    - Python calls the **__contains__** special method on `seq`
    - That is, `seq.__contains__(elem)`
  - If we want the **in** operator to work for the objects of our class, we can do so by implementing the **__contains__** special method

```python
# in operator calls __contains__() method
def __contains__(self, val):
    if self._value == val:
        return True
    elif self._rest is None:
        return False
    else:
        # same as calling self.__contains__(val)
        return val in self._rest
```
+ Operator: `__add__`

- `__add__(self, other)`
  - When using lists, we can concatenate two lists together into one list using the `+` operator (this always returns a new list)
  - To support the `+` operator in our `LinkedList` class, we need to implement `__add__` special method
  - Make the end of our first list point to the beginning of the other
  - Basic idea:
    - Walk along first list until we reach the end
    - Set `_rest` to be the beginning of second list
    - More recursion!
+ Operator: __add__

- __add__(self, other)
  - When using lists, we can concatenate two lists together into one list using the + operator (this always returns a new list)
  - To support the + operator in our LinkedList class, we need to implement __add__ special method
  - Make the end of our first list point to the beginning of the other

```python
# + operator calls __add__() method
# + operator returns a new instance of LinkedList
def __add__(self, other):
    # other is another instance of LinkedList
    # if we are the last item in the list
    if self._rest is None:
        # set _rest to other
        self._rest = other
    else:
        # else, recurse until we reach the last item
        self._rest.__add__(other)
    return self
```

Note: Technically this does not return a new list. This is more like extend. Let's not worry about this for now!

self is the “head” or beginning of the list. Note that it didn’t change!
Useful list methods:
  .append(), .prepend(), .insert()
Useful List Method: `append`

- `append(self, val)`
  - When using lists, we can add an element to the end of an existing list by calling `append` (note that `append` mutates our list)
  - Basic idea:
    - Walk to end of list
    - Create a new `LinkedList(val)` and add it to the end
Useful List Method: **append**

- **append**(self, val)
  - When using lists, we can add an element to the end of an existing list by calling **append** (note that **append** mutates our list)
  - Basic idea:
    - Walk to end of list
    - Create a new LinkedList(val) and add it to the end
Useful List Method: `append`

- `append(self, val)`
  - When using lists, we can add an element to the end of an existing list by calling `append` (note that `append` mutates our list)
  - This entails setting the `_rest` attribute of the last element to be a new LinkedList with the given value.

```python
def append(self, val):
    # if am at the end of the list
    if self._rest is None:
        # add a new LinkedList to the end
        self._rest = LinkedList(val)
    else:
        # else recurse until we find the end
        self._rest.append(val)
```
Useful List Method: `prepend`

- `prepend(self, val)`
  - We may also want to add elements to the beginning of our list (this will mutate our list, similar to `append`)
  - The `prepend` operation is really efficient, we don’t need to walk through the list at all — just do some variable reassignments.

```python
def prepend(self, val):
    old_val = self._value
    old_rest = self._rest
    self._value = val
    self._rest = LinkedList(old_val, old_rest)
```

![Diagram of LinkedList](image)
Useful List Method: **prepend**

- **prepend**(self, val)
  - We may also want to add elements to the beginning of our list (this will mutate our list, similar to **append**)
  - The **prepend** operation is really efficient, we don’t need to walk through the list at all — just do some variable reassignments.

```python
def prepend(self, val):
    old_val = self._value
    old_rest = self._rest
    self._value = val
    self._rest = LinkedList(old_val, old_rest)
```

- **val**: `self._value`
- **old_val**: `self._value`
- **old_rest**: `self._rest`
- **new_val**: `val`
- **new_rest**: `self._rest`
Useful List Method: `insert`

- `insert(self, val, index)`
  - Finally, we want to allow for insertions at a specific index.
  - Basic idea:
    - If the specified index is 0, we can just add to the beginning (easy!)
    - Otherwise, we walk to the appropriate index in the list, and reassign the `_rest` attribute at that location to point to a new LinkedList with the given value, and whose `_rest` attribute points to the linked list it is displacing.

```
_value
val
_rest

_value
5
_rest

_value
3
_rest

_value
11
_rest

None
```
Useful List Method: `insert`

- `insert(self, val, index)`
  - Finally, we want to allow for insertions at a specific index.
  - Basic idea:
    - If the specified index is 0, we can just add to the beginning (easy!)
    - Otherwise, we walk to the appropriate index in the list, and reassign the `_rest` attribute at that location to point to a new LinkedList with the given value, and whose `_rest` attribute points to the linked list it is displacing.
Useful List Method: `insert`

- `insert(self, val, index)`
  - If the specified index is 0, we can just use the `prepend` method.
  - Otherwise, check to see if we're at end of the list
  - Otherwise, we walk to the appropriate index in the list, and perform the insertion

```python
def insert(self, val, index):
    # if index is 0, we found the item we need to return
    if index == 0:
        self.prepend(val)
    # elif we have reached the end, so just append
    elif self._rest is None:
        self._rest = LinkedList(val)
    # else we recurse until index reaches 0
    else:
        self._rest.insert(val, index - 1)
```
Operator: `__getitem__`, `__setitem__`

- `__getitem__(self, index)` and `__setitem__(self, index, val)`

- With lists, we can get or set the item at a specific index by using the `[]` operator:
  - **get**: `val = mylist[1]`
  - **set**: `mylist[2] = new_val`

- To support the `[]` operator in our `LinkedList` class, we need to implement `__getitem__` and `__setitem__`

- Basic idea:
  - Walk out to the element at `index`
  - Get or set value at that index accordingly
  - Recursive!
mylist[2]

• implicitly: `mylist.__getitem__(2)`
  • When using lists, we can get the item at a specific index by using the `[]` operator (e.g., `val = mylist[2]`)

• What might be the base case?
  We've reached the index, return the value!

• What might be the recursive case?
  Cut one item off the front of our list, and subtract one from our index. Keep looking!
mylist[2]

• implicitly: `mylist.__getitem__(2)`

• When using lists, we can get the item at a specific index by using the `[]` operator (e.g., `val = mylist[2]`)

```python
def __getitem__(self, index):
    if index == 0:
        return self._value
    else:
        return self._rest[index - 1]
```

```
my_lst = LinkedList(5, LinkedList(3, LinkedList(11)))
my_lst[2]
```

```
__getitem__(2)
...     return LinkedList(3, LinkedList(11))[1]
```

```
__getitem__(0)
...     if index == 0:
            return LinkedList(11)._value
```

```
__getitem__(1)
...     return LinkedList(11)[0]
```
[] Operator: `__getitem__`, `__setitem__`

- `__getitem__`(self, index) and `__setitem__`(self, index, val)

- With lists, we can get or set the item at a specific index by using the `[]` operator (e.g., `val = mylist[1]` or `mylist[2] = new_val`)

# [] list index notation also calls `__setitem__()` method
# index specifies which item we want, val is new value

def __setitem__(self, index, val):
    # if index is 0, we found the item we need to update
    if index == 0:
        self._value = val
    else:
        # else we recurse until index reaches 0
        # remember that this implicitly calls `__setitem__`
        self._rest[index - 1] = val
== Operator: __eq__

- __eq__(self, other)
  - When using lists, we can compare their values using the == operator
  - To support the == operator in our LinkedList class, we need to implement __eq__
  - We want to walk the lists and check the values
  - Make sure the sizes of lists match, too
== Operator: __eq__

- __eq__(self, other)
- When using lists, we can compare their values using the == operator
- To support the == operator in our LinkedList class, we need to implement __eq__

```python
# == operator calls __eq__() method
# if we want to test two LinkedLists for equality, we test
# if all items are the same
# other is another LinkedList
def __eq__(self, other):
    # If both lists are empty
    if self._rest is None and other.get_rest() is None:
        return True
    # If both lists are not empty, then value of current list elements
    # must match, and same should be recursively true for
    # rest of the list
    elif self._rest is not None and other.get_rest() is not None:
        return self._value == other.get_value() and self._rest == other.get_rest()
    # If we reach here, then one of the lists is empty and other is not
    return False
```
Other Special Methods

• There are many other “special” methods in Python.

• __eq__ (self, other): x == y
• __ne__ (self, other): x != y
• __lt__ (self, other): x < y
• __gt__ (self, other): x > y
• __add__ (self, other): x + y
• __sub__ (self, other): x - y
• __mul__ (self, other): x * y
• __truediv__ (self, other): x / y
• __pow__ (self, other): x ** y
• There are others!
Looking Ahead

- In CS136 you’ll see doubly linked lists! Overcomes some of the inefficiencies of singly linked lists