CS 134: Special Methods & Linked Lists
Announcements & Logistics

- **Lab 7 and 8** feedback coming soon
- **HW 8** due tonight at 11 pm (please don’t forget the week!)
- **Lab 9 Boggle**
  - **Parts 1 & 2 (BoggleLetter & BoggleBoard)** due Wed/Thur
  - We will run our tests and return automated feedback, but we won't assign grades
  - **Part 3 (BoggleGame)** due May 4/5

Do You Have Any Questions?
Demo!
Last Time

• Finished implementation of Tic Tac Toe game
  • (Fun?) Application of object-oriented design and inheritance
• Designed to help with the Boggle lab
• Advice as you make your way through the lab:
  • Isolate functionality and test often (use __str__ to print values as needed)
  • Check individual methods
  • **Discuss logic with partner before writing any code**
  • Worry about common cases first, but don’t forget the “edge” cases
Today’s Plan

• We will build a **recursive list class**
  • Our own implementation of list
• On the way, we will learn how to implement some **special (aka magic) methods** which override the behavior of existing operators/functions in Python
  • We have already seen some examples: **__str__**
  • Automatically called when we use the `str()` or `print()` function
• Today we will see:
  • **__len__** (called when you use `len` function)
  • **__contains__** (called when we use `in` operator)
  • **__getitem__** (called when we index into a sequence using `[]`)
  • Many more!
Python's Built-in list Class

- A class with methods (that someone else implemented)
- `pydoc3 list`
What exactly is a list?

- A container for a sequence of values
  - Recall that sequence implies an order
- Another way to think about this:
  - A chain of values, or a **linked list**
  - Each value has something after it: the rest of the sequence (recursion!)
- How do we know when we reach the end of our list?
  - Rest of the list is **None**
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*
Initializing Our LinkedList

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

```python
myList = LinkedList(5, LinkedList(3, LinkedList(11)))
```

```python
type(myList)
```

```
__main__.LinkedList
```
Special Methods (Review)

- `__init__(self, val)`
  - When is it called?
    - When we `create` an instance (object) of the class
  - Can also call it as `obj.__init__(val)` (where `obj` is an instance of the class)

- `__str__(self)`
  - When is it called?
    - When we `print` an instance of the class using `print(obj)`
    - Also called whenever we convert an instance of the class to `str`, that is, when we call `str` function on it: `str(obj)`
    - Can also call it as `obj.__str__()`
Recursive Implementation: `__str__`

```
# 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)
```

```
myList = LinkedList(5, LinkedList(3, LinkedList(11)))

print(myList)  # testing __str__
```

```
5, 3, 11
```

This is recursion! Since str calls itself. The base case is implicit when self._rest is `None`.

Diagram:
```
_\text{value}  \rightarrow _\text{value}  \rightarrow _\text{value}  \rightarrow \text{None}
5               3               11
_\text{rest}    _\text{rest}    _\text{rest}
```

This is a linked list with values 5, 3, and 11, where each `None` represents the end of the linked list.
Recursive Implementation: `__str__`

- What if we want to enclose the elements in the square brackets `[ ]`?
- It helps to have a helper method that does the same thing as `__str__()` on the previous slide, and then call that helper between concatenating the square brackets.

```python
def __strElements(self):
    if self._rest is None:
        return str(self._value)
    else:
        return str(self._value) + ', ' + self._rest.__strElements()

def __str__(self):
    return "[" + self.__strElements() + "]"
```

```python
myList = LinkedList(5, LinkedList(3, LinkedList(11)))

print(myList) # testing __str__
```

`[5, 3, 11]`
An Aside: __repr__

- In Labs 8 and 9, we included __repr__ methods in your starter code.
- You do not need to worry about them! (Just ignore these methods in Lab 9!)
- For your reference, here is a quick summary:
  - Like __str__(), __repr__() returns a string, useful for debugging.
  - Unlike __str__(), the format of the string is very specific.
  - __repr__() returns a string representation of an instance of a class that can be used to recreate the object.

```python
# repr() function calls __repr__() method
# return value should be a string that is a valid Python expression that can be used to recreate the LinkedList

def __repr__(self):
    return "LinkedList({}, {})".format(self._value, repr(self._rest))
```

```
In [62]: myList = LinkedList(5, LinkedList(3, LinkedList(11)))
In [64]: myList  # testing __repr__
Out[64]: LinkedList(5, LinkedList(3, LinkedList(11, None)))
```

Notice we did not say print(myList) here.
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

```
_\text{value} \quad _\text{rest} \quad _\text{value} \quad _\text{rest} \quad _\text{value} \quad _\text{rest}
\hline
5 \quad \text{None}
3 \quad 11
\hline
```
Implementing Recursively

• As our LinkedList class is defined recursively, let's implement the __len__ method recursively
  • Example of fruitful recursion that returns an int (num of elements)
• What is the base case?
• What about the recursive case?
  • Count self (so, +1), and then call len() on the rest of the list!
Recursive Implementation: \_\_len\_\_

```python
# len() function calls __len__() method
def __len__(self):
    # base case: i'm the last item
    if self._rest is None:
        return 1
    else:
        # same as return 1 + self._rest.__len__()
        return 1 + len(self._rest)

_value
_5

_value
_3

_value
_11

_value
None
```

Note: It is preferred to use is or is not operators (as opposed to == or !=) when comparing a user-defined object to a None value. This is because __eq__ and __ne__ are also special methods that can be made to behave differently for classes.
What About Other Special Methods?

- What other functionality does the built-in list have in Python that we can incorporate into our own class?
  - Can check if an item is in the list (in operator): `__contains__`
  - Concatenate two lists using `+`: `__add__`
  - Index a list with `[ ]`: `__getitem__`
  - **Set** an item to another val, e.g. `myList[2] = "hello"`: `__setitem__`
  - Compare the values of two lists for equality using `==`: `__eq__`
  - **Reverse/sort** a list
  - **Append** an item to the list: `append` method
  - Many others!
- Let's try to add some of these features to our `LinkedList`
**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)`
  - Thus 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)
  • Thus 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
```
Operator: `__getitem__`, `__setitem__`

- `__getitem__(self, index)` and `__setitem__(self, index, val)`
  - When using lists, we can get or set the item at a specific index by using the `[]` operator (e.g., `val = mylist[1]` or `mylist[2] = newVal`)
  - 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!
Operator: `__getitem__`, `__setitem__`

- `__getitem__(self, index)` and `__setitem__(self, index, val)`
- When using lists, we can get or set the item at a specific index by using the `[]` operator (e.g., `val = mylist[1]` or `mylist[2] = newVal`)

```python
# [] list index notation calls __getitem__() method
# index specifies which item we want
def __getitem__(self, index):
    # if index is 0, we found the item we need to return
    if index == 0:
        return self._value
    else:
        # else we recurse until index reaches 0
        # remember that this implicitly calls __getitem__
        return self._rest[index - 1]
```
Operator: `__getitem__`, `__setitem__`

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

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

```python
# [] 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.getRest() 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.getRest() is not None:
        return self._value == other.getValue() and self._rest == other.getRest()

    # If we reach here, then one of the lists is empty and other is not
    return False
```
Many Other Special Methods

• Examples:
  • `__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`
  • ...

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` (mutates our list)
  - Thus `append` is similar to `__add__`, except we are only adding a single element rather than an entire list (so it’s a bit easier to accomplish)

- Basic idea:
  - Walk to end of list
  - Create a new `LinkedList(val)` and add it to 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` (mutates our list)
  - Thus `append` is similar to `__add__`, except we are only adding a single element rather than an entire list (so it’s a bit easier to accomplish)

```python
# append is not a special method, but it is a method
# that we know and love from the Python list class.
# unlike __add__, we do not return a new LinkedList instance
def append(self, val):
    # if am at the list item
    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)
```
Making our List an Iterable

• We can iterate over a Python list in a for loop

• It would be nice if we could iterate over our LinkedList in a for loop

• This won’t quite work right now

```python
In [108]:
   for item in myList:
      print(item)

5
3
11
```

```
TypeError
<ipython-input-108-4bf86db75685> in <module>
----> 1 for item in myList:
      2   print(item)

<ipython-input-104-8a5ab5d1919c> in __getitem__(self, index)
     68     # else we recurse until index reaches 0
     69     # remember that this implicitly calls __getitem__
---  70             return self._rest[index - 1]

TypeError: 'NoneType' object is not subscriptable
```
Making our List an Iterable

• We can iterate over a Python list in a `for loop`

• It would be nice if we could iterate over our LinkedList in a for loop

• This won’t quite work right now

• What do we need?
  • Next time we will discuss the special method `__iter__`
  • We will look behind the scenes at a for loop and see how it works!