CSCI 134 Fall 2021:

Special Methods & Linked Lists

Nov 17, 2021

Shikha Singh, 9AM
Jeannie Albrecht, 10AM
Announcements & Logistics

- **Lab 7** feedback returned, **Lab 8** feedback coming soon
- **No homework** this week
- **HW 8**: Question 2 clarification; points given back to everyone
- **Lab 9 Boggle**
  - **Parts 1 & 2 (BoggleLetter & BoggleBoard)** due tonight/tomorrow @10pm
  - We will run our tests and return automated feedback (similar to Lab 4 part 1), but we won't assign grades (you are allowed to correct your errors)
  - **Parts 3 & 4 (BoggleWords & Game)** are due Dec 1/2 the week after Thanksgiving break
    - Windows: instructions on how to get graphics to work on handout
- Lab next week: Keep working on Boggle, optional (but encouraged) **attendance**

Do You Have Any Questions?
Last Time and Takeaways

• Finished implementation of Tic Tac Toe game
  • (Fun?) Application of object-oriented design and inheritance
• Designed to help you with the Boggle lab
  • How is your progress with BoggleLetter and BoggleBoard?
• Advice as you make your way through the lab:
  • Don’t forget the bigger picture as you implement individual methods
  • Think holistically about how the objects/classes work together
  • Isolate functionality and test often (use __str__ and __repr__ to print values as needed)
  • 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 our own **recursive list class**
  • Our own implementation of list
• On the way, we will learn how to implement some **special methods**
  which override the behavior of existing operators/functions in Python
  • Already seen some examples: **__str__, __init__, __repr__**
• Today we will see:
  • **__len__** (called when you use **len** function)
  • **__contains__** (called when we use **in** operator)
  • **__getitem__** (called when we index [ ] into an object)
  • Many more!
Python's Built-in list Class

- A class with methods (that someone else implemented)
- `pydoc3 list`
What 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
- 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
In [1]: class LinkedList:
    """Implements our own recursive list data structure"""
    __slots__ = ['_value', '_rest']

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

In [2]: myList = LinkedList(5, LinkedList(3, LinkedList(11)))

In [3]: type(myList)

Out[3]: __main__.LinkedList
```

rest is another instance of our LinkedList class
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
    - 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__`

```python
@property
def value(self):
    return self._value

def __str__(self):
    # Alternate: return "value: {}, rest: {}".format(self.value, self.rest)
    # format calls str
    return "value: " + str(self.value) + ", rest: (" + str(self.rest) + ")"
```

```
In [2]: myList = LinkedList(5, LinkedList(3, LinkedList(11)))

In [3]: print(myList)  # testing __str__
    value : 5, rest : (value : 3, rest : (value : 11, rest : (None)))
```
Special Method  \_\_len\_\_

-  \_\_len\_(self)
  - Called when we use the built-in function \texttt{len} in Python on an object \texttt{obj} of the class: \texttt{len(obj)}
  - We can call \texttt{len} function on any object whose class has the \_\_len\_\_ special method already 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

\begin{figure}
\centering
\includegraphics[width=\textwidth]{linked_list_diagram.png}
\end{figure}
Implementing Recursively

• As our LinkedList class is defined recursively, let's implement the __len__ method recursively
  • Example of fruitful recursion
• What is the base case?
• What about the recursive case?
  • Count yourself, and then call len on the rest of the list!

```
5
_\text{rest}
\text{value}
3
_\text{rest}
\text{value}
11
_\text{rest}
\text{value}
None
```
Recursive Implementation: __len__

```python
def __len__(self):
    # base case: I'm last item
    if self.rest is None:
        return 1
    else:
        # same as return 1 + self.rest.__len__()
        return 1 + len(self.rest)
```

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__` methods 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__
  • Append an item to the list: append
  • Set an item to another val, e.g. myList[2] = “hello”: __setitem__
  • Compare the values of two lists for equality: __eq__ (==)
  • Reverse/sort a list
  • Many others
• Let's try to add some of these features to our LinkedList
in Operator: __contains__

• __contains__(self, val)
  • When we say elem in seq in Python:
    • Behind the scenes it 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 you find the value you’re looking for, return True
  • If you 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 `elem in seq` in Python:
    - Behind the scenes it 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
def __contains__(self, val):
    if self.value == val:
        return True
    elif self.rest is None:
        return False
    else:
        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.
  - To support the `+` operator in our `LinkedList` class, we need to implement `__add__`.
  - We want to 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.
    - Recursive!
+ Operator: `__add__`

- `__add__(self, other)`
  - When using lists, we can concatenate two lists together into one list using the `+` operator
  - To support the `+` operator in our `LinkedList` class, we need to implement `__add__`
  - We want to make the end of our first list point to the beginning of the other

```python
def __add__(self, other):
    if self.rest is None:
        self._rest = other
    else:
        self.rest.__add__(other)
    return self
```

`self` is the “head” or beginning of the list. Note that it didn’t change!
Operator: `append`

`append(self, val)`

- When using lists, we can add an element to the end of an existing list by calling `append`
- 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
Operator: `append`

- `append(self, val)`
  - When using lists, we can add an element to the end of an existing list by calling `append`
  - 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
def append(self, val):
    if self.rest is None:
        self._rest = LinkedList(val)
    else:
        self.rest.append(val)
```
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`)
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
  - Can we do recursively?
    - Why is it difficult?
      - Need to keep track of when either list "runs out" of elements (be careful about base cases)
    - Let us use a loop instead, and we'll return to the recursive approach afterwards
== 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

```python
def __eq__(self, other):
    # while both have elements, compare and move one forward
    while self and other and self.value == other.value:
        self = self.rest
        other = other.rest
    # if both are empty (and same so far)
    if not self and not other:
        return True
    # otherwise
    return False
```
Making our List an Iterable

• We can iterate over a Python list in a **for loop**
• It would be nice if we can iterate over our own linked list in a for loop
• What do we need?
  • On Friday, we will discuss the special method `__iter__`
  • We will look behind the scenes of a for loop and how it works!
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`