CSCI 134 Fall 2021:

Linked Lists & Iterators

Nov 19, 2021

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

• **Lab 8** feedback coming soon!

• **No homework** this week

• **Lab 9 Boggle**
  • We’ll run tests on Parts 1 & 2 and return automated feedback soon
  • **Parts 3 & 4 (BoggleWords & Game)** are due Dec 1/2 the week after Thanksgiving break

• **Lab next week**: Boggle, optional (but encouraged) attendance

• No herd meetings this weekend

• No TA hours after Monday 11/22 evening

Do You Have Any Questions?
Working from Off Campus

• **Connect to VPN:**
  • Make sure this works *before* you leave campus!
  • To connect to evolene from off campus, **you need to connect to VPN to git push/pull**
  • You can commit without VPN

• **Do not upgrade your computer for now (especially Mac OS)**
  • Latest updates “break” I34 setup

Do You Have Any Questions?
Last Time

- Started the implementation of our own linked list class
  - Why? Help us understand what's happening in Python's built-in classes
  - A glimpse of data structure design (precursor to CS136)
- Implemented `append` method and the following special methods:
  - `__init__`, `__str__`, `__len__`, `__contains__` (in), `__add__` (+),
  - `__getitem__`, `__setitem__` ([ ] brackets to get/set value at index)
  - `__eq__` (==)

```
_5 = Value
_3 = Value
_11 = Value
None
```
Today

• Wrap up our linked list class:
  • Revisit and review `__eq__`
• Discuss how we can turn our LinkedList into an "iterator"
  • This will allow us to iterate over our lists in a for loop
  • We'll also look behind the scenes at how for loops work in Python
• Implement more special methods: `__iter__` and `__next__`

```
_5_ _value |
|______|
|value _rest |
|__________|
|__________|
|__________|
|__________|

_3_ _value |
|______|
|value _rest |
|__________|
|__________|
|__________|
|__________|
|__________|

_11_ _value |
|______|
|value _rest |
|__________|
|__________|
|__________|
|__________|
|__________|
|__________|

None
```
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__.
  - Conceptually, we want to “walk” the two lists and compare their values.
  - Can we do this recursively?
    - Yes! But there are several different cases to worry about.
    - Let us look at both an iterative and recursive approach.
== Operator: `__eq__`

- **Iterative approach**

  - Consider all possible cases as we walk down our lists:
    - If any pairs of the values are different: return False
    - If one list "runs out" of elements (reaches end): return False
    - If you make it to the end in both (and everything is same), return True!

```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
```
== Operator: __eq__

- Recursive approach
  - Same idea! But let’s do it recursively.

```python
def __eq__(self, other):
    # base case 1: at the end of both, check val
    if self.rest is None and other.rest is None:
        return self.value == other.value

    # one of them ends before other
    if self.rest is None or other.rest is None:
        return False

    # recursive case
    return (self.value == other.value
            and self.rest == other.rest)
```
Making our List *Iterable*

- Our implementation of a *LinkedList* from last class already allows us to iterate over our lists using a while loop or a for loop and the indices in `range(len(myList))`

- How do we usually iterate over sequences in Python?
  - Standard approach is to iterate directly (without using indices):
    ```python
    for elem in seq
    <do something>
    ```

- What happens when we try to iterate over our LinkedList in this way?

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

for item in newList:
    print(item)
```

```
TypeError: 'LinkedList' object is not iterable
```

How can we make our LinkedList iterable?
Making our List **Iterable**

- A Python object is considered **iterable** if it supports the `iter()` function: that is, the special method `__iter__` is defined
  - All **sequences** in Python are **iterable**, e.g., strings, lists, ranges, tuples, even files
  - We can iterate over an **iterable** directly in a for loop
  - When an **iterable** is passed to the `iter()` function, it creates an **iterator**
    - An **iterator** object can generate values from the sequence **on demand**
      - This is accomplished using the `next()` function (and `__next__` method) which simply provides the "next" value in the sequence
      - We have already seen a few iterators that used `next()`: file objects, CSV reader objects, etc
For loop: Behind the Scenes

- A for loop in Python iterates directly over `iterable` objects. For example:

```python
# a simple for loop to iterate over a list
for item in numList:
    print(item)
```

- Behind the scenes, the for loop is simply a while loop in disguise, driving iteration within a `try-except` statement. The above loop is really:

```python
try:
    it = iter(numList)
    while True:
        item = next(it)
        print(item)
except StopIteration:
    pass
```

Call the `iter` method on object to get an iterator

Access the `next` item if it exists, then print it
As Aside: **try–except** blocks

- The try/except block has the following form:

```python
try:
    <possibly faulty suite>
except <error>:
    <cleanup suite>
```

- The **<possibly faulty suite>** is a collection of statements that has the potential to fail and generate an error:
  - If the failure occurs, rather than causing the program to crash, the statements inside the `except` branch are run.

- You can even have more than one `except`, to handle different types of errors.

- Fortunately, Python handles this automatically for us in for loops!
Python's Built-in Iterables

• We can create **iterators** for lists/strings/tuples by passing them to **iter()**
  
  • Benefit? We can generate values on demand (one at a time)
  
  • An **iterator** maintains state between calls to **next()**
  
  • Once all values in the sequence have been iterated over, the **iterator** "runs dry" (and becomes empty)
    
    • We can only iterate over values once (unless we create another iterator)
Python's Built-in Iterators

- File and CSV objects in Python are **iterators**: return lines one by one on demand (by calling to `next()`)

- Advantages:
  - Allows us to skip header rows/lines
  - Allows us to traverse extremely large data sets “lazily”

```python
In [10]: fsuper = open('superpowers.txt')

In [11]: next(fsuper)
Out[11]: 'Superhuman intelligence\n'

In [12]: next(fsuper).strip()
Out[12]: 'Reading minds'

In [13]: next(fsuper).strip () # do repeatedly
Out[13]: 'Teleport'

In [15]: fort = open('fortunes.txt')

In [16]: next(fort)
Out[16]: 'You will become friends with the ghost in Lehman.\n'

In [17]: next(fort).strip()
Out[17]: 'There will be three mountain days'

In [18]: next(fort).strip()
Out[18]: 'You will get hit by truck and survive'
```
Creating an Iterator

• To create an iterator for a class we need to implement two methods:
  • `__iter__()` which is called to creates the iterator
  • `__next__()` which is called to advance to the next value

• The key aspect of creating iterators: maintaining state to keep track of where you are currently in the sequence (and what is next value that should be returned)

• Thus, `__iter__()` should always "reset" the current state to the beginning, and `__next__()` should update this state each time its called

• We will investigate two methods for creating an iterator for our linked list:
  • Method 1: Implement `__iter__` and `__next__` directly
  • Method 2: Use generators (simpler and preferred way in Python)
Creating an Iterator: Method 1

• Note: We added a new attribute '_current' to __slots__

  • _current keeps track of where we are in the iterator

• Reminder: We have implemented properties for value, and rest

```python
def __iter__(self):
    # set current to head
    self._current = self
    return self
def __next__(self):
    if self._current is None:
        raise StopIteration
    else:
        val = self._current.value
        self._current = self._current.rest
        return val
```

```
In [2]:
testList = LinkedList()
testList.append("w")
testList.append("o")
testList.append("o")
testList.append("t")
for char in testList:
    print(char)
```

```text
w
o
o
t
None
```
Creating an Iterator: Method 1

• Note: We added a new attribute '_current' to __slots__

• _current keeps track of where we are in the iterator

• Reminder: We have implemented properties for value, and rest

```python
def __iter__(self):
    # set current to head
    self._current = self
    return self

def __next__(self):
    if self._current is None:
        raise StopIteration
    else:
        val = self._current.value
        self._current = self._current.rest
        return val
```

In [2]:
testList = LinkedList()
testList.append("w")
testList.append("o")
testList.append("o")
testList.append("t")

for char in testList:
    print(char)

```

w
o
o
t

None
```
Creating an Iterator: Method 1

- Note: We added a new attribute \_current\ to \_slots\.
  - \_current\ keeps track of where we are in the iterator.
- Reminder: We have implemented properties for \texttt{value}, and \texttt{rest}.

```python
def \_iter\_(self):
    # set current to head
    self._current = self
    return self
def \_next\_(self):
    if self._current is None:
        raise StopIteration
    else:
        val = self._current.value
        self._current = self._current.rest
        return val
```

In [2]:
```python
testList = LinkedList()
testList.append("w")
testList.append("o")
testList.append("o")
testList.append("t")
for char in testList:
    print(char)
```
Creating an Iterator: Method 1

- Note: We added a new attribute `_current` to `__slots__`

- `_current` keeps track of where we are in the iterator

- Reminder: We have implemented properties for `value`, and `rest`

```python
def __iter__(self):
    # set current to head
    self._current = self
    return self

def __next__(self):
    if self._current is None:
        raise StopIteration
    else:
        val = self._current.value
        self._current = self._current.rest
        return val
```

```python
In [2]:
testList = LinkedList()
testList.append("w")
testList.append("o")
testList.append("o")
testList.append("t")
for char in testList:
    print(char)
```

```
<table>
<thead>
<tr>
<th>_value</th>
<th>_value</th>
<th>_current</th>
<th>_value</th>
</tr>
</thead>
<tbody>
<tr>
<td>w</td>
<td>o</td>
<td>o</td>
<td>t</td>
</tr>
<tr>
<td>_rest</td>
<td>_rest</td>
<td>_rest</td>
<td>_rest</td>
</tr>
</tbody>
</table>
```

None
Creating an Iterator: Method 2

• Python provides a way to create iterators without having to worry about keeping state within our data structures or implementing the `__next__` method
  • Through the use of generators

• A generator is a function that can "pause" or "freeze" its state and yield values lazily
  • Automatically supports `iter` and `next` functions
  • Perfect for creating iterators, but can also do much more!
Generators
(a Pythonic way to create iterators)
Generators and Yield

- A function that has a `yield` statement in it is called a generator.
- A `yield` statement changes the behavior of the function.
  - A yield statement causes the function to pause and hold state.

```
>>> def genF(num):
    yield num

>>> g = genF(10)
>>> g
<generator object genF at 0x10a55ac50>
```  

```
>>> def simpleF(num):
    return num

>>> f = simpleF(10)
>>> f
10
```  

Invoking a generator function creates a **generator object**.

Invoking a regular (non-generator) function **returns the output**.
Generator: Simple Example

- **genF()** does nothing other than yield the value that is passed in as an argument. However, invoking it like a “normal” function does not produce a returned value, but instead creates a generator object.

- If we call the `next()` function on the generator object `g`, it “yields” or “produces” the expected value. After it, the generator `g` is exhausted.

```python
A generator function

def genF(num):
    yield num

A “normal” function

def simpleF(num):
    return num
```

```python
>>> g = genF(10)
>>> g
<generator object genF at 0x10a55ac50>
>>> next(g)
10
>>> next(g)
File "<stdin>", line 1, in <module>
  StopIteration
```

Calling `next` on it again throws a `StopIteration` exception.
Understanding Yield

- If a `yield exp` statement is reached, the function's state is **frozen**, and the value of the expression `exp` is returned after a subsequent call to `next()`.
- That is, all local state of variables is retained in the function frame, and execution is “resumed” when `next()` is invoked again.
- The control flow proceeds exactly where it left off.
- A function can contain multiple yield (along with return) statements.

**Yield vs Return**

- **Similarity**: Both `yield` and `return` will return a value from a function.
- **Difference**: A `return` statement terminates the function entirely. A `yield` statement **pauses** the function (saving all its state) and continues execution later.
Generators: More Examples

```python
In [14]: def anotherGen():
    yield "a"
    yield "e"
    yield "z"

In [15]: g = anotherGen()

In [16]: next(g)
Out[16]: 'a'

In [17]: next(g)
Out[17]: 'e'

In [18]: next(g)
Out[18]: 'z'

In [19]: next(g)
```

Generator is now exhausted (automatic `StopIteration` raised)
Mechanics of Generators

- **Generators** contains one or more `yield` statements.

- When a generator function is invoked, it returns a generator object (which is a type of iterator) but does not run any code.

- When `next()` is called on the generator object, execution runs up to the first `yield` statement.

- When a generator function *yields* a value, execution is paused until `next()` is called again.
  - Local variables and their states are saved.

- Finally, when the function terminates (either by reaching an explicit return statement or reaching the end of function body), a `StopIteration` is raised automatically on further `next()` calls.

- Such exceptions are handled automatically if iterating over the generator object in a for loop.
On Demand Fortune Generator!

```python
In [23]:
    import random
def randomFortunes():
        """reads in a filename 'fortunes.txt' and generates
        random lines (a fortune) from it one at a time""
        fortunes = [fortune.strip() for fortune in open('fortunes.txt')]
        while True:
            index = random.randint(0, len(fortunes)-1)
            yield fortunes[index]

In [24]:
    fortuneGen = randomFortunes()

In [25]:
    next(fortuneGen)

Out[25]: 'You will make a new friend today'

In [26]:
    next(fortuneGen)

Out[26]: 'You are an angel. Beware of those that collect feathers.'

In [28]:
    for _ in range(5):
        print(next(fortuneGen))

There will be three mountain days
Look behind you
You will fulfill your potential.
You will get a 4.0 GPA this semester
Enlightenment comes upon you in the near future
LinkedList Iterator via Generator

• We don't need to maintain state through an global attribute
• We don't need to define a __next__ method
• Can just write an __iter__ method that maintains current state and yields the next value
  • Don't need to raise StopIteration exception
  • Raised automatically when we run out of elements!

```python
def __iter__(self):
    # set current to head
    current = self
    # while current is not None
    while current:
        yield current.value
        current = current.rest
```
Using our New Iterable LinkedList

In [2]:
```python
testList = LinkedList()
testList.append("w")
testList.append("o")
testList.append("o")
testList.append("t")
```

In [3]:
```python
next(testList)  # will this work?
```

```
TypeError
Traceback (most recent call last)
/var/folders/h8/n5myy3jdld7cfv42cw42flt80000gn/T/ipykernel_44602/3718056709.py in <module>
----> 1 next(testList)  # will this work?

TypeError: 'LinkedList' object is not an iterator
```

In [4]:
```python
listIterator = iter(testList)
```

In [5]:
```python
next(listIterator)
```

Out[5]:
```
'w'
```

In [6]:
```python
next(listIterator)
```

Out[6]:
```
'o'
```

In [7]:
```python
next(listIterator)
```

Out[7]:
```
'o'
```

In [8]:
```python
next(listIterator)
```

Out[8]:
```
't'
```
What's Next in CS134

• Pre-midterm
  • Emphasis on basics of programming
  • Python's built-in data structures: lists, dictionaries, tuples, sets
  • Script vs module and functions

• Post-midterm
  • Advanced programming topics (including today's topics!)
  • Recursive functions
  • Classes and OOP
  • Recursive data structures
  • Brief introduction to searching/sorting and efficiency analysis
  • JAVA!!!