CS 134: Iterators
Announcements & Logistics

- **Lab 7 and 8** feedback coming soon!
- **No homework** this week
- **Lab 9 Boggle**
  - **Parts 1 & 2 (BoggleBoard and BoggleLetter)** due today/tomorrow
  - **Parts 3 (BoggleGame)** due next week
- **Lab next week:** More Boggle!

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 several special methods:
  • `__init__`, `__str__`, `__len__`, `__contains__` (in), `__add__` (+)
  • `__getitem__`, `__setitem__` ([ ] brackets to get/set value at index)
Today

• Wrap up our linked list class:
  • Look at __eq__, prepend, append, insert
• Discuss how we can turn our LinkedList into an “iterable” object
  • 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__

```
_value
5
_rest

_value
3
_rest

_value
11
_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__
  - 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** (note that **append** mutates our list)
  - Basic idea:
    - Walk to end of list
    - Create a new **LinkedList(val)** and add it to the end

![Diagram showing the Append method applied to a list: 5 -> 3 -> 11 -> None]
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 (mutates our list)
  - Adding it to the end just entails setting the _rest attribute of the last element to be a new LinkedList with the given value. The following implementation is recursive.

```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)
```
Useful List Method: **prepend**

- **prepend**(self, val)
  - We may also want to add elements to the beginning of our list (this will also 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
# prepend allows us to add an element to the beginning of our list.
# like append, it will mutate the LinkedList instance it is called on
# LinkedLists are really fast at doing prepend operations -- you can
# see that there's no for loop required, just a few variable re-assignments!
def prepend(self, val):
    oldVal = self._value
    oldRest = self._rest
    self._value = val
    self._rest = LinkedList(oldVal, oldRest)
```

```
self 5 --> 3 --> 11 --> None
```
Useful List Method: `prepend`

- `prepend(self, val)`
  - We may also want to add elements to the beginning of our list (this will also 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
# prepend allows us to add an element to the beginning of our list.
# like append, it will mutate the LinkedList instance it is called on
# LinkedLists are really fast at doing prepend operations -- you can
# see that there's no for loop required, just a few variable re-assignments!
def prepend(self, val):
    oldVal = self._value
    oldRest = self._rest
    self._value = val
    self._rest = LinkedList(oldVal, oldRest)
```

```
val 5 3 11 None
```
Useful List Method: **insert**

- **insert**(self, val, index)
  - Finally, we may want to allow for list insertions at any point specified by some valid index.
  - Basic idea:
    - If the specified index is 0, we can just use the prepend method.
    - 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, we walk to the appropriate index in the list, and perform the insertion

```python
# inserts need a bit of iteration, but only until the index where # we'd like to insert the new element. once we reach that spot -- the # insertion operation itself is easy

def insert(self, val, index):
    if index == 0:
        self.prepend(val)
    else:
        currList = self
        while index > 1:
            index -= 1
            currList = currList._rest
        currList._rest = LinkedList(val, currList._rest)
```
Useful List Method: \texttt{insert}

- \texttt{insertRec(self, val, index)}
  - If the specified index is 0, we can just use the prepend method.
  - Otherwise, we walk to the appropriate index in the list, and perform the insertion.
  - Here is the recursive version

```python
# here is a recursive version of insert
def insertRec(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 of the list, so just append to the end
    elif self._rest is None:
        self._rest = LinkedList(val)
    # else we recurse until index reaches 0
    else:
        self._rest.insertRec(val, index - 1)
```
Iterating Over Our List

• 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                                  Traceback (most recent call last)
<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]
     71
     72     # [] list index notation also calls __setitem__() method

TypeError: 'NoneType' object is not subscriptable
Iterating Over Our List

• Currently, we can only indirectly iterate over the list using a loop over a `range` object.

• We'd really like to iterate directly over the elements of the list (without using a range)

• Side note: given our LinkedList implementation, this loop is also inefficient! A call to `len()` iterates over the entire list. Each indexing call `newList[i]` also iterates over the list up to index `i` each time.

```python
newList = LinkedList(5)
newList.append(10)
newList.append(42)

for i in range(len(newList)):
    print(newList[i])
```

5
10
42
Making our List **Iterable**

- What do we need to directly iterate over our list?
  - We need to make our class **iterable**
  - We need to implement the special methods `__iter__` and `__next__`
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

  This is a way to “hide” the error
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 from the sequence 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)
Creating an Iterator

• To create an iterator for a class we need to implement two methods:
  • `__iter__()`: which is called to create 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 the *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 it's
  called
Creating an Iterator for LinkedList

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

• current keeps track of where we are in the iterator

```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)
Creating an Iterator for LinkedList

• Note: We added a new attribute 'current' to __slots__
  - current keeps track of where we are in the iterator

```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 for LinkedList

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

  • _current keeps track of where we are in the iterator

```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)
```
Creating an Iterator for LinkedList

- Note: We added a new attribute `__current__` to `__slots__`
- `__current__` keeps track of where we are in the iterator

```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)
```
Using our New Iterable LinkedList

In [38]:

testList = LinkedList("w")
testList.append("o")
testList.append("o")
testList.append("t")
print("testList: ",testList)

    # for loops automatically use iterators
    for char in testList:
        print(char)

    testList:  [w, o, o, t]
    w
    o
    o
    t

In [39]:

    listIterator = iter(testList)

In [40]:

    print(next(listIterator))
    print(next(listIterator))
    print(next(listIterator))
    print(next(listIterator))

    w
    o
    o
    t