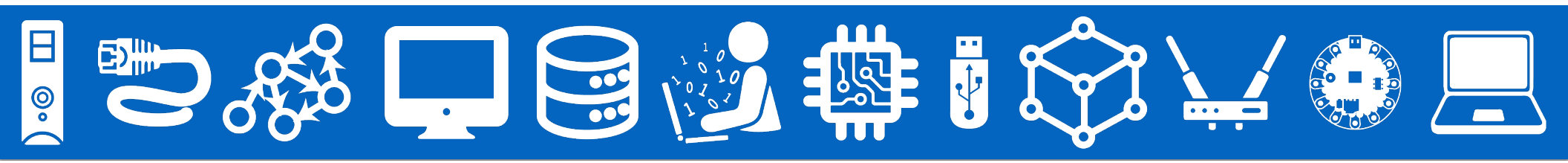
CS134:
Iterators



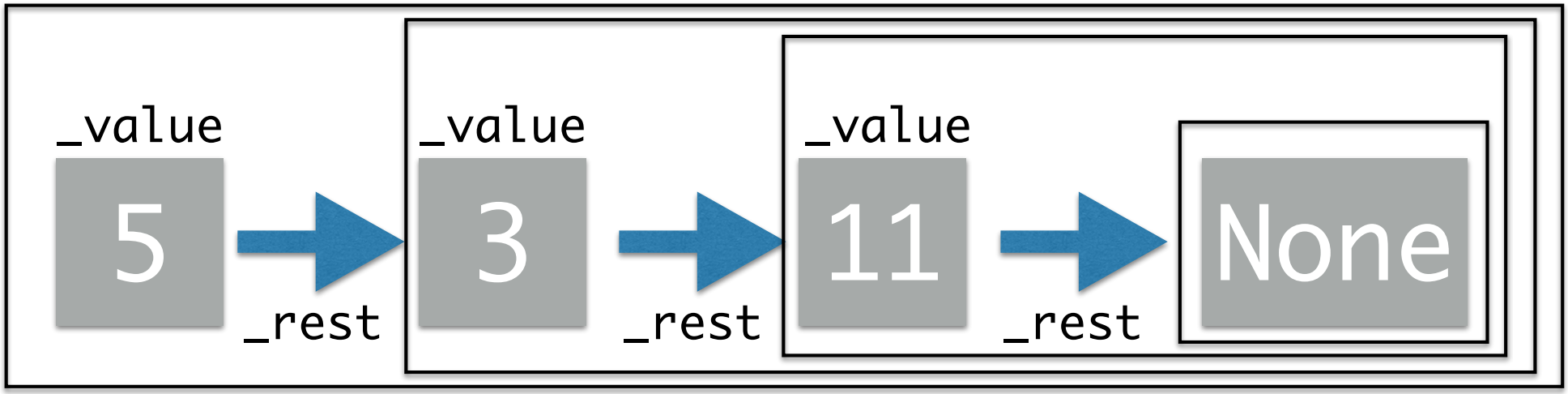
Announcements & Logistics

- **Lab 8** feedback coming soon! (Sorry!)
- **Lab 9 Boggle**
  - **Parts 3 (BoggleGame)** due Nov 30/Dec 1
- **Attendance in lab is optional next week**

Do You Have Any Questions?

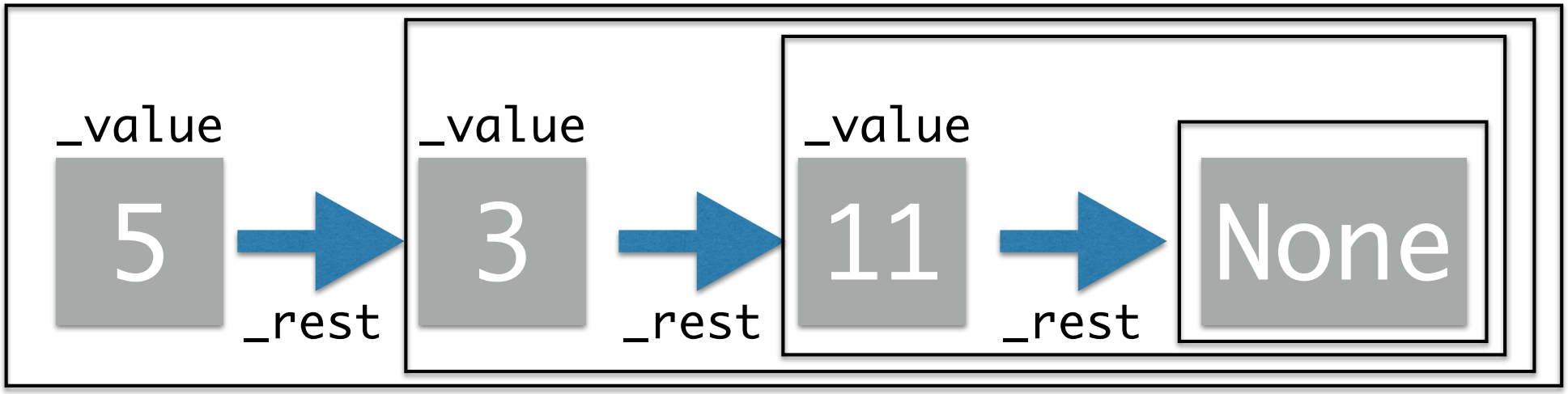
Last Time

- 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__` (+)



Today's Plan

- Wrap up our linked list class:
  - `__getitem__, __setitem__` (`[ ]` brackets to get/set value at index)
  - 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
  - Implement more special methods: `__iter__` and `__next__`



[] Operator: `__getitem__, __setitem__`

- `__getitem__(self, index)` and `__setitem__(self, index, val)`
  - In lists, we can get or set an item at a specific index using `[]`
    - get: `val = mylist[1]`
      set: `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__`

- We can **get** the item at a specific index using the [] operator (e.g., `val = mylist[2]`)

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

This is the same as `self._rest.__getitem__(index-1)`

```
>>> myList = LinkedList(5, LinkedList(3, LinkedList(11)))
>>> print(myList[2])
11
```

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

```
__getitem__(1)
    ...
    return LinkedList(11)[0]
```

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

11

[] Operator: `__getitem__, __setitem__`

- We can also **set** the item at a specific index using the [] operator (e.g., `mylist[2] = newVal`)

```
# [] 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__
        # this is the same as self._rest.__setitem__(index - 1, val)
        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)`
  - To support the == operator in our `LinkedList` class, we need to implement `__eq__`

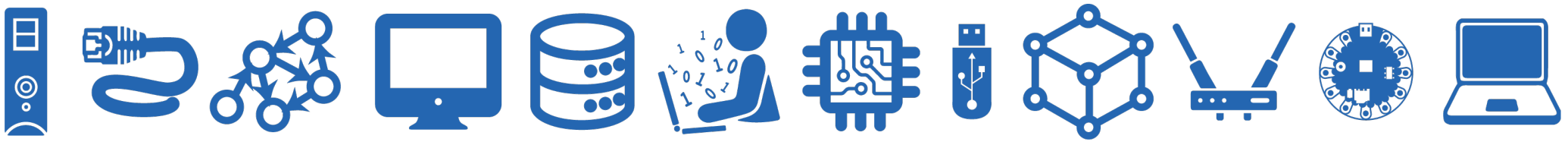
```
# == operator calls __eq__() method
# if we want to test two LinkedLists for equality, we test
# if all items are the same
# other is another LinkedList
def __eq__(self, other):
    # If both lists contain 0 or 1 item(s)
    if self._rest is None and other.getRest() is None:
        return self._value == other.getValue()

    # 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, so return false
    else:
        return False
```

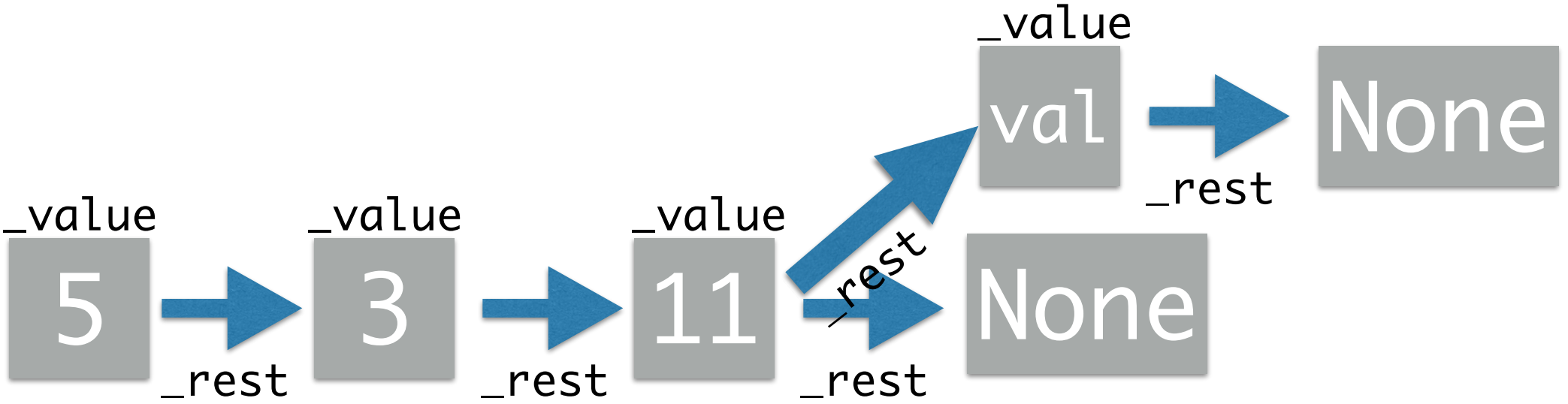
Useful list methods:

`.append()`, `.prepend()`, `.insert()`



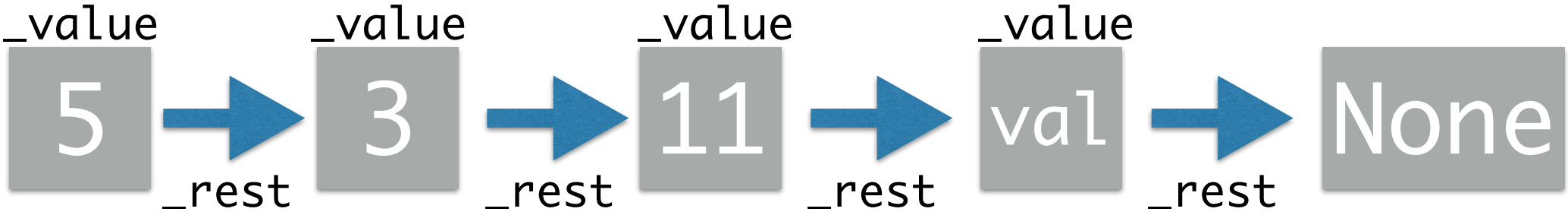
Useful List Method: `append`

- `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**

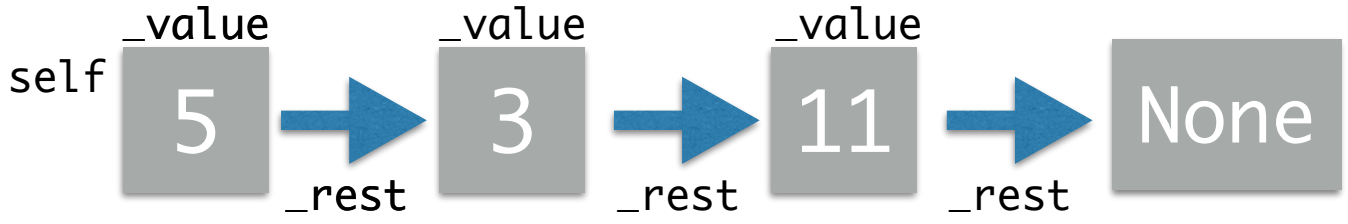
- `append(self, val)`
  - When using lists, we can add an element to the end of an existing list by calling **append** (note that **append** mutates our list)
  - This entails setting the `_rest` attribute of the last element to be a ***new*** LinkedList with the given value.

```
# append is not a special method, but it is a method
# that we know and love from the Python list class.
def append(self, val):
    # if this is the last 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 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.

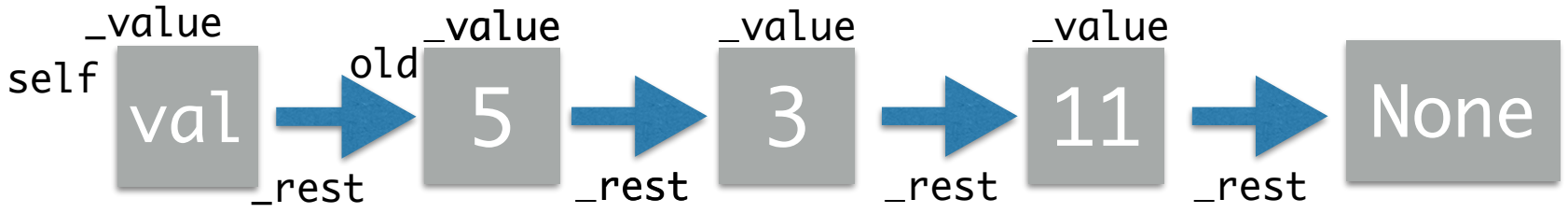
```
# prepend allows us to add an element to the beginning of our list.
# like append, it will mutate the LinkedList instance it is called on.
# LinkedLists are really fast at doing prepend operations!
# No recursion required, just a few variable re-assignments!
def prepend(self, val):
    oldVal = self._value
    oldRest = self._rest
    self._value = val
    self._rest = LinkedList(oldVal, oldRest)
```



Useful List Method: `prepend`

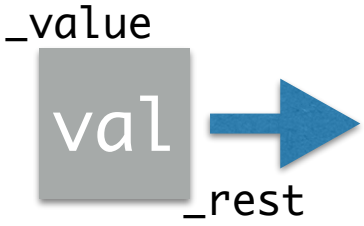
- `prepend(self, val)`
  - We may also want to add elements to the beginning of our list (this will mutate our list, similar to **append**)
  - The **prepend** operation is really efficient, we don't need to walk through the list at all — just do some variable reassignments.

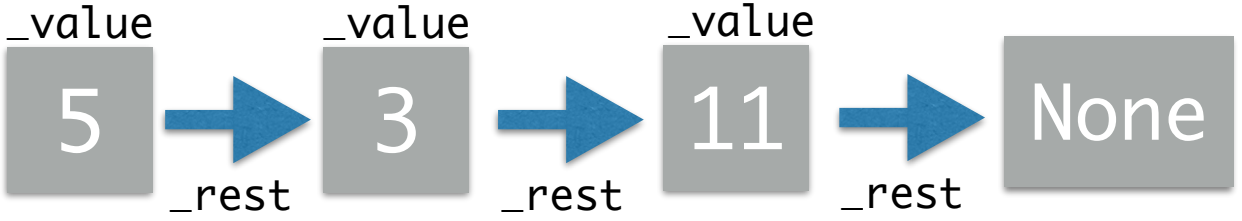
```
# prepend allows us to add an element to the beginning of our list.
# like append, it will mutate the LinkedList instance it is called on.
# LinkedLists are really fast at doing prepend operations!
# No recursion required, just a few variable re-assignments!
def prepend(self, val):
    oldVal = self._value
    oldRest = self._rest
    self._value = val
    self._rest = LinkedList(oldVal, oldRest)
```



Useful List Method: `insert`

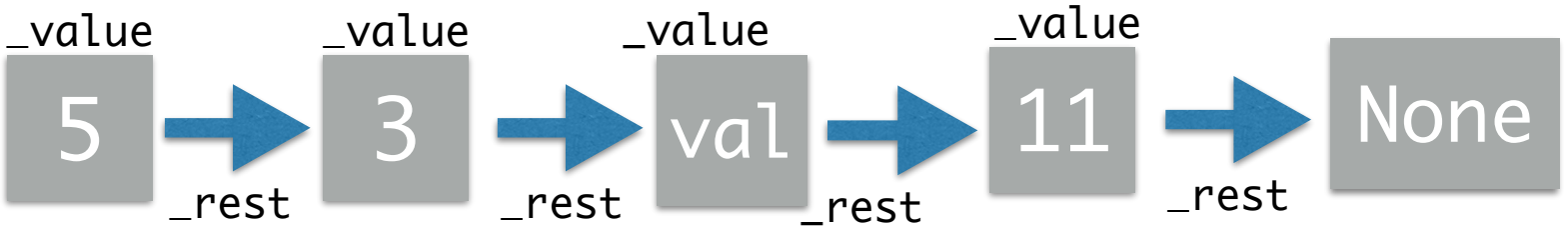
- `insert(self, val, index)`
  - Finally, we want to allow for insertions at a specific index.
  - Basic idea:
    - If the specified index is 0, we can just add to the beginning (easy!)
    - Otherwise, we walk to the appropriate index in the list, and reassign the `_rest` attribute at that location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

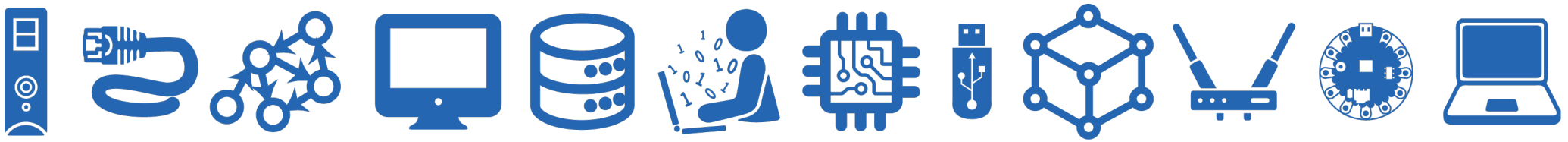


Useful List Method: `insert`

- `insert(self, val, index)`
  - If the specified index is 0, we can just use the `prepend` method.
  - Otherwise, we walk to the appropriate index in the list, and perform the insertion

```
# here is a recursive version of insert
def insert(self, val, index):
    # if index is 0, we found the item we need to return
    if index == 0:
        return self.prepend(val)
    else:
        # else we recurse until index reaches 0
        return self._rest.insert(val, index - 1)
```

Iterating Over Our List



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

```
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

TypeError: 'NoneType' object is not subscriptable
```

Iterating Over Our List

- Currently, we can only indirectly **iterate** over our LinkedList using a loop and a `range` object.
- We'd really like to **iterate** directly over the elements of the list (without using a range)
- *An aside:* Given our LinkedList implementation, this loop is very inefficient! Each call `newList[i]` walks the list out to index `i` each time.

```
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 linked list?
    - We need to make our class **iterable**
    - We need to implement the special methods `__iter__` and `__next__`

- First, let's start with a few definitions

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** object directly in a for loop
  - When an **iterable** object 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
- Note: **iterable** is an adjective, **iterator** is a noun, **iterate** is a verb

Python's Built-in Iterable Types

- 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()` (it remembers where we are)
- 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**)

```
>>> charList = list("rain")
>>> print(charList)
['r', 'a', 'i', 'n']
>>> charIterator = iter(charList)
>>> next(charIterator)
'r'
>>> next(charIterator)
'a'
>>> next(charIterator)
'i'
>>> next(charIterator)
'n'
>>> next(charIterator)
Traceback:
  File "<stdin>", line 1
StopIteration
```

This means there are no elements left!

Creating an Iterator

- To create an **iterator** for our 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 the **next** value that should be returned)
- Thus, `__iter__()` should always "reset" the current state to the beginning of our list, and `__next__()` should update this state (i.e., move to the next element) each time its called
- Python for loops automatically (and implicitly) create an iterator and call `next()` until the `StopIteration` exception is reached (see leftover slides at the end for more info!)

Creating an Iterator for LinkedList

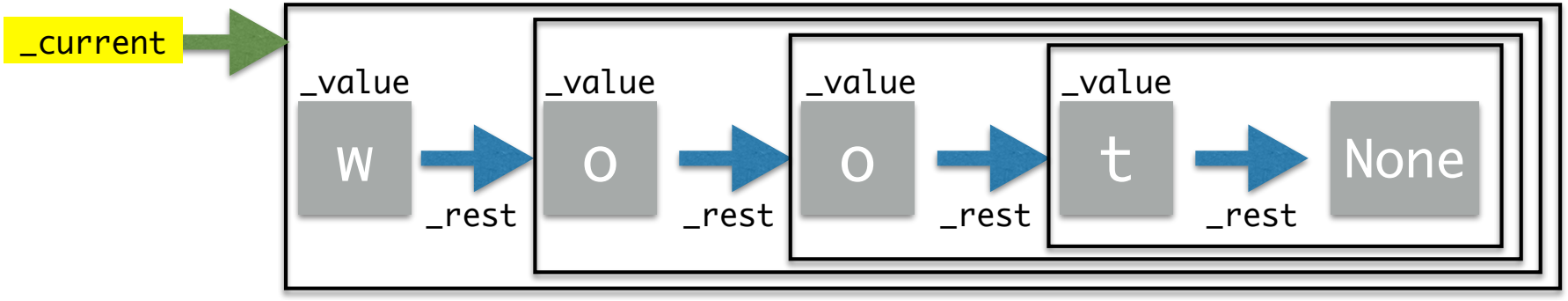
- First we add a new attribute '`_current`' to `__slots__`
  - `_current` keeps track of where we are in the iterator

```
def __iter__(self):
    # set current attribute to head (front of list)
    self._current = self
    return self

def __next__(self):
    if self._current is None:
        # we have reached the end of the list
        raise StopIteration
    else:
        # advance current to the next element in the list
        val = self._current._value
        self._current = self._current._rest
        return val
```

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

```
w
o
o
t
```



Creating an Iterator for LinkedList

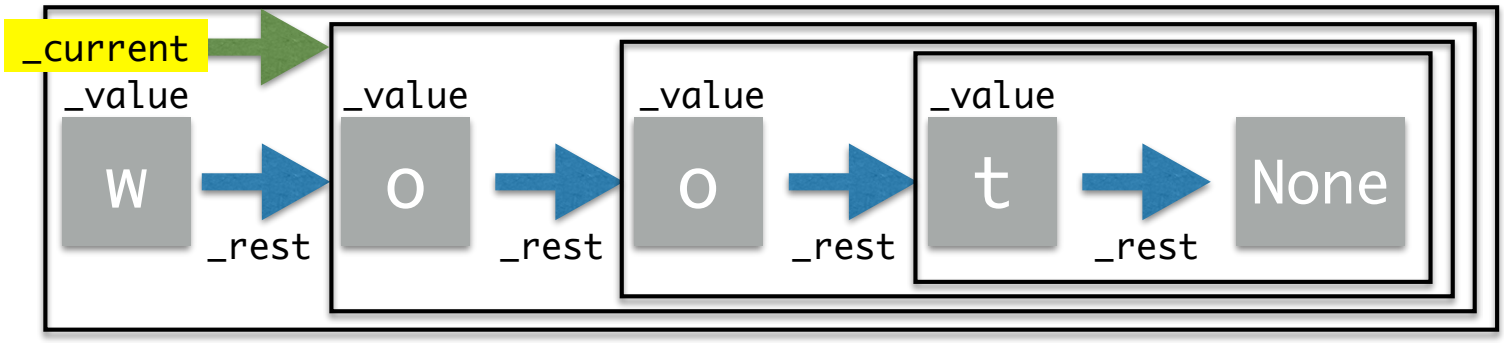
- First we add a new attribute '`_current`' to `__slots__`
  - `_current` keeps track of where we are in the iterator

```
def __iter__(self):
    # set current attribute to head (front of list)
    self._current = self
    return self

def __next__(self):
    if self._current is None:
        # we have reached the end of the list
        raise StopIteration
    else:
        # advance current to the next element in the list
        val = self._current._value
        self._current = self._current._rest
        return val
```

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

```
w
o
o
t
```



Creating an Iterator for LinkedList

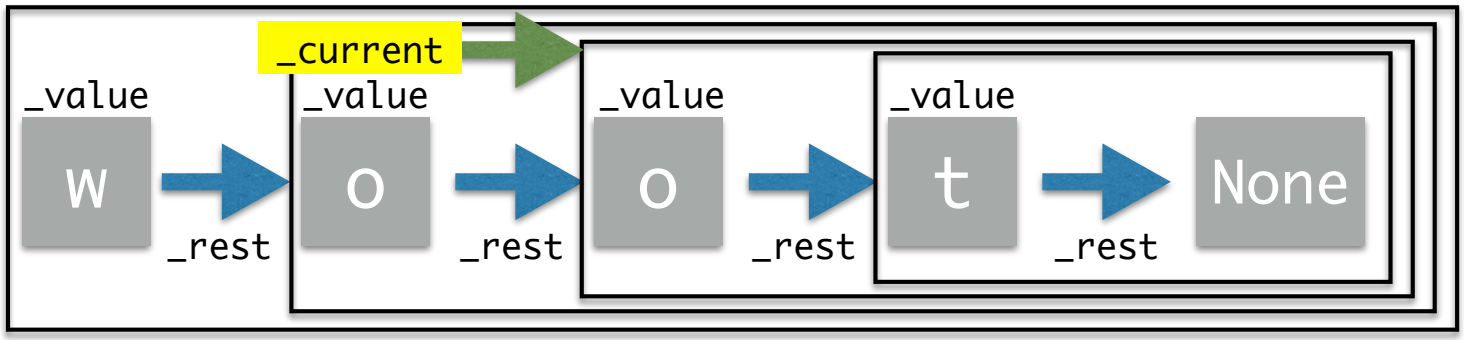
- First we add a new attribute '`_current`' to `__slots__`
  - `_current` keeps track of where we are in the iterator

```
def __iter__(self):
    # set current attribute to head (front of list)
    self._current = self
    return self

def __next__(self):
    if self._current is None:
        # we have reached the end of the list
        raise StopIteration
    else:
        # advance current to the next element in the list
        val = self._current._value
        self._current = self._current._rest
        return val
```

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

```
w
o
o
t
```



Creating an Iterator for LinkedList

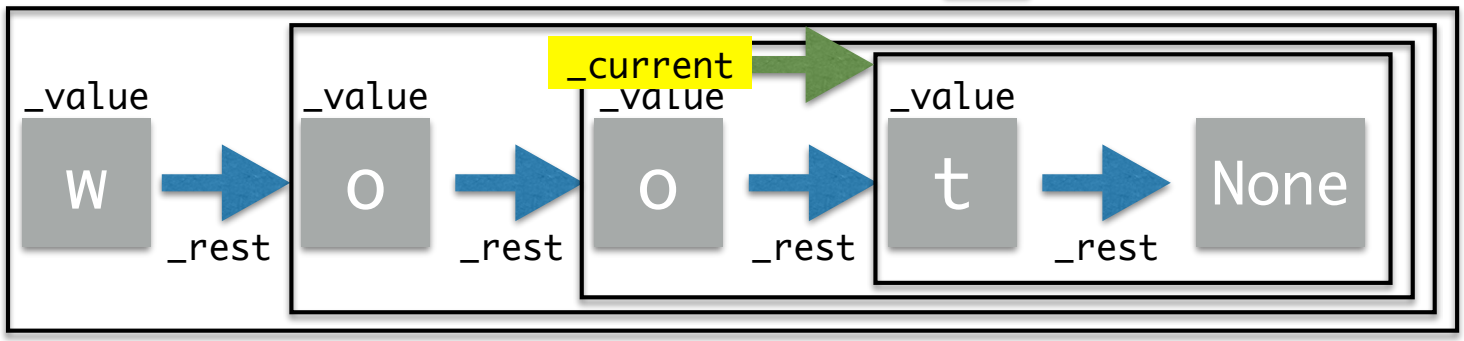
- First we add a new attribute '`_current`' to `__slots__`
  - `_current` keeps track of where we are in the iterator

```
def __iter__(self):
    # set current attribute to head (front of list)
    self._current = self
    return self

def __next__(self):
    if self._current is None:
        # we have reached the end of the list
        raise StopIteration
    else:
        # advance current to the next element in the list
        val = self._current._value
        self._current = self._current._rest
        return val
```

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

```
w
o
o
t
```



Creating an Iterator for LinkedList

- First we add a new attribute '`_current`' to `__slots__`
  - `_current` keeps track of where we are in the iterator

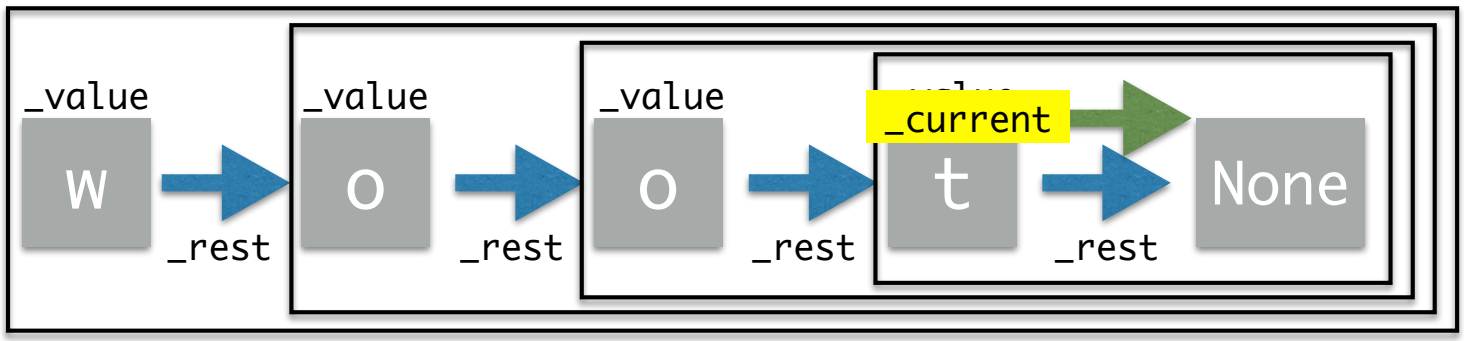
```
def __iter__(self):
    # set current attribute to head (front of list)
    self._current = self
    return self

def __next__(self):
    if self._current is None:
        # we have reached the end of the list
        raise StopIteration
    else:
        # advance current to the next element in the list
        val = self._current._value
        self._current = self._current._rest
        return val
```

This means there are no elements left!

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

```
w
o
o
t
```



Using our New Iterable LinkedList

```
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)
```

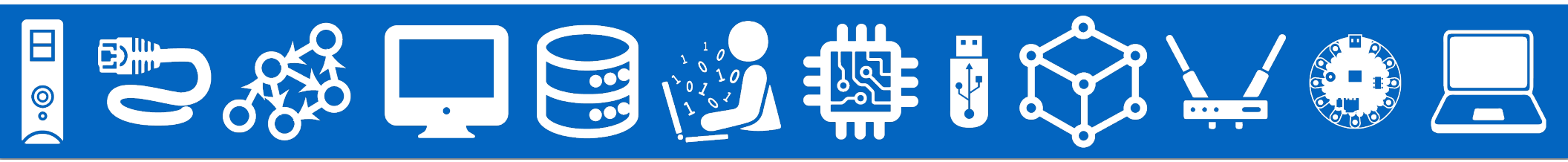
```
testList:  [w, o, o, t]
w
o
o
t
```

```
listIterator = iter(testList)
```

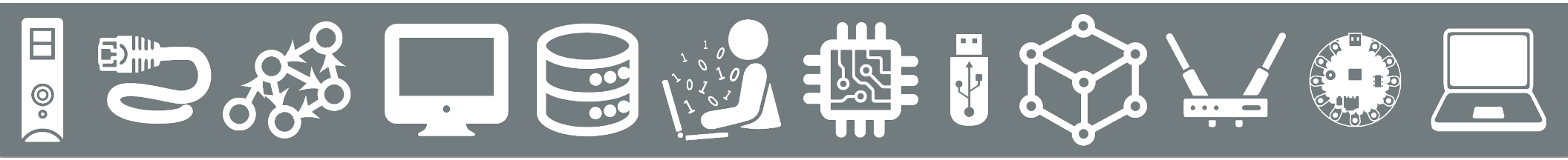
```
print(next(listIterator))
print(next(listIterator))
print(next(listIterator))
print(next(listIterator))
```

```
w
o
o
t
```

The end!



Leftover Slides



For loop: Behind the Scenes

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

```
# 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:

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

Call the `iter` method on object

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:

```
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!

What's Next in CS134

- Pre-midterm
    - Emphasis on basics of programming (conditionals, loops, etc)
    - Python's built-in data structures: lists, dictionaries, tuples, sets
    - Scripts, modules, and functions
- Post-midterm
    - Advanced programming topics
    - Recursive functions
    - Classes and OOP
    - Recursive data structures
    - **Brief introduction to searching/sorting and efficiency analysis**
    - **JAVA!**