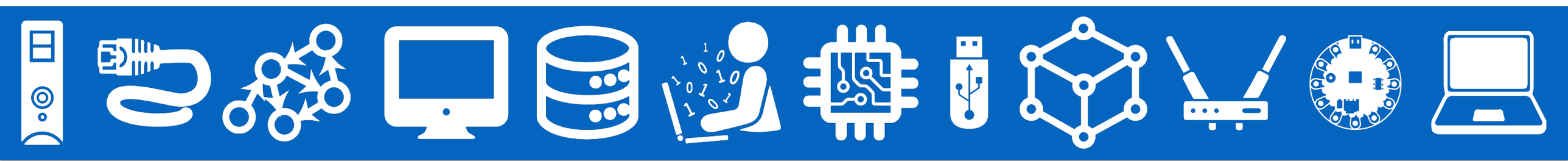
CS134 Lecture:
Special Methods & Linked Lists



Announcements & Logistics

- **HW 10** will be released today
- **Lab 9 Boggle**: two-week lab now in progress
  - **Part 1** due today/tomorrow
  - You can fix anything broken before turning in Part 2
  - **Part 2** handout will be posted Friday
  - Part 2 also has a **prelab!**
    - Asks you to draw out the Boggle game logic

Do You Have Any Questions?

Last Time

- Finished implementation of **Tic Tac Toe game**
    - (Fun!!) Application of object-oriented design and inheritance
    - (Fun!!) We can make our own data **types**!
    - A little exposure to software design
- Designed to help with the **Boggle lab**

Today's Plan

- Discuss special methods *more*, their purpose and how to call them
- Build a **recursive list class**
    - Our own implementation of `list`!
    - Preview of the fun world of design and implementation of data structures
- Learn how to implement several **special methods** which let us utilize built-in operators in Python for user-defined types

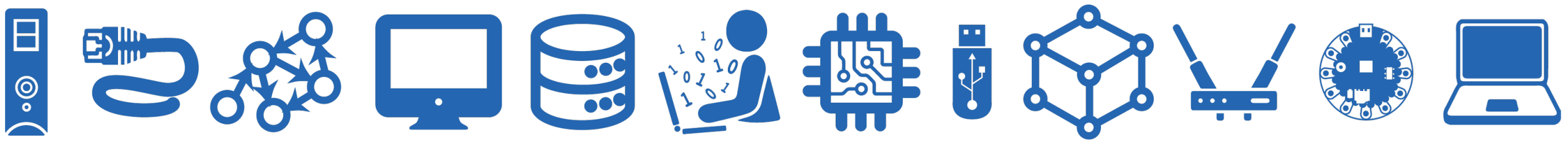
Python's Built-in list Class

- A class with methods (th
- `pydoc3 list`
- Let's implement our ow

```
Help on class list in module builtins:

class list(object)
 |  list(iterable=(), /)
 |
 |  Built-in mutable sequence.
 |
 |  If no argument is given, the constructor creates a new empty list.
 |  The argument must be an iterable if specified.
 |
 |  Methods defined here:
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __contains__(self, key, /)
 |      Return key in self.
 |
 |  __delitem__(self, key, /)
 |      Delete self[key].
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __ge__(self, value, /)
 |      Return self>=value.
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __getitem__(...)
 |      x.__getitem__(y) <==> x[y]
 |
 |  __gt__(self, value, /)
:
```

Notice the double underscores: these are special methods

Special Methods



Special Methods

- Start and end with __ (double underscore)
  - Called magic methods (or informally dunder methods)
- Often not called explicitly using dot notation and called by other means
- What special methods have we already used seen/used so far?
- **`__init__(self, val)`**
  - When is it called?
    - Automatically when we ***create*** an instance (object) of the class
    - Can also be invoked as `obj.__init__(val)` (where `obj` is an instance of the class)

Special Methods

- **`__str__(self)`**
  - When is it called?
    - When we ***print*** an instance of the class using `print(obj)`
    - Also called whenever we call `str` function on it: `str(obj)`
      - Can also be invoked as `obj.__str__()`
- **`__repr__(self)`**
  - Also returns a string but its format is very specific (can be used to recreate the object of the class)
  - Useful for debugging
  - Don't worry about any more specifics for this class

Special Methods for Operators

- We can use mathematical and logical operators such as `==/+` to compare/add two objects of a class by defining the corresponding special method
- Example of polymorphism (using a single method or operator for different uses)
  - `__eq__ (self, other):`
  - `__ne__ (self, other):`
  - `__lt__ (self, other):`
  - `__gt__ (self, other):`
  - `__add__(self, other) :`
  - `__sub__(self, other):`
  - `__mul__(self, other):`
- There are many others!

```
x == y
x != y
x < y
x > y
x + y
x - y
x * y
```

__add__: why we can concatenate sequences with **+** as well as add ints with **+**

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
    - All built-in collection data types we saw (string, list, range, tuple, set, dictionaries) have this special method implemented
    - This is why we are able to call `len` on them
- What is an example of a built-in type that we can't call `len` on?
  - `int, float, Bool, None`

Other Special Methods for Sequences

- What other sequence operators have we used in this class?
- They each have a special method that is called whenever they are used
  - **Get** an item at an index a sequence using `[ ]:` calls `__getitem__`
    - e.g., `word_lst[2]` implicitly calls `word_lst.__getitem__(2)`
  - **Set** an item at an index to another `val` using `[ ]`: calls `__setitem__`
    - e.g., `word_lst[0] = "hello"` implicitly calls `word_lst.__setitem__(0, "hello)`

`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)`
  - If we want the `in` operator to work for the objects of our class, we can do so by implementing the `__contains__` special method

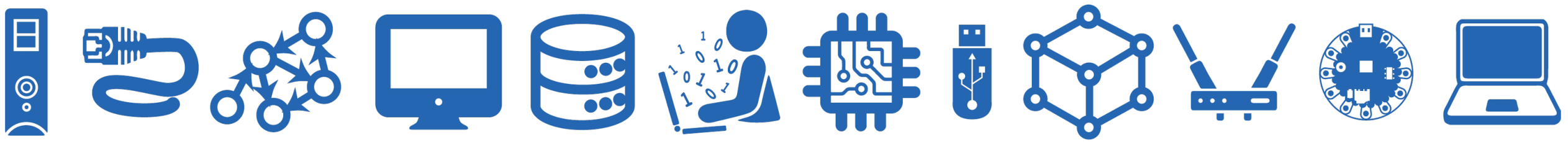
Python's Built-in list Class

- A class with methods (th
- `pydoc3 list`
- Let's implement our ow

```
Help on class list in module builtins:

class list(object)
 |  list(iterable=(), /)
 |
 |  Built-in mutable sequence.
 |
 |  If no argument is given, the constructor creates a new empty list.
 |  The argument must be an iterable if specified.
 |
 |  Methods defined here:
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __contains__(self, key, /)
 |      Return key in self.
 |
 |  __delitem__(self, key, /)
 |      Delete self[key].
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __ge__(self, value, /)
 |      Return self>=value.
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __getitem__(...)
 |      x.__getitem__(y) <==> x[y]
 |
 |  __gt__(self, value, /)
:
```

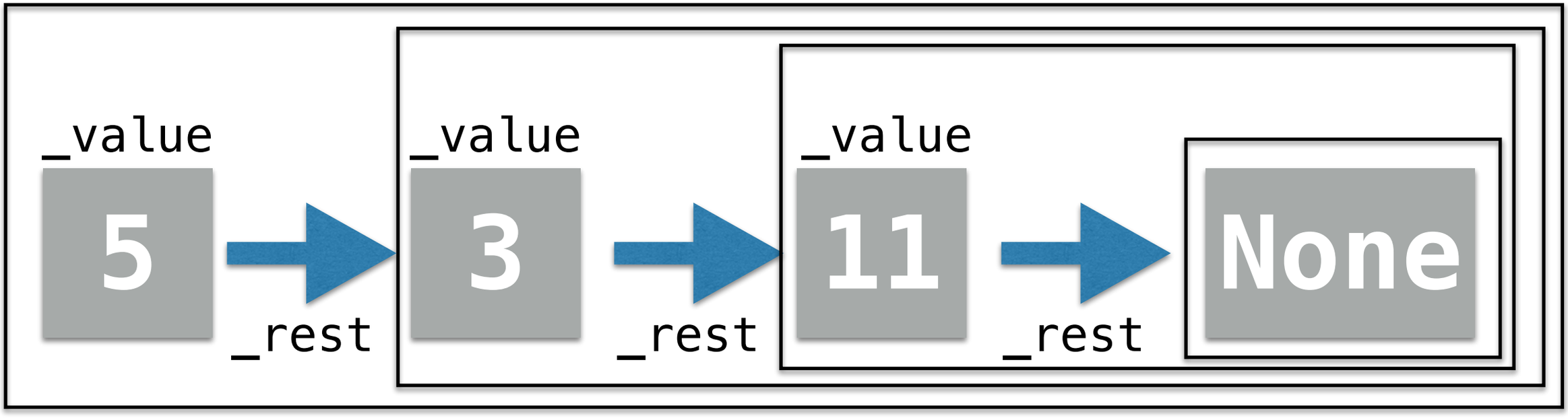
Notice the double underscores: these are special methods

Building Our Own List



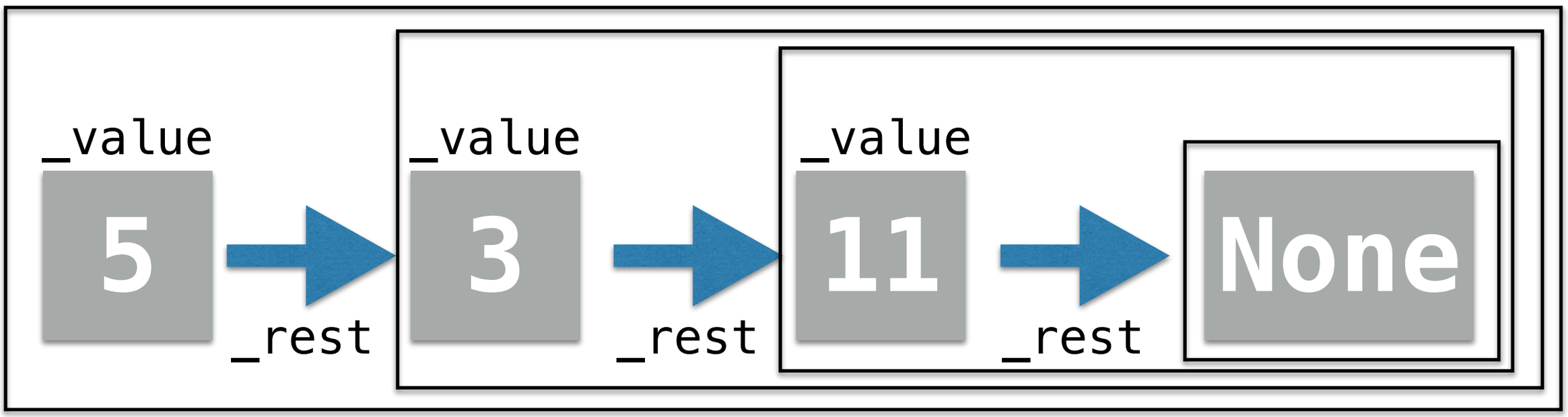
What exactly is a list?

- A container for a sequence of values
    - Recall that ***sequence*** implies an order
- Another way to think about this:
    - A nested *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`

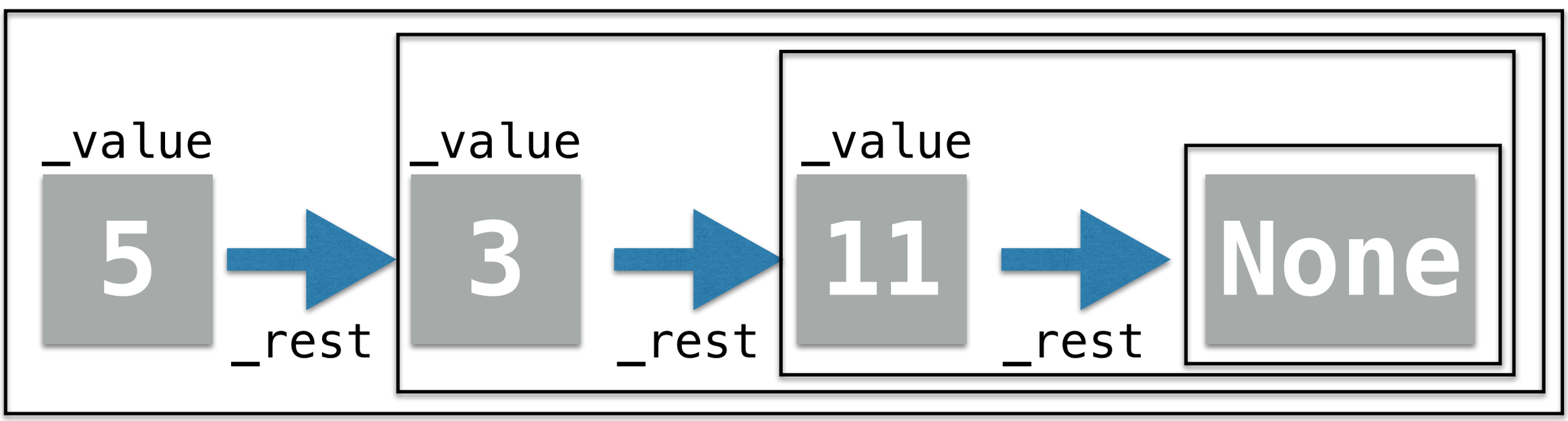
- Attributes:
  - `_value, _rest`
- **Recursive class**:
  - `_rest` points to another instance of the ***same class***
  - Any instance of a class that is created by using another instance of the class is a ***recursive class***



Initializing Our LinkedList

```
class LinkedList:
    """Implements our own recursive list data structure"""

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

    # getters/setters
    def get_rest(self):
        return self._rest

    def get_value(self):
        return self._value
```

`rest` is another instance of our LinkedList class



Special Methods (Review)

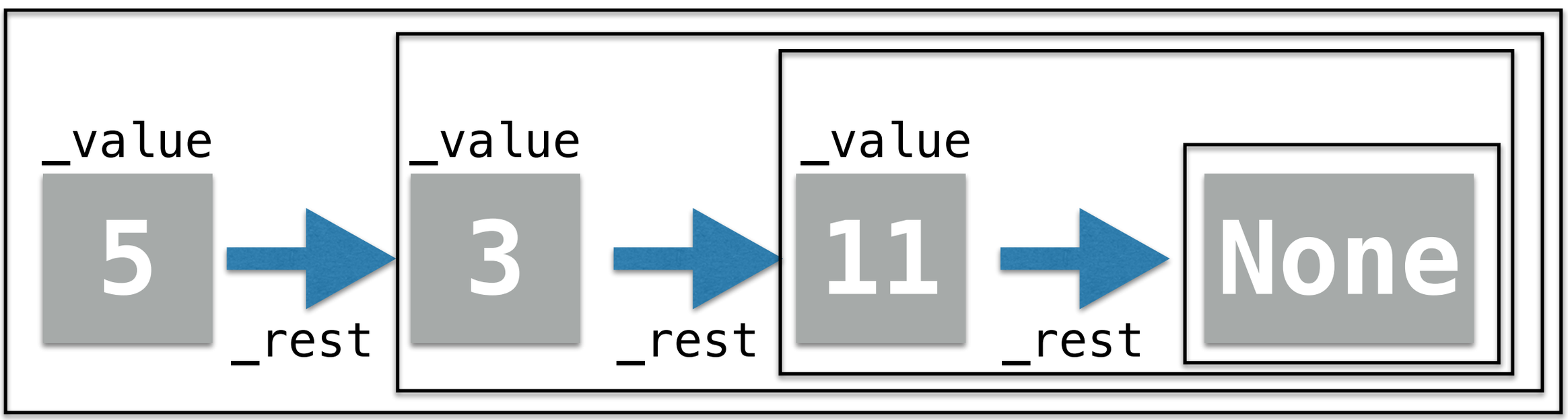
- **`__init__(self, val)`**
  - When is it called?
    - Automatically when we ***create*** an instance (object) of the class
    - Can also be invoked as `obj.__init__(val)` (where `obj` is an instance of the class)
- **`__str__(self)`**
  - When is it called?
    - When we ***print*** an instance of the class using `print(obj)`
    - Also called whenever we call `str` function on it: `str(obj)`
    - Can also be invoked 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)
```

This is recursion since `str` calls `__str__`. The base case is when `self._rest` is `None`

```
>>> my_lst = LinkedList(5, LinkedList(3, LinkedList(11)))
>>> print(my_lst) # testing __str__
5, 3, 11
```



Recursive Implementation: `__str__`

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

```
def __str_elements(self):
    # helper function for __str__()
    if self._rest is None:
        return str(self._value)
    else:
        return str(self._value) + ", " + self._rest.__str_elements()

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

```
>>> my_lst = LinkedList(5, LinkedList(3, LinkedList(11)))
>>> print(myList) # testing __str__
[5, 3, 11]
```

Looks more like Python list format

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

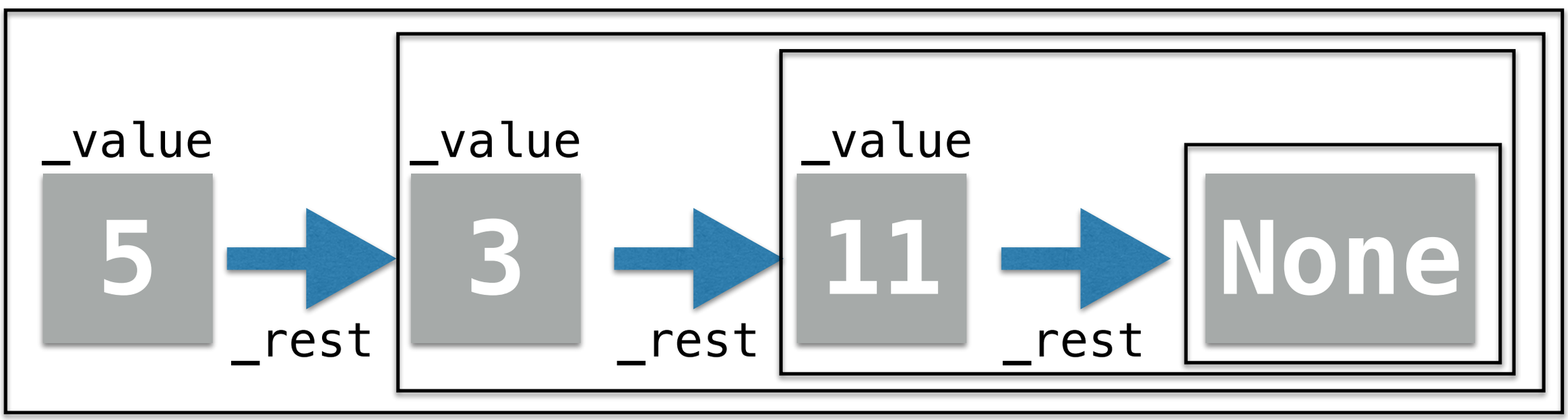
```
# repr() function calls __repr__() method
# return value should be a string that is a valid Python
# expression that can be used to recreate the LinkedList
def __repr__(self):
    return "LinkedList({}, {})".format(self._value, repr(self._rest))
```

```
>>> my_lst = LinkedList(5, LinkedList(3, LinkedList(11)))
>>> my_lst # testing __repr__
LinkedList(5, LinkedList(3, LinkedList(11, None)))
```

Notice we did not say print(myList) here

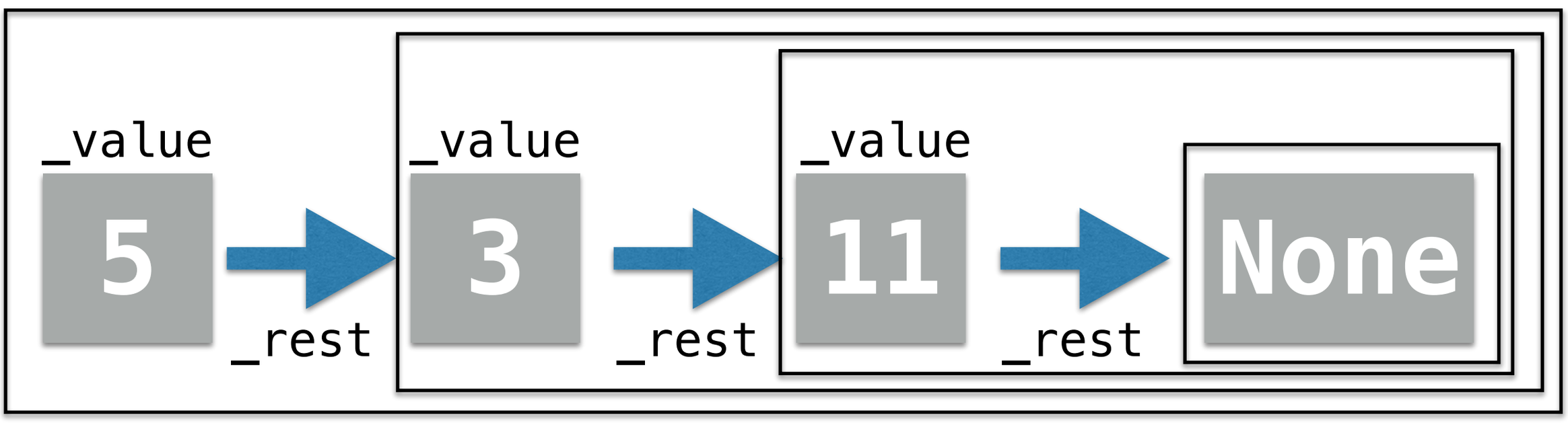
Special Method: `__len__`

- **`__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



Implementing Recursively

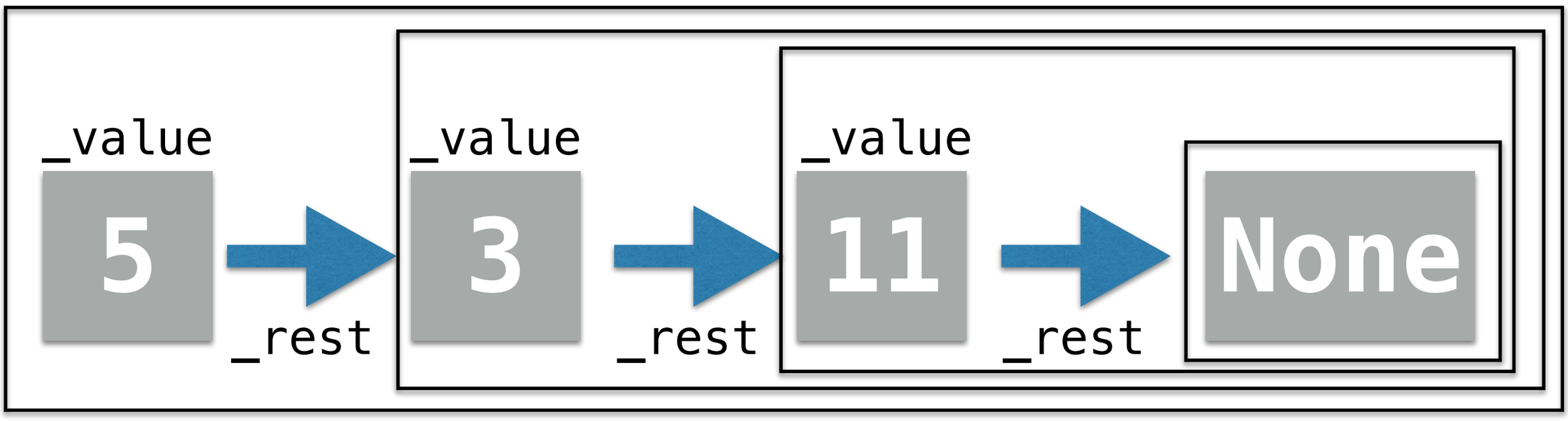
- As our `LinkedList` class is defined recursively, let's implement the `__len__` method recursively
    - Method will return an int (num of elements)
- What is the base case(s)?
- What about the recursive case?
    - Count self (so, +1), and then call `len()` on the rest of the list!



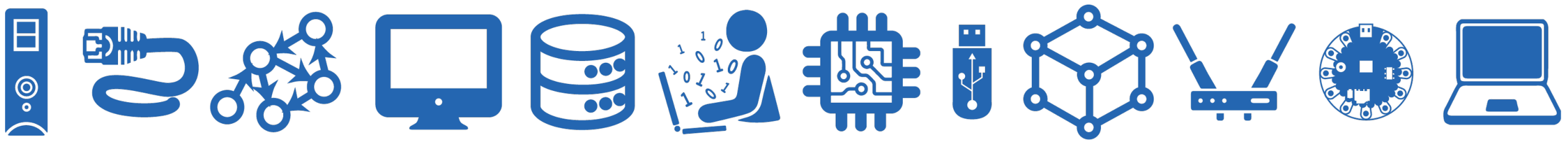
Recursive Implementation: `__len__`

```
# len() function calls __len__() method
def __len__(self):
    # base case: i'm the empty list
    if self._rest is None and self._value is None:
        return 0
   # base case: i'm the last item
   elif self._rest is None and self._value is not None:
        return 1
   #recursive case
   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.



Other Special Methods



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/Prepend** an item to the list: `append/prepend 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)`
- 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!

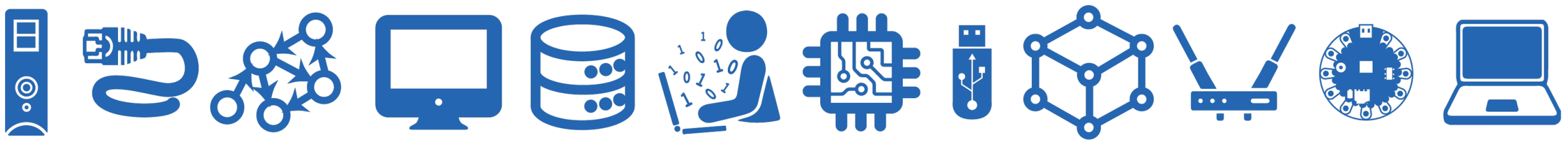
`in` Operator: `__contains__`

- **`__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)`
  - If we want the `in` operator to work for the objects of our class, we can do so by implementing the `__contains__` special method

```
    # 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
```

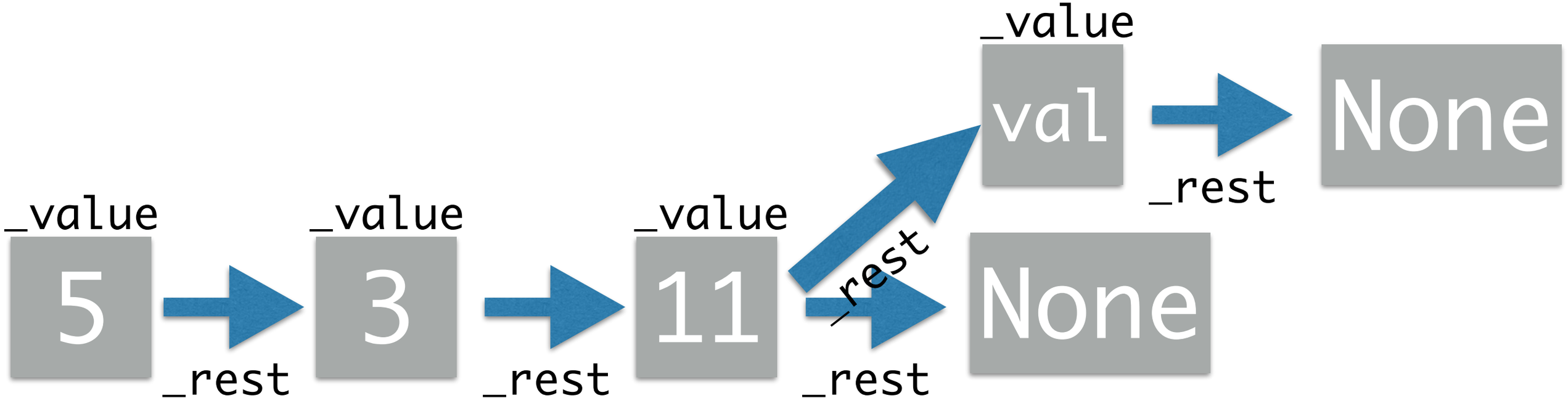
Useful list methods:

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



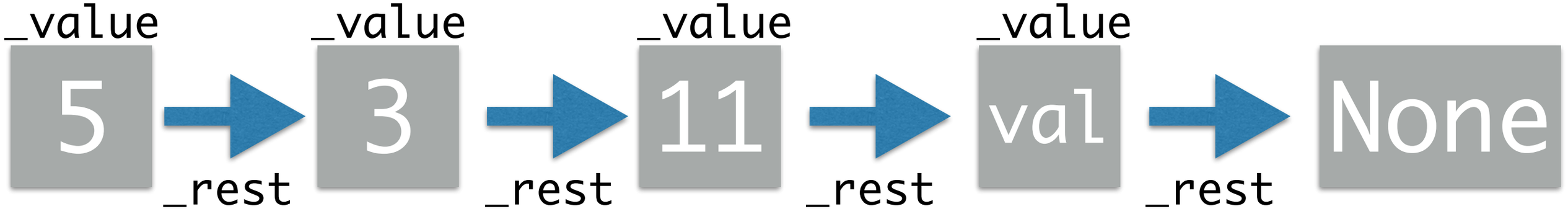
Useful List Method: **append**

- `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`**

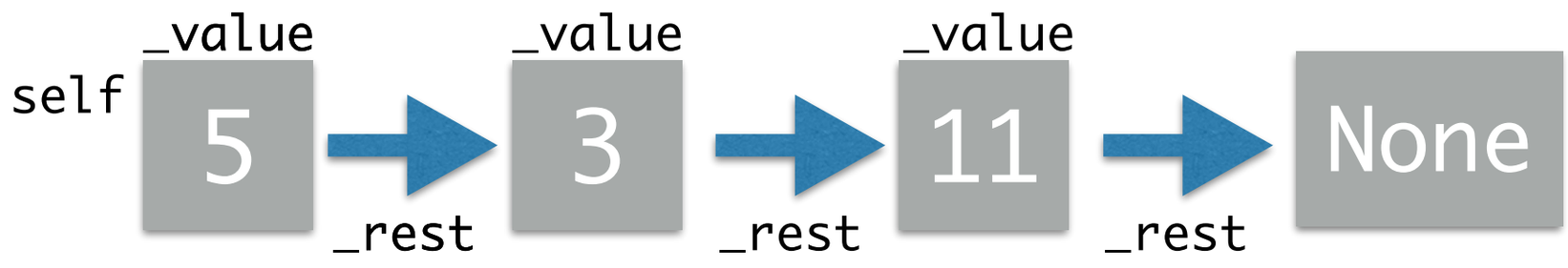
- `append(self, val)`
  - When using lists, we can add an element to the end of an existing list by calling **`append`** (note that **`append`** mutates our list)
  - This entails setting the **`_rest`** attribute of the last element to be a ***new*** LinkedList with the given value.

```
    def append(self, val):
        # if am at the end of the list
        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.

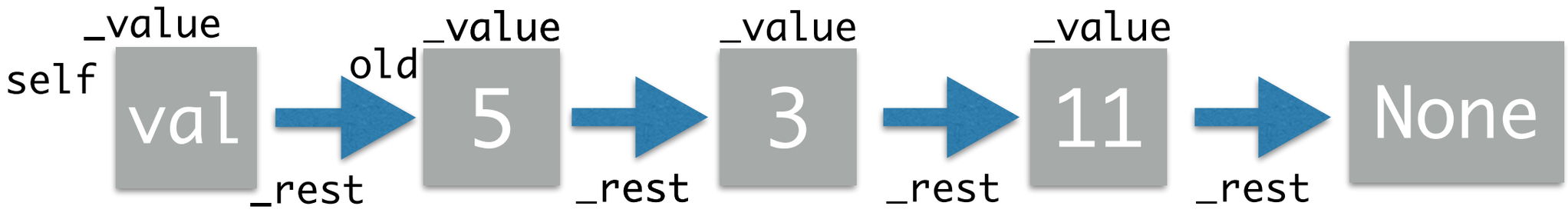
```
    def prepend(self, val):
        old_val = self._value
        old_rest = self._rest
        self._value = val
        self._rest = LinkedList(old_val, old_rest)
```



Useful List Method: `prepend`

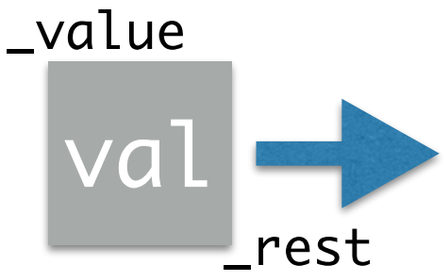
- `prepend(self, val)`
  - We may also want to add elements to the beginning of our list (this will mutate our list, similar to **`append`**)
  - The **`prepend`** operation is really efficient, we don't need to walk through the list at all — just do some variable reassignments.

```
    def prepend(self, val):
        old_val = self._value
        old_rest = self._rest
        self._value = val
        self._rest = LinkedList(old_val, old_rest)
```



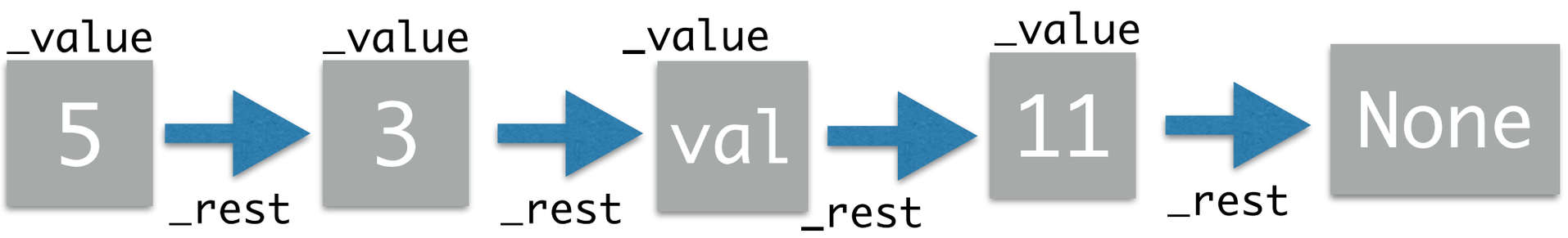
Useful List Method: `insert`

- `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, check to see if we're at end of the list
  - Otherwise, we walk to the appropriate index in the list, and perform the insertion

```
def insert(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, so just append
    elif self._rest is None:
        self._rest = LinkedList(val)
    # else we recurse until index reaches 0
    else:
        self._rest.insert(val, index - 1)
```

`[]` Operator: `__getitem__, __set_item__`

- **`__getitem__(self, index)`** and **`__setitem__(self, index, val)`**
  - With lists, we can *get* or *set* the item at a specific index by using the `[]` operator
    - get: `val = mylist[1]`
    - set: `mylist[2] = new_val`
  - 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!

`mylist[2]`

- implicitly: **`mylist.__getitem__(2)`**
  - When using lists, we can get the item at a specific index by using the `[]` operator (e.g., `val = mylist[2]`)
  - What might be the base case?

    We've reached the index, return the value!

  - What might be the recursive case?

    Cut one item off the front of our list, and subtract one from our index. Keep looking!

mylist[2]

- implicitly: **`mylist.__getitem__(2)`**
  - When using lists, we can get the item at a specific index by using the `[]` operator (e.g., `val = mylist[2]`)

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

base case

recursive case

```
my_lst = LinkedList(5, LinkedList(3, LinkedList(11)))
my_lst[2]
```

```
__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__, __set_item__`

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

```
# [] 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__`

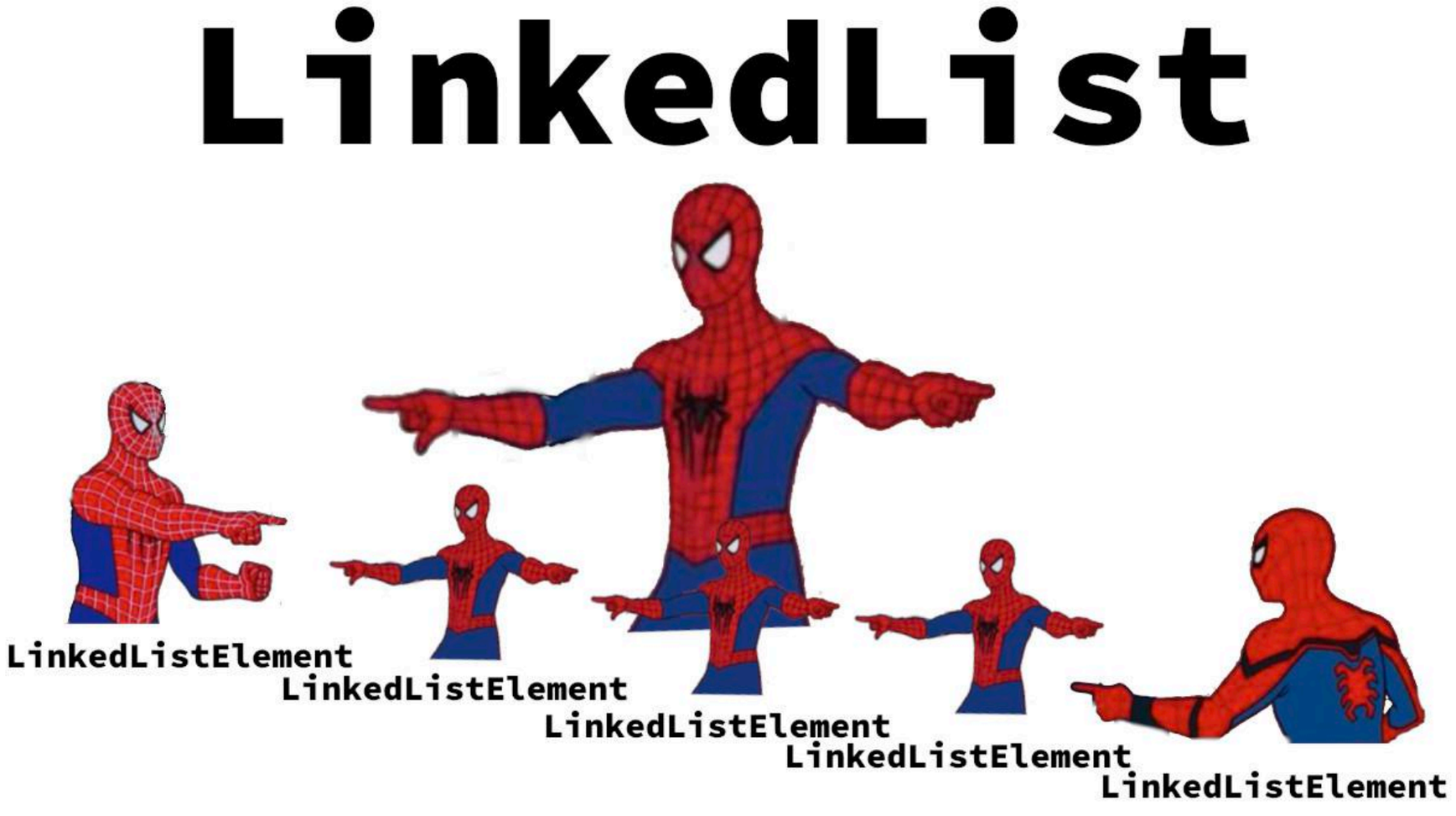
```
# == operator calls __eq__() method
# if we want to test two LinkedLists for equality, we test
# if all items are the same
# other is another LinkedList
def __eq__(self, other):
    # If both lists are empty
    if self._rest is None and other.get_rest() is None:
        return True

    # 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.get_rest() is not None :
        return self._value == other.get_value() and self._rest == other.get_rest()

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

Other Special Methods

- There are many other "special" methods in Python.
  - `__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`
  - There are others!

Looking Ahead

- In CS136 you'll see doubly linked lists! Overcomes some of the inefficiencies of singly linked lists



The end!