In this lab

- Linked Lists
  - Lists which store and manage items as a series of connected nodes
- Iterators
  - Specific objects which traverse through all elements contained in an *Iterable* container

List ADT

- A List ADT:
  - a *list* is a collection of items where each item holds a **relative position** with respect to the others. We can consider the list as having a first item, a second item, a third item, and so on. We can also refer to the **beginning** of the list (the first item) and the **end** of the list (the last item)
  - We can *add* and *remove* items from a list
  - We can *search* for the existence of an item in a list
  - We can make a distinction between *ordered* and *unordered* lists:

`54, 26, 93, 17, 77, 31`

Items are not stored in a sorted fashion

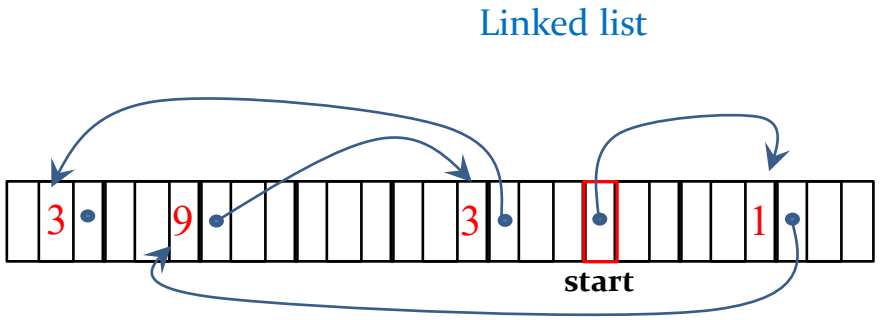
`17, 26, 31, 54, 77, 93`

Items are stored in a sorted fashion

  - We will start by considering only unordered lists

List ADT implementation using a Linked List

- The ADT can be implemented using a Linked List using the Node class



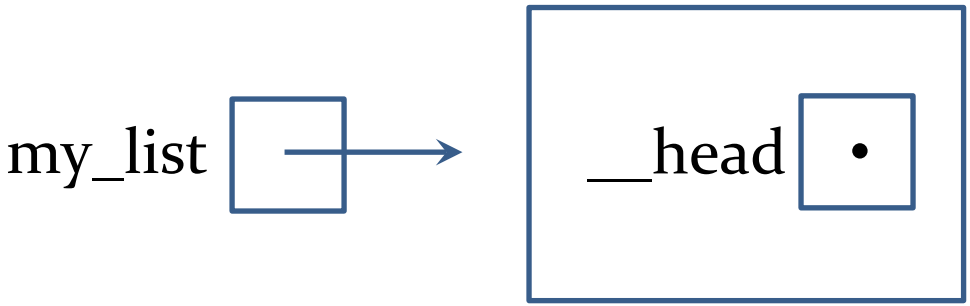
Q: What are the elements of this list?

- Add is O(1) (if we add at head)
- Remove is O(1)
  (Only if we have pointer to element – if not have to do search first)
- Search is O(n) ☹

```
class Node:
    def __init__(self, init_data):
    def get_data(self):
    def get_next(self):
    def set_data(self, new_data):
    def set_next(self, new_next):
    def __str__(self):

class LinkedList:

    def __init__(self):
          pass  ## Complete this

    def add(self, item): #add to the
          ## beginning of the list
          pass  ## Complete this

    def is_empty(self):
          pass  ## Complete this

    def size(self):
          pass  ## Complete this

    def search(self,item):
          pass  ## Complete this

    def remove(self, item):
          pass  ## Complete this
```

The LinkedList() constructor

```
def __init__(self):
        self.__head = None
```

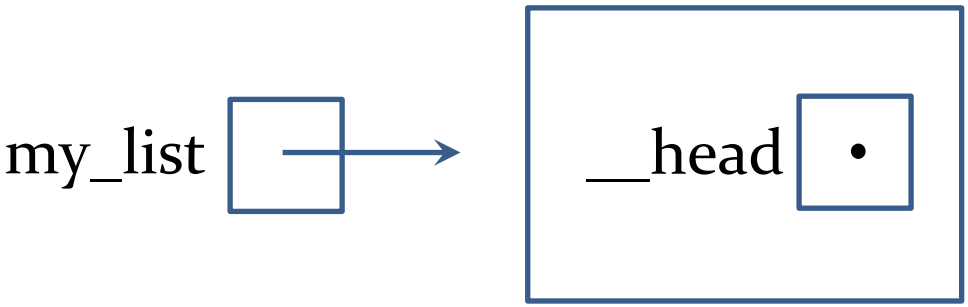


The "head" reference variable:

- references the list's first node (but for a new list is initially "None")
- always exists even when the list is empty

The is_empty () method

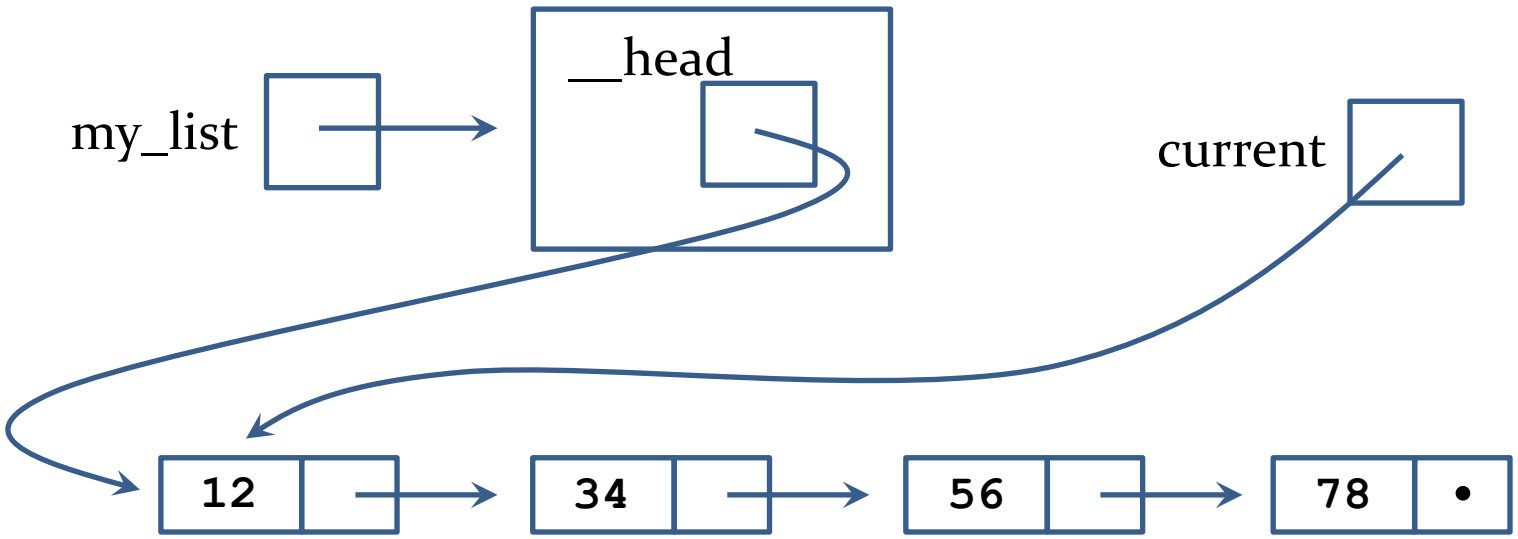
Can just check the value of head



If head is None, the list is empty!

The size() method

If all we have is a reference to the first node, we must traverse the entire chain




```
current = self.__head
```

Set a pointer "current" to be the same address as head and initialise the count to zero.
If element is not None increase count by one and move to next node
If element is None return count

The add() method

- To insert at the beginning of a linked list
  - Create a new Node and store the new data into it

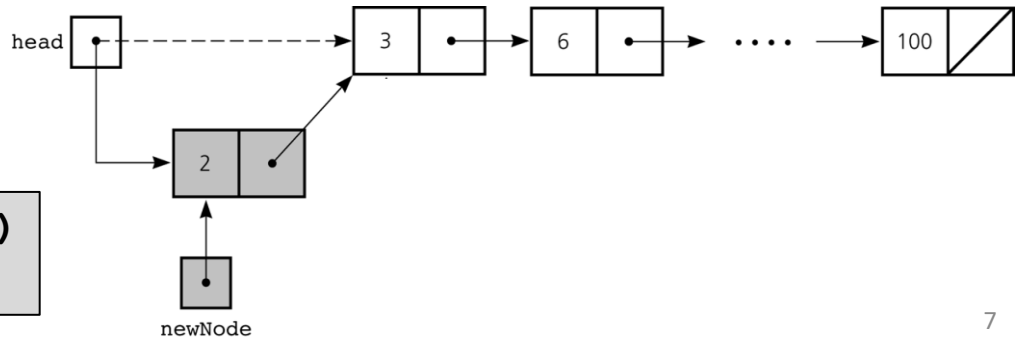
```
new_node = Node(item)
```

  - Connect the new node to the linked list by changing references
    - change the **next** reference of the new node to refer to the **old** first node of the list

```
new_node.set_next(self.__head)
```

    - modify the **head** of the list to refer to the **new** node

```
self.__head = new_node
```

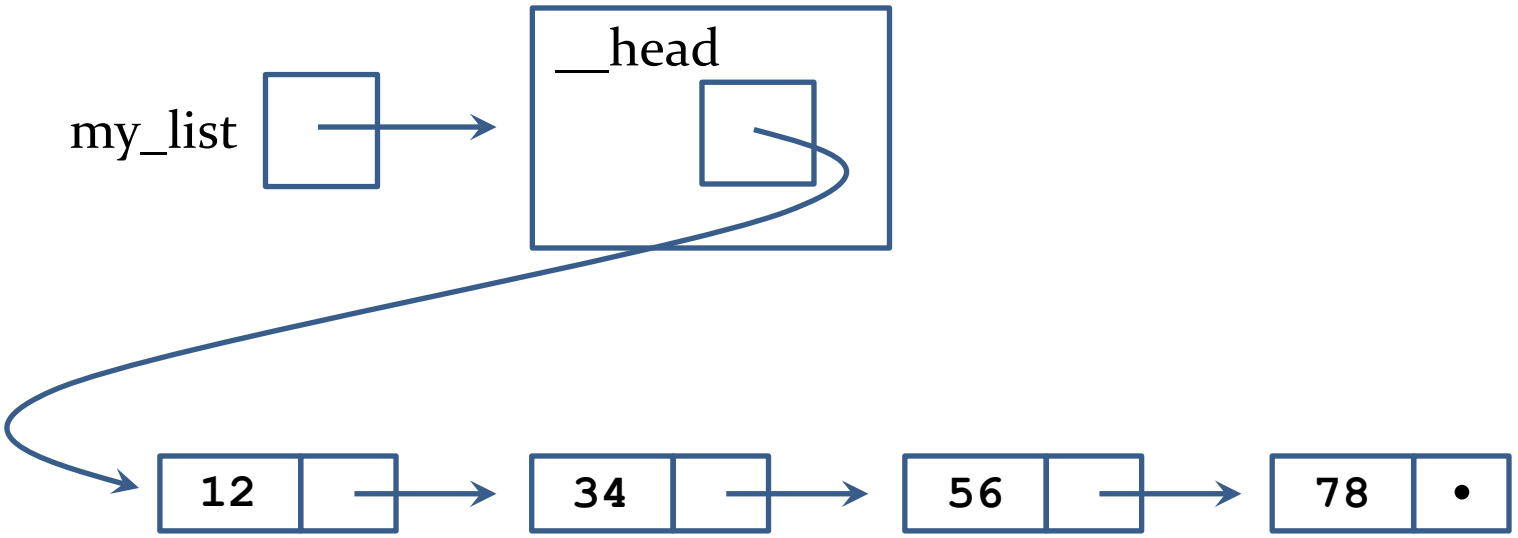


Or, combine steps together:

```
new_node = Node(item, self.__head)
self.__head = new_node
```

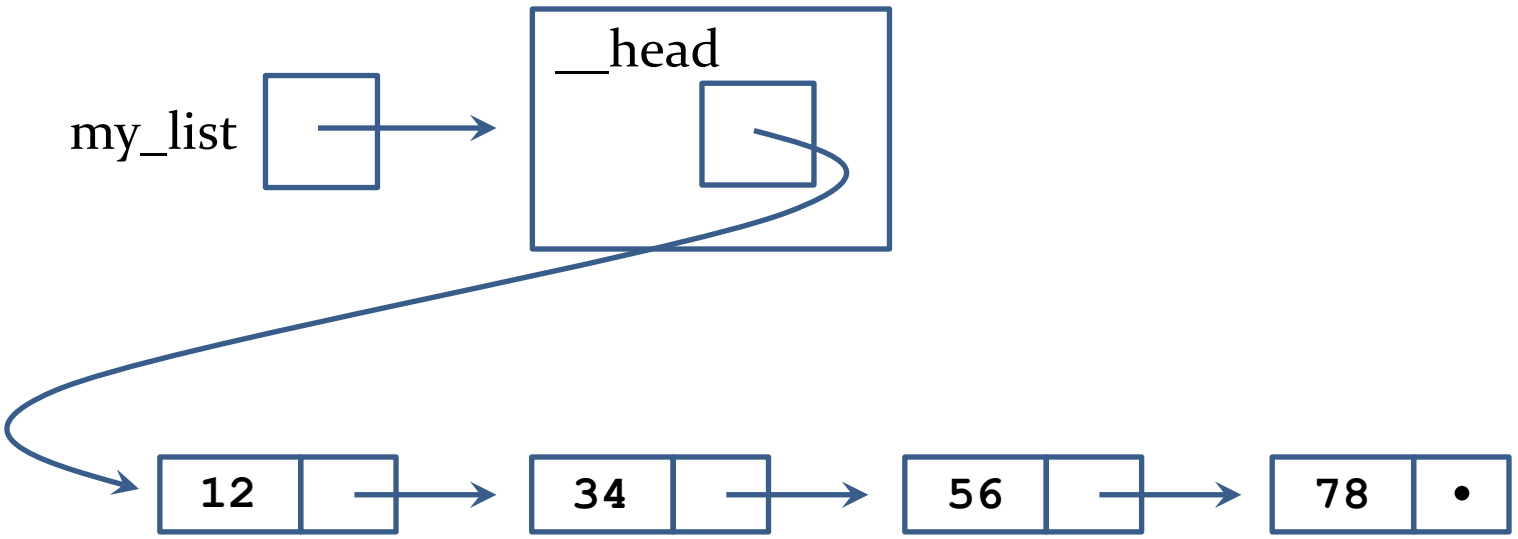
The __contains__ (i.e. search) method

- To search an item in a linked list:
  - set a pointer to be the same address as **head**
  - look to see if this is the data being searched for
  - move the pointer to the **next** node, and so on
  - loop stops when either the item is found or when the next pointer is None



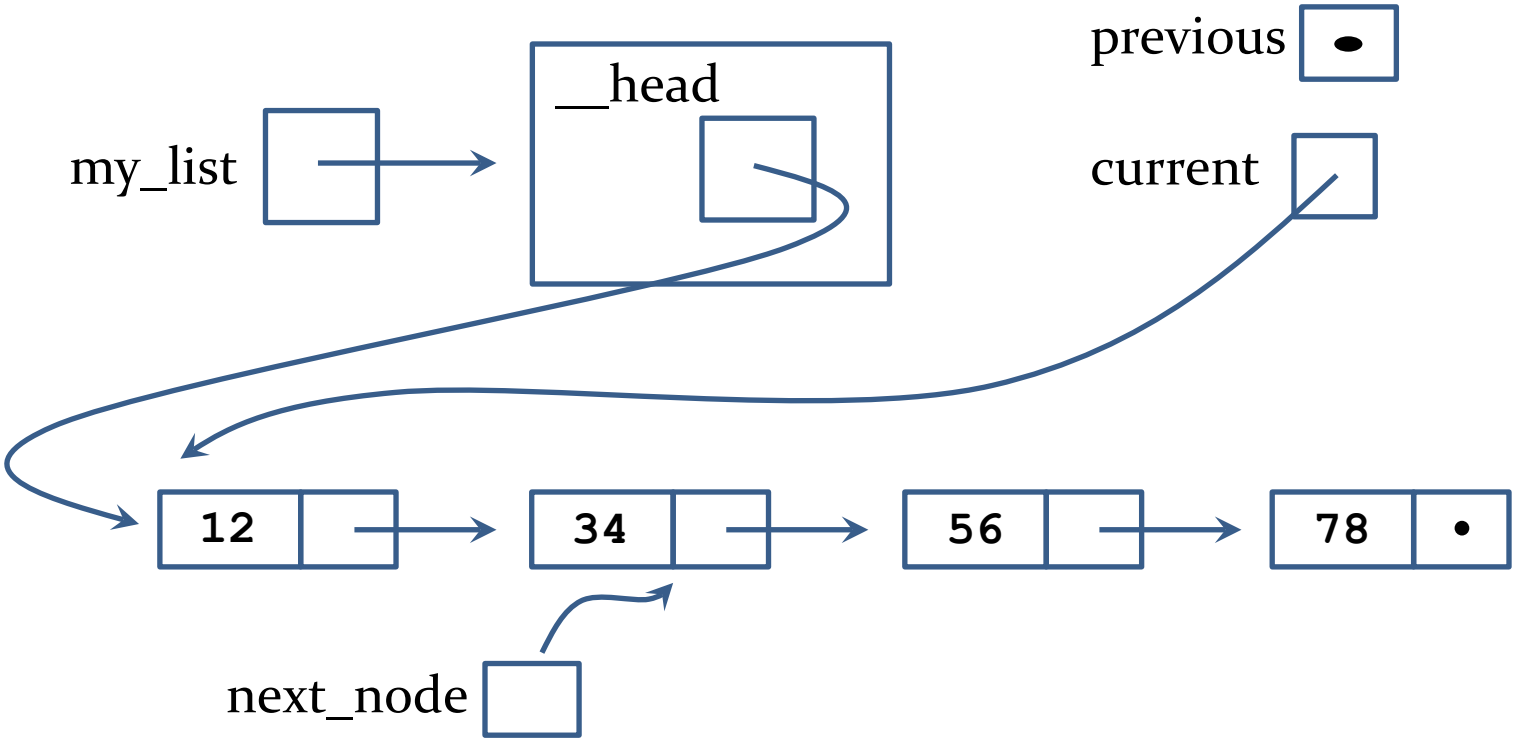
The __getitem__ method

- To get an item at a specific index from a linked list:
  - set a pointer to be the same address as **head**
  - move the pointer to the **next** node according to the given index
  - return the element



The reverse method

- To reverse the list by changing links between nodes:
  - set a pointer (previous) to None
  - set a pointer (current) to be the same address as **head**
  - Iterate trough the linked list. In loop, do following
    - set the next_node pointer to the **next** node of current
    - set the next node of current to previous
    - set previous to current
    - set current to next_node
  - set __head to previous



Iterators

- Iterators are objects which traverse through all the elements contained in an *iterable* container object.
- For example the following:

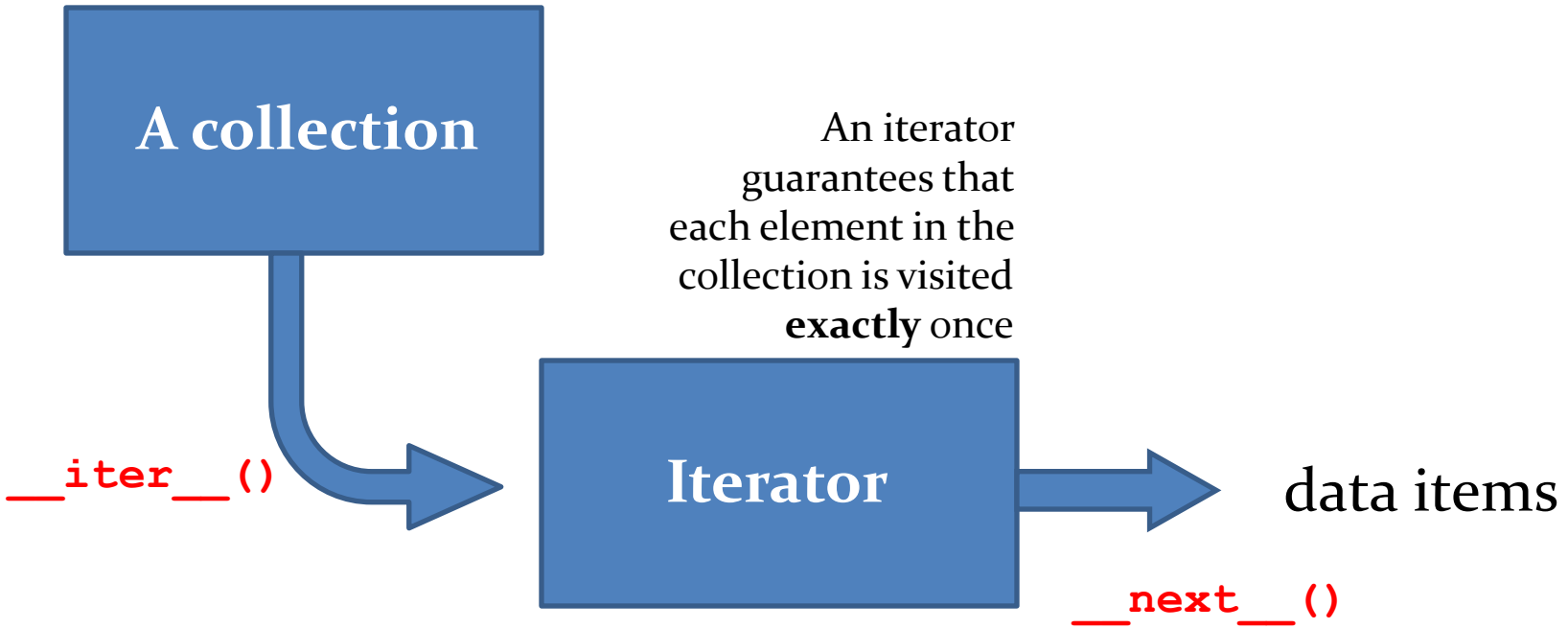
```
for value in my_linked_list:
…
```

Will only work if we have made the LinkedList class *iterable*

- Iterable objects implement the __iter__() method
- When we try to loop over values (like in a for loop) python calls the __iter__() method to get an Iterator object
- Iterators return values one at a time by calling the __next__() method
- When iterations have stopped, a StopIteration() exception is raised

Iterators

- When Python encounters a for loop, it automatically translates this into code that uses a special type of object called an *iterator*



- The iterator protocol requires two functions:
  - **__iter__()**: this returns the actual iterator object
  - **__next__()**: this method is called on the iterator object, and it returns the next item in the collection

Iterator for LinkedList

- We want an iterator to traverse a list with a for-loop, e.g.:

```
my_list = LinkedList()

my_list.add(1)
my_list.add(2)
my_list.add(3)
my_list.add(4)
my_list.add(5)

total = 0
for n in my_list:
    total += n

print('Sum = ', total)
```

```
class LinkedList:

    ....

    def __iter__(self):
        return
LinkedListIterator(self.__head)
```

```
class LinkedListIterator:

    def __init__(self, head):
    store head in self.__current

    def __next__(self):

    If we are at the end of the list:
            raise StopIteration
    Get the current item
    Advance the reference to the next item
    Return the current item
```