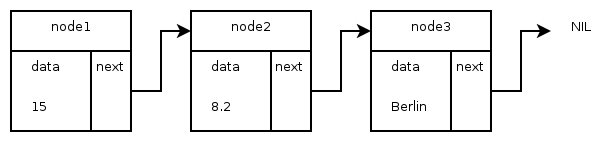
**Python Linked Lists**

By [Frank Hofmann](https://twitter.com/hofmannedv) • November 06, 2017 • [0 Comments](https://stackabuse.com/python-linked-lists/#disqus_thread)

A linked list is one of the most common data structures used in computer science. It is also one of the simplest ones too, and is as well as fundamental to higher level structures like stacks, circular buffers, and queues.

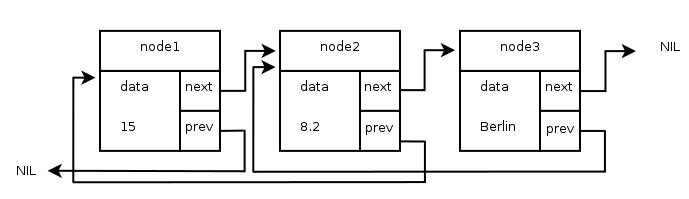
Generally speaking, a list is a collection of single data elements that are connected via references. C programmers know this as pointers. For example, a data element can consist of address data, geographical data, geometric data, routing information, or transaction details. Usually, each element of the [linked list](https://en.wikipedia.org/wiki/Linked_list) has the same data type that is specific to the list.

A single list element is called a node. The nodes are not like arrays which are stored sequentially in memory. Instead, it is likely to find them at different memory segments, which you can find by following the pointers from one node to the next. It is common to mark the end of the list with a NIL element, represented by the Python equivalent None.



*Figure 1: Single-linked list*

There exist two kinds of lists - single and [double-linked lists](https://en.wikipedia.org/wiki/Linked_list#Doubly_linked_list). A node in a single-linked list only points to the next element in the list, whereas a node in a double-linked list points to the previous node, too. The data structure occupies more space because you will need an additional variable to store the further reference.



*Figure 2: Double-linked list*

A single-linked list can be traversed from head to tail whereas traversing backwards is not as easy as that. In contrast, a double-linked list allows traversing the nodes in both directions at the same cost, no matter which node you start with. Also, adding and deleting of nodes as well as splitting single-linked lists is done in not more than two steps. In a double-linked list four pointers have to be changed.

The Python language does not contain a pre-defined datatype for linked lists. To cope with this situation we either have to create our own data type, or have to make use of additional Python modules that provide an implementation of such a data type.

In this article we'll go through the steps to create our own linked list data structure. First we create a corresponding data structure for the node. Second, you will learn how to implement and use both a single-linked list, and finally a double-linked list.

**Step 1: Node as a Data Structure**

To have a data structure we can work with, we define a node. A node is implemented as a class named ListNode. The class contains the definition to create an object instance, in this case, with two variables - data to keep the node value, and next to store the reference to the next node in the list. Furthermore, a node has the following methods and properties:

* \_\_init\_(): initialize the node with the data
* self.data: the value stored in the node
* self.next: the reference pointer to the next node
* has\_value(): compare a value with the node value

These methods ensure that we can initialize a node properly with our data (\_\_init\_\_()), and cover both the data extraction and storage (via the self.data property) as well getting the reference to the connected node (via the self.next property). The method has\_value() allows us to compare the node value with the value of a different node.

*Listing 1: The ListNode class*

class ListNode:

def \_\_init\_\_(self, data):

"constructor to initiate this object"

# store data

self.data = data

# store reference (next item)

self.next = None

return

def has\_value(self, value):

"method to compare the value with the node data"

if self.data == value:

return True

else:

return False

Creating a node is as simple as that, and instantiates an object of class ListNode:

*Listing 2: Instantiation of nodes*

node1 = ListNode(15)

node2 = ListNode(8.2)

node3 = ListNode("Berlin")

Having done that we have available three instances of the ListNode class. These instances represent three independent nodes that contain the values 15 (integer), 8.2 (float), and "Berlin" (string).

**Step 2: Creating a Class for a Single-Linked List**

As the second step we define a class named SingleLinkedList that covers the methods needed to manage our list nodes. It contains these methods:

* \_\_init\_\_(): initiate an object
* list\_length(): return the number of nodes
* output\_list(): outputs the node values
* add\_list\_item(): add a node at the end of the list
* unordered\_search(): search the list for the nodes with a specified value
* remove\_list\_item\_by\_id(): remove the node according to its id

We will go through each of these methods step by step.

The \_\_init\_\_() method defines two internal class variables named head and tail. They represent the beginning and the end nodes of the list. Initially, both head and tail have the value None as long as the list is empty.

*Listing 3: The SingleLinkedList class (part one)*

class SingleLinkedList:

def \_\_init\_\_(self):

"constructor to initiate this object"

self.head = None

self.tail = None

return

**Step 3: Adding Nodes**

Adding items to the list is done via add\_list\_item(). This method requires a node as an additional parameter. To make sure it is a proper node (an instance of class ListNode) the parameter is first verified using the built in Python function isinstance(). If successful, the node will be added at the end of the list. If item is not a ListNode, then one is created.

In case the list is (still) empty the new node becomes the head of the list. If a node is already in the list, then the value of tail is adjusted accordingly.

*Listing 4: The SingleLinkedList class (part two)*

def add\_list\_item(self, item):

"add an item at the end of the list"

if not isinstance(item, ListNode):

item = ListNode(item)

if self.head is None:

self.head = item

else:

self.tail.next = item

self.tail = item

return

The list\_length() method counts the nodes, and returns the length of the list. To get from one node to the next in the list the node property self.next comes into play, and returns the link to the next node. Counting the nodes is done in a while loop as long as we do not reach the end of the list, which is represented by a None link to the next node.

*Listing 5: The SingleLinkedList class (part three)*

def list\_length(self):

"returns the number of list items"

count = 0

current\_node = self.head

while current\_node is not None:

# increase counter by one

count = count + 1

# jump to the linked node

current\_node = current\_node.next

return count

The method output\_list() outputs the node values using the node property data. Again, to get from one node to the next the link is used that is provided via next property.

*Listing 6: The SingleLinkedList class (part four)*

def output\_list(self):

"outputs the list (the value of the node, actually)"

current\_node = self.head

while current\_node is not None:

print(current\_node.data)

# jump to the linked node

current\_node = current\_node.next

return

Based on the class SingleLinkedList we can create a proper list named track, and play with its methods as already described above in *Listings 3-6*. Therefore, we create four list nodes, evaluate them in a for loop and output the list content. *Listing 7* shows you how to program that, and *Listing 8* shows the output.

*Listing 7: Creation of nodes and list output*

# create four single nodes

node1 = ListNode(15)

node2 = ListNode(8.2)

item3 = "Berlin"

node4 = ListNode(15)

track = SingleLinkedList()

print("track length: %i" % track.list\_length())

for current\_item in [node1, node2, item3, node4]:

track.add\_list\_item(current\_item)

print("track length: %i" % track.list\_length())

track.output\_list()

The output is as follows, and shows how the list grows:

*Listing 8: Adding nodes to the list*

$ python3 simple-list.py

track length: 0

track length: 1

15

track length: 2

15

8.2

track length: 3

15

8.2

Berlin

track length: 4

15

8.2

Berlin

15

**Step 4: Searching the List**

Searching the entire list is done using the method unordered\_search(). It requires an additional parameter for the value to be searched. The head of the list is the starting point.

**Subscribe to our Newsletter**

Top of Form

Get occassional tutorials, guides, and reviews in your inbox. No spam ever. Unsubscribe at any time.

Bottom of Form

Top of Form

Bottom of Form

While searching we count the nodes. To indicate a match we use the corresponding node number. The method unordered\_search() returns a list of node numbers that represent the matches. As an example, both the first and fourth node contain the value 15. The search for 15 results in a list with two elements: [1, 4].

*Listing 9: The search method unordered\_search()*

def unordered\_search (self, value):

"search the linked list for the node that has this value"

# define current\_node

current\_node = self.head

# define position

node\_id = 1

# define list of results

results = []

while current\_node is not None:

if current\_node.has\_value(value):

results.append(node\_id)

# jump to the linked node

current\_node = current\_node.next

node\_id = node\_id + 1

return results

**Step 5: Removing an Item from the List**

Removing a node from the list requires adjusting just one reference - the one pointing to the node to be removed must now point to the next one. This reference is kept by the node to be removed, and must be replaced. In the background the Python garbage collector takes care of unreferenced objects, and tidies up.

The following method is named remove\_list\_item\_by\_id(). As a parameter it refers to the number of the node similar to the value returned by unordered\_search().

*Listing 10: Removing a node by node number*

def remove\_list\_item\_by\_id(self, item\_id):

"remove the list item with the item id"

current\_id = 1

current\_node = self.head

previous\_node = None

while current\_node is not None:

if current\_id == item\_id:

# if this is the first node (head)

if previous\_node is not None:

previous\_node.next = current\_node.next

else:

self.head = current\_node.next

# we don't have to look any further

return

# needed for the next iteration

previous\_node = current\_node

current\_node = current\_node.next

current\_id = current\_id + 1

return

**Step 6: Creating a Double-Linked List**

To create a double-linked list it feels natural just to extend the ListNode class by creating an additional reference to the *previous* node. This affects the methods for adding, removing, and sorting nodes. As shown in *Listing 11*, a new property named previous has been added to store the reference pointer to the previous node in the list. We'll change our methods to use this property for tracking and traversing nodes as well.

*Listing 11: Extended list node class*

class ListNode:

def \_\_init\_\_(self, data):

"constructor class to initiate this object"

# store data

self.data = data

# store reference (next item)

self.next = None

# store reference (previous item)

self.previous = None

return

def has\_value(self, value):

"method to compare the value with the node data"

if self.data == value:

return True

else:

return False

Now we are able to define a double-linked list as follows:

*Listing 12: A DoubleLinkedList class*

class DoubleLinkedList:

def \_\_init\_\_(self):

"constructor to initiate this object"

self.head = None

self.tail = None

return

def list\_length(self):

"returns the number of list items"

count = 0

current\_node = self.head

while current\_node is not None:

# increase counter by one

count = count + 1

# jump to the linked node

current\_node = current\_node.next

return count

def output\_list(self):

"outputs the list (the value of the node, actually)"

current\_node = self.head

while current\_node is not None:

print(current\_node.data)

# jump to the linked node

current\_node = current\_node.next

return

def unordered\_search (self, value):

"search the linked list for the node that has this value"

# define current\_node

current\_node = self.head

# define position

node\_id = 1

# define list of results

results = []

while current\_node is not None:

if current\_node.has\_value(value):

results.append(node\_id)

# jump to the linked node

current\_node = current\_node.next

node\_id = node\_id + 1

return results

As described earlier, adding nodes requires a bit more action. *Listing 13* shows how to implement that:

*Listing 13: Adding nodes in a double-linked list*

def add\_list\_item(self, item):

"add an item at the end of the list"

if isinstance(item, ListNode):

if self.head is None:

self.head = item

item.previous = None

item.next = None

self.tail = item

else:

self.tail.next = item

item.previous = self.tail

self.tail = item

return

Removing an item from the list similar costs have to be taken into account. *Listing 14* shows how to do that:

*Listing 14: Removing an item from a double-linked list*

def remove\_list\_item\_by\_id(self, item\_id):

"remove the list item with the item id"

current\_id = 1

current\_node = self.head

while current\_node is not None:

previous\_node = current\_node.previous

next\_node = current\_node.next

if current\_id == item\_id:

# if this is the first node (head)

if previous\_node is not None:

previous\_node.next = next\_node

if next\_node is not None:

next\_node.previous = previous\_node

else:

self.head = next\_node

if next\_node is not None:

next\_node.previous = None

# we don't have to look any further

return

# needed for the next iteration

current\_node = next\_node

current\_id = current\_id + 1

return

Listing 15 shows how to use the class in a Python program.

*Listing 15: Building a double-linked list*

# create three single nodes

node1 = ListNode(15)

node2 = ListNode(8.2)

node3 = ListNode("Berlin")

node4 = ListNode(15)

track = DoubleLinkedList()

print("track length: %i" % track.list\_length())

for current\_node in [node1, node2, node3, node4]:

track.add\_list\_item(current\_node)

print("track length: %i" % track.list\_length())

track.output\_list()

results = track.unordered\_search(15)

print(results)

track.remove\_list\_item\_by\_id(4)

track.output\_list()

As you can see, we can use the class exactly as before when it was just a single-linked list. The only change is the internal data structure.

**Step 7: Creating Double-Linked Lists with deque**

Since other engineers have faced the same issue, we can simplify things for ourselves and use one of the few existing implementations available. In Python, we can use the [deque](https://docs.python.org/3/library/collections.html#collections.deque) object from the collections module. According to the module documentation:

Deques are a generalization of stacks and queues (the name is pronounced "deck" and is short for "double-ended queue"). Deques support thread-safe, memory efficient appends and pops from either side of the deque with approximately the same O(1) performance in either direction.

For example, this object contains the following methods:

* append(): add an item to the right side of the list (end)
* append\_left(): add an item to the left side of the list (head)
* clear(): remove all items from the list
* count(): count the number of items with a certain value
* index(): find the first occurrence of a value in the list
* insert(): insert an item in the list
* pop(): remove an item from the right side of a list (end)
* popleft(): remove an item from the left side of a list (head)
* remove(): remove an item from the list
* reverse(): reverse the list

The underlying data structure of deque is a Python list which is double-linked. The first list node has the index 0. Using deque leads to a significant simplification of the ListNode class. The only thing we keep is the class variable data to store the node value. *Listing 16* is as follows:

*Listing 16: ListNode class with deque (simplified)*

from collections import deque

class ListNode:

def \_\_init\_\_(self, data):

"constructor class to initiate this object"

# store data

self.data = data

return

The definition of nodes does not change, and is similar to *Listing 2*. With this knowledge in mind we create a list of nodes as follows:

*Listing 17: Creating a List with deque*

track = deque([node1, node2, node3])

print("three items (initial list):")

for item in track:

print(item.data)

Adding an item at the beginning of the list works with the append\_left() method as *Listing 18* shows:

*Listing 18: Adding an element at the beginning of a list*

# add an item at the beginning

node4 = ListNode(15)

track.append\_left(node4)

print("four items (added as the head):")

for item in track:

print(item.data)

Similarly, append() adds a node at the end of the list as *Listing 19* shows:

*Listing 19: Adding an element at the end of the list*

# add an item at the end

node5 = ListNode("Moscow")

print("five items (added at the end):")

track.append(node5)

for item in track:

print(item.data)

**Conclusion**

Linked lists as data structures are easy to implement, and offer great usage flexibility. It is done with a few lines of code. As an improvement you could add a node counter - a class variable that simply holds the number of nodes in the list. This reduces the determination of the list length to a single operation with O(1), and you do not have to traverse the entire list.

For further reading and alternative implementations you may have a look here:

* llist - Linked list datatypes for Python (<https://pythonhosted.org/llist/>)
* collections - Container datatypes (<https://docs.python.org/3.6/library/collections.html>)