11.2 Traversing Linked Lists

In the previous section, we created a `LinkedList` object with three items, and used the following expressions to access each item in turn:

```
>>> linky._first.item
111
>>> linky._first.next.item
-5
>>> linky._first.next.next.item
9000
```

This is, of course, a very cumbersome way of accessing list elements! In this section, we'll study how to *traverse* a linked list: that is, how to write code that visits each element of a linked list one at a time, regardless of how long that linked list actually is. The basic structure of this code is quite general, so we will apply it to a bunch of different methods. This may seem repetitive, but linked list code is one of the most technically challenging and important parts of the course, so spend the time to master it!

Before we write code to traverse a linked list, let's remember how traversal might look for a Python `list`, manually using an index variable `i` to keep track of where we are in the list.[1]

[1] The following code is written to be a nice lead-in to linked list traversal. But you know better ways of iterating through a `list` in Python!

```
i = 0
while i < len(my_list):
    ... do something with my_list[i] ...
    i = i + 1
```

This code snippet consists of four parts:

1. Initialize the loop variable `i` (`0` refers to the starting index of the list).
2. Check if we've reached the end of the list in the loop condition.
3. In the loop, do something with the current element `my_list[i]`.
4. Increment the index loop variable.

This code takes advantage of the fact that Python already gives us a way to access individual list elements using the indexing operation (`[i]`). In a linked list, we don't have this luxury, and so the major difference to this pattern is that we now use a loop variable to

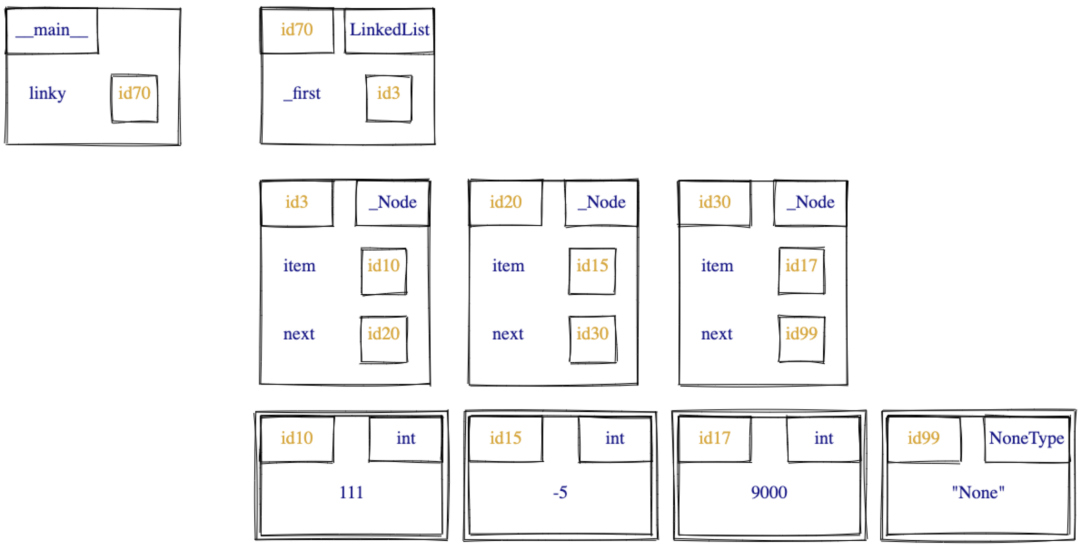
refer to the `_Node` object we're currently on in the loop, rather than the current index. Other than this change, the steps are exactly the same!

```
curr = my_linked_list._first   # 1. Initialize curr to the start of the list.
while curr is not None:        # 2. curr is None if we've reached the end of
         the list.
    ... curr.item ...          # 3. Do something with the current *element*,
         curr.item.
    curr = curr.next           # 4. "Increment" curr, assigning it to the
         next node.
```

For example, here is a `LinkedList` method that prints out every item in a linked list.

```
class LinkedList:
    def print_items(self) -> None:
        """Print out each item in this linked list."""
        curr = self._first
        while curr is not None:
            print(curr.item)      # Note: this is the only line we needed to
         fill in!
            curr = curr.next
```

Let's trace through the code on our example linked list from last time, as shown in the following memory model diagram:



Suppose we call `linky.print_items()`. What happens?

1. In the first line of the method body, `curr = self._first`, the loop variable `curr` is assigned to the first `_Node` in the linked list, with `id3` from our diagram.
2. Since `curr` is not `None`, the loop body executes.
    a. `curr.item`, which is `111`, is printed.

    b. Then `curr` is reassigned to `curr.next`, which is the `_Node` with `id20`.
3. Again, since `curr` is not `None`, the loop body executes.
    a. `curr.item`, which is `-5`, is printed.
    b. Then `curr` is reassigned to `curr.next`, which is the `_Node` with `id30`.
4. Again, since `curr` is not `None`, the loop body executes.
    a. `curr.item`, which is `9000`, is printed.
    b. Then `curr` is reassigned to `curr.next`, which is `None` (at `id99`).
5. Since `curr` is `None` now, the while loop stops and the method ends.

Linked list traversal and loop accumulators

Here is a second `LinkedList` method that combines our linked list traversal pattern with the loop accumulation pattern we studied back in Chapter 4. The "accumulation" part is code you've seen many times before, but now in the context of linked lists.

```
class LinkedList:
    def to_list(self) -> list:
        """Return a built-in Python list containing the items of this linked
         list.

        The items in this linked list appear in the same order in the
         returned list.
        """
        items_so_far = []

        curr = self._first
        while curr is not None:
            items_so_far.append(curr.item)
            curr = curr.next

        return items_so_far
```

So if we call `linky.to_list()` with our example linked list, `[111, -5, 9000]` is returned.[2]

[2] In the next section, we'll study how to implement the reverse: going from a built-in `list` to a linked list.

Reminder: how to use code templates

Just as we've seen with previous code templates like the loop accumulation pattern, the *linked list traversal pattern* is a good starting point for structuring your code when working with linked lists.

However, remember that all code templates are just a starting point! They are only meant to provide an overall code structure, and not to replace the hard work of actually thinking about how to write code. In other words, we use templates to make it easier to *get started* writing code.

Consider again our template for iterating through a linked list:

```
curr = self._first  # self is a LinkedList
while curr is not None:
    ... curr.item ...
    curr = curr.next
```

Whenever you're starting to write code to iterate through a linked list, your *first* step should be to copy-and-paste this template into your code. But that's the easy part; the next part involves the thinking required to fill in the ellipsis (`...`) and modify the template to suit your particular needs. As you progress through this course, you'll get lots of practice with that!

Looping with an index

Now let's consider a slightly different problem: indexing into a linked list, the equivalent of `my_list[3]`. Here is the header of the method we'll implement:[3]

[3] As you'll recall from Section 9.9, the double underscores in the name indicate that this is a *special method*, meaning `__getitem__` serves some special purpose in Python. We'll explain this at the end of this section.

```
class LinkedList:
    def __getitem__(self, i: int) -> Any:
        """Return the item stored at index i in this linked list.

        Raise an IndexError if index i is out of bounds.

        Preconditions:
            - i >= 0
        """
```

We can start by using our linked list traversal pattern, with a second loop variable to keep track of our current index in the list:

```
    def __getitem__(self, i: int) -> Any:
        """Return the item stored at index i in this linked list.

        Raise an IndexError if index i is out of bounds.

        Preconditions:
            - i >= 0
        """
        curr = self._first
        curr_index = 0
```

```
        while curr is not None:
            ... curr.item ...

            curr = curr.next
            curr_index = curr_index + 1
```

By updating `curr` and `curr_index` together, we get the following *loop invariant*:

> `curr` refers to the node at index `curr_index` in the linked list.

Now, there are two ways of completing the implementation of this method. The first is to use an *early return* inside the loop, stopping as soon as we've reached the node at the given index `i`:

```
    def __getitem__(self, i: int) -> Any:
        """..."""
        # Version 1
        curr = self._first
        curr_index = 0

        while curr is not None:
            if curr_index == i:
                return curr.item

            curr = curr.next
            curr_index = curr_index + 1

        # If we've reached the end of the list and no item has been returned,
        # the given index is out of bounds.
        raise IndexError
```

This approach builds on our "early return" pattern, and keeps the code quite simple. However, in other cases we might want to stop the loop early but execute some more code before returning. To do this, we can use a second approach of *modifying the while loop condition* so that the loop stops when it either reaches the end of the list or the correct index.

Compound loop conditions (with more than one logical expression) can be a bit tricky to reason about, and often it is more intuitive to write the stopping condition first, and then negate it to get the loop condition. In our case, our stopping condition is:

- `curr is None` (if we've reached the end of the list) *or* `curr_index == i` (if we've reached the correct index).

With this in mind, let's start a second implementation of `LinkedList.__getitem__`:

```
    def __getitem__(self, i: int) -> Any:
        """..."""
        # Version 2
        curr = self._first
        curr_index = 0

        while not (curr is None or curr_index == i):
            curr = curr.next
            curr_index = curr_index + 1

        ...  # How should we complete this?
```

After the loop stops, we know the loop condition is False, and so we know what must be True when the loop ends:

```
    def __getitem__(self, i: int) -> Any:
        """..."""
        # Version 2
        curr = self._first
        curr_index = 0

        while not (curr is None or curr_index == i):
            curr = curr.next
            curr_index = curr_index + 1

        assert curr is None or curr_index == i
        ...  # How should we complete this?
```

Writing the condition's negation as an explicit assert statement gives us insight into how to complete this method: we have two cases to consider, and can use an if statement to separate them out.

```
    def __getitem__(self, i: int) -> Any:
        """..."""
        # Version 2
        curr = self._first
        curr_index = 0

        while not (curr is None or curr_index == i):
            curr = curr.next
            curr_index = curr_index + 1

        assert curr is None or curr_index == i
        if curr is None:
            # index is out of bounds
            raise IndexError
        else:
```

```
            # curr_index == i, so curr is the node at index i
            return curr.item
```

Our second version is certainly more complex than the first version, but please study both carefully! We want you to get comfortable writing while loops with compound conditions, as you will encounter these more in this course and beyond.

`__getitem__` and list indexing expressions

Suppose we create a `LinkedList` object `linky` with the elements `111`, `-5`, `9000`. We can call our `__getitem__` method using dot notation, as you would normally expect:

```
>>> linky.__getitem__(0)
111
>>> linky.__getitem__(1)
-5
>>> linky.__getitem__(2)
9000
```

However, the `__getitem__` special method is also called automatically by the Python interpreter when we use *list indexing expressions*. That means that we can use the following syntax to index into our `LinkedList` object, just as we would a Python list:

```
>>> linky[0]  # Equivalent to linky.__getitem__(0)
111
>>> linky[1]  # Equivalent to linky.__getitem__(1)
-5
>>> linky[2]  # Equivalent to linky.__getitem__(2)
9000
```

Pretty amazing! We'll see a few more examples throughout this course of using Python's special methods to allow our classes to take advantage of built-in Python syntax and functions.