11.1 Introduction to Linked Lists

Back in Section 8.6, we learned that Python `lists` are an array-based implementation of the List ADT. This means they have some efficiency drawbacks: inserting and deleting items in a Python `list` can require shifting many elements in computer memory. In the extreme case, inserting and deleting at the *front* of a Python `list` takes $\Theta(n)$ time, where n is the length of the `list`, because every item in the list needs to be shifted.

In this chapter, we're going to study a different implementation of the List ADT that will attempt to address this efficiency shortcoming. To do so, we'll use a new data structure called the **linked list**. Our goal will be to create a new Python class that behaves exactly the same as the built-in `list` class, changing only what goes on in the private implementation of the class. This will mean that, ultimately, code such as this:

```
for i in range(n):
    nums.append(i)

print(nums)
```

will work whether `nums` is a Python `list` or an instance of the class we are going to write. We'll even learn how to make list indexing operations such as `nums[3] = 'spider'` work on instances of our class!

Specifying order through links

The reason why a Python `list` operations often requires elements to be shifted back and forth is that the elements of a Python `list` are stored in contiguous locations in memory. What if we didn't attempt to have this contiguity? If we had a variable referring to the first element of a list, how would we know where the rest of the elements were? We can solve this if we store along with each element a reference to the *next* element in the list.

This bundling of data—an element plus a reference to the next element–should suggest something familiar to you: the need for a new data type whose instance attributes are exactly these pieces of data. We'll call this data type a *node*, and implement it in Python as follows:[1]

[1] We use a preceding underscore for the class name to indicate that this entire class is *private*: it shouldn't be accessed by client code directly, but instead is only used by the "main" class described in the next section.

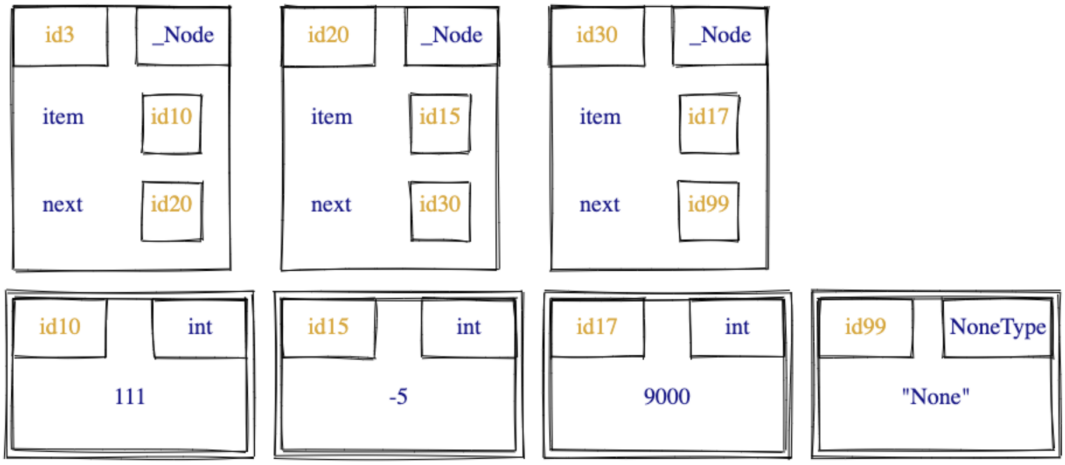
```
@dataclass
class _Node:
```

```
    """A node in a linked list.

    Instance Attributes:
      - item: The data stored in this node.
      - next: The next node in the list, if any.
    """
    item: Any
    next: Optional[_Node] = None  # By default, this node does not link to
         any other node
```

An instance of `_Node` represents a *single element* of a list; to represent a list of n elements, we need n `_Node` instances. The references in all of their `next` attributes link the nodes together into a sequence, even though they are not stored in consecutive locations in memory. It is these references, or "links", which give linked lists their name.

For example, here are three `_Node` objects that could represent the sequences of numbers `111, -5, 9000`.



Given a bunch of `_Node` objects, we can follow each `next` attribute to recover the sequence of items that these nodes represent. But how do we know where the sequence starts? Or put another way, in our above example how do we know there isn't another `_Node` object in memory referring to `id3`?

The `LinkedList` class

The second class we'll use in our list implementation is `LinkedList`, which will represent the list itself. This class is the one we want client code to use, and in it we'll implement methods that obey the same interface as the built-in `list` class. Our `LinkedList` class has a private attribute `_first` that refers to the first node in the list—this is how we know where the list starts, in answer to our question above.

Our first version of this class has a very primitive initializer that always creates an empty list.

```
class LinkedList:
    """A linked list implementation of the List ADT.
    """
    # Private Instance Attributes:
    #   - _first: The first node in this linked list, or None if this list is
    #     empty.
    _first: Optional[_Node]

    def __init__(self) -> None:
        """Initialize an empty linked list.
        """
        self._first = None
```

Example: building links manually

Of course, in order to do anything interesting with linked lists, we need to be able to create arbitrarily long linked lists! We'll see more sophisticated ways of doing this later, but for practice here we'll violate privacy concerns and just manipulate the private attributes directly.

```
>>> linky = LinkedList()  # linky is an empty linked list
>>> linky._first is None
True
>>> node1 = _Node(111)   # New node with item 111
>>> node2 = _Node(-5)    # New node with item -5
>>> node3 = _Node(9000)  # New node with item 900
>>> node1.item
111
>>> node1.next is None   # By default, new nodes do not link to another node
True
>>> node1.next = node2   # Let's set some links
>>> node2.next = node3
>>> node1.next is node2  # Now node1 links to node2!
True
>>> node1.next.item
-5
>>> node1.next.next is node3
True
>>> node1.next.next.item
9000
>>> linky._first = node1   # Finally, set linky's first node to node1
>>> linky._first.item      # linky now represents the list [111, -5, 9000]
111
>>> linky._first.next.item
-5
>>> linky._first.next.next.item
9000
```

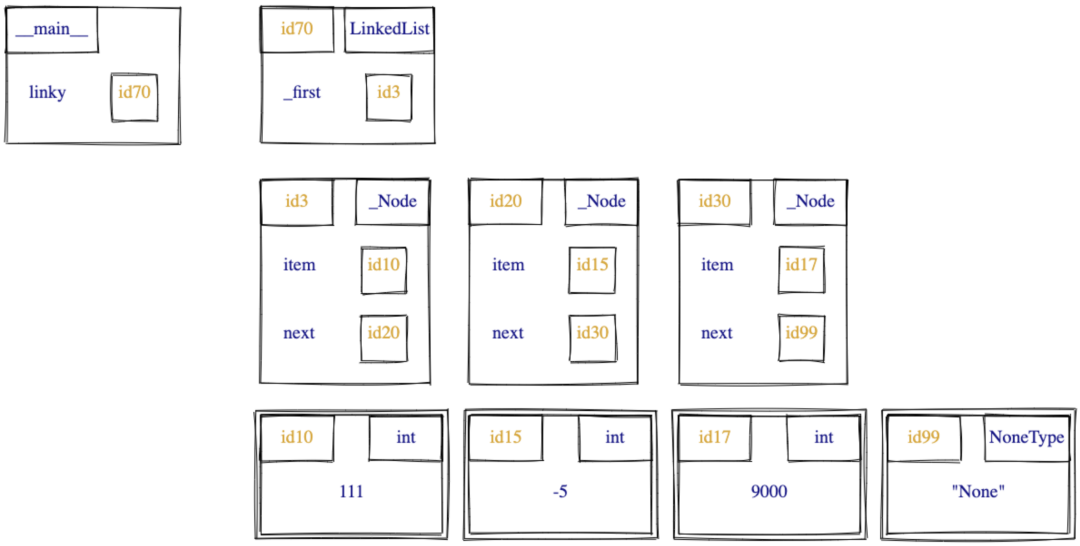
Warning: The most common mistake students make when first starting out with linked lists is confusing an individual `_Node` object with the item it stores. So in the example above, there's a big difference between `node3` and `node3.item`: the former is a `_Node` object containing the value `111`, while the latter *is* the value `111` itself!

```
>>> node3
_Node(item=9000, next=None)
>>> node3.item
9000
```

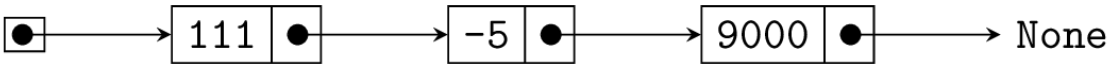
As you start writing code with linked lists, you'll sometimes want to operate on nodes, and sometimes want to operate on items. Making sure you always know exactly which type you're working with is vital to your success.

Linked list diagrams

The following is a diagram illustrating our above linked list example, with a variable `linky` referring to a `LinkedList` object representing the sequence `111, -5, 9000`. (We've omitted the other variables `node1`, `node2`, and `node3`.)



Because each element of a linked list is wrapped in a `_Node` object, complete memory model diagrams of linked lists are quite a bit larger than those corresponding to Python's array-based lists. While memory model diagrams are always a useful tool for understanding subtle memory errors—which certainly come up with linked lists!—they can be overkill if you want a quick and dirty linked list diagram. So below we show a stripped down version of this memory model diagram, writing the values of each item within the node boxes and using arrows to represent `next` attribute references. Note that we use a separate arrow from a smaller box representing the `_first` attribute of a `LinkedList`.


111
-5
9000
None