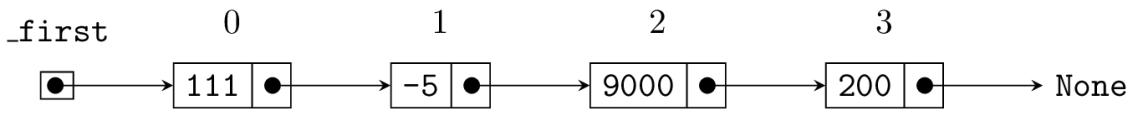
11.4 Index-Based Mutation

In the previous section, we looked at how to implement a simple mutating `LinkedList.append` method that adds an item to the end of a linked list. Now suppose we want to implement a more general form of insertion that allows the user to specify the list index where the new item should be inserted:[1]

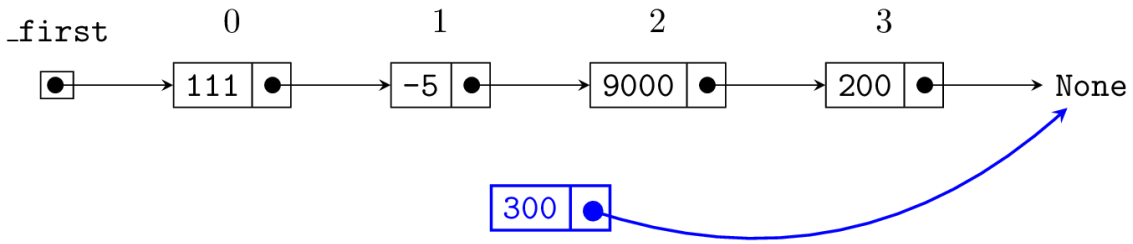
[1] This is analogous to the `list.insert` method.

```
class LinkedList:
    def insert(self, i: int, item: Any) -> None:
        """Insert the given item at index i in this linked list.

        Raise IndexError if i is greater than the length of self.

        If i *equals* the length of self, add the item to the end
        of the linked list, which is the same as LinkedList.append.

        Preconditions:
            - i >= 0

        >>> lst = LinkedList([111, -5, 9000, 200])
        >>> lst.insert(2, 300)
        >>> lst.to_list()
        [111, -5, 300, 9000, 200]
        """
```

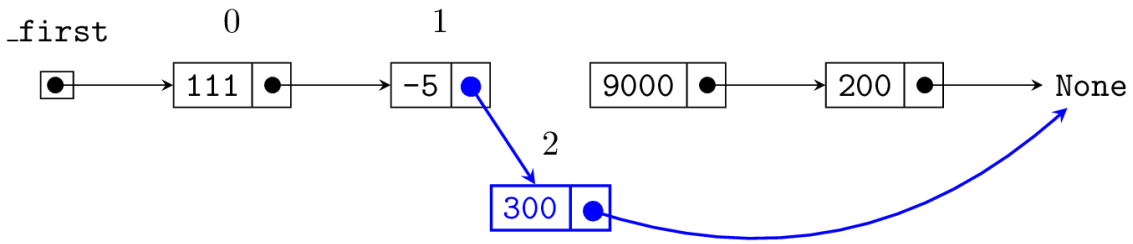
Before getting to any code, let's look at a few diagrams, illustrating our doctest example. First, here is the original linked list `linky`, consisting of the elements `111`, `-5`, `9000`, and `200`. We have added index labels above each node.



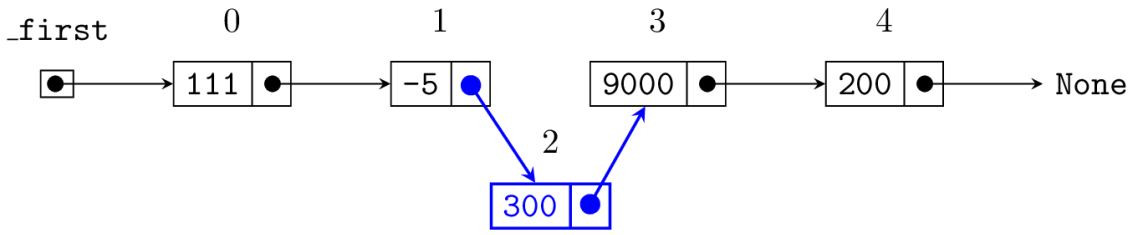
We want to insert item `300` at index 2 in this list. Like `LinkedList.append`, we can start by creating a new node, which by default links to `None`:



To insert the new node into the linked list, we need to modify some links. Because we want the new node to be at index 2, we need to modify the current node at index **1** to link to it:



But that's not all! If we just modify the index-1 node's link, we'll lose references to the other nodes after it. So we also need to modify the new node to link to the old node at index 2. Here is our final diagram:



Now let's put this into code!

Implementing `LinkedList.insert`

In general, if we want the node to be inserted into position `i`, we need to access the node at position `(i - 1)`. To do so, we can use the same approach as the `LinkedList.__getitem__` method we studied in the previous section.[2]

[2] You'll notice that we're using the "compound loop condition" approach, rather than the "early return" approach. After you're done reading this section, try re-writing this code using an early return, and compare the two implementations.

```
    def insert(self, i: int, item: Any) -> None:
        """..."""
        new_node = _Node(item)

        curr = self._first
        curr_index = 0

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

        # After the loop is over, either we've reached the end of the list
        # or curr is the (i - 1)-th node in the list.
        assert curr is None or curr_index == i - 1

        if curr is None:
            ...
        else: # curr_index == i - 1
            ...
```

Now, if `curr is None` then the list doesn't have a node at position `i - 1`, and so `i` is out of bounds. In this case, we should raise an `IndexError`.

On the other hand, if `curr is not None`, then we've reached the desired index, and can insert the new node using the same strategy as `append`, except we also need to remember to update the `next` attribute of the new node, as well.

```
    def insert(self, i: int, item: Any) -> None:
        """..."""
        new_node = _Node(item)

        curr = self._first
        curr_index = 0

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

        # After the loop is over, either we've reached the end of the list
        # or curr is the (i - 1)-th node in the list.
        assert curr is None or curr_index == i - 1

        if curr is None:
            # i - 1 is out of bounds. The item cannot be inserted.
            raise IndexError
        else: # curr_index == i - 1
            # i - 1 is in bounds. Insert the new item.
```

```
            new_node.next = curr.next
            curr.next = new_node
```

Warning! Common error ahead! (and solution)

When writing mutating methods on linked lists, we very often update the links of individual nodes to add and remove nodes in the list. We must be very careful when doing so, because the order in which we update the links really matters, and often only one order results in the correct behaviour.

For example, this order of link updates in the final `else` branch doesn't work:

```
curr.next = new_node
new_node.next = curr.next
```

On the second line, `curr.next` has already been updated, and its old value lost. The second line is now equivalent to writing `new_node.next = new_node`, which is certainly not what we want!

The reason this type of error is so insidious is that the code *looks* very similar to the correct code (only the order of lines is different), and so you can only detect it by carefully tracing through the updates of the links line-by-line.

This is a good time to remind you of *parallel assignment* in Python, which allows us to write the two attribute reassignments in a single statement, without worry about order. These two parallel assignment statements are *equivalent*, and both do what we want:

```
# Version 1
curr.next, new_node.next = new_node, curr.next

# Version 2
new_node.next, curr.next = curr.next, new_node
```

Tidying up: don't forget about corner cases!

Our `insert` implementation has one problem: what if `i == 0`? In this case, it doesn't make sense to iterate to the (`i-1`)-th node! Just as we saw with `LinkedList.append`, this is a special case that we need to handle separately by modifying `self._first`. Here is our final, complete code for this method. Notice our use of parallel assignment in both of these cases.

```
    def insert(self, i: int, item: Any) -> None:
        """..."""
        new_node = _Node(item)
```

```
        if i == 0:
            # Insert the new node at the start of the linked list.
            self._first, new_node.next = new_node, self._first
        else:
            curr = self._first
            curr_index = 0

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

            # After the loop is over, either we've reached the end of the
         list
            # or curr is the (i - 1)-th node in the list.
            assert curr is None or curr_index == i - 1

            if curr is None:
                # i - 1 is out of bounds. The item cannot be inserted.
                raise IndexError
            else: # curr_index == i - 1
                # i - 1 is in bounds. Insert the new item.
                curr.next, new_node.next = new_node, curr.next
```

Exercise: Index-based deletion

The analogue of Python's `list.insert` is `list.pop`, which allows the user to *remove* an item at a specified index in a list. Because this is quite similar to insertion, we won't develop the full code here, but instead outline the basic steps in some pseudo-code:

```
class LinkedList:
    def pop(self, i: int) -> Any:
        """Remove and return item at index i.

        Preconditions:
            - i >= 0

        Raise IndexError if i >= the length of self.

        >>> lst = LinkedList([1, 2, 10, 200])
        >>> lst.pop(2)
        10
        >>> lst.pop(0)
        1
        >>> lst.to_list()
        [2, 200]
        """
        # 1. If the list is empty, you know for sure that index is out of
         bounds...
```

```
        # 2. Else if i is 0, remove the first node and return its item.

        # 3. Else iterate to the (i-1)-th node and update links to remove
        #    the node at position index. But don't forget to return the item!
```