CSC 148: Introduction to Computer Science

Week 5

Linked lists

It's all about the links...

University of Toronto Mississauga,

Department of Mathematical and Computational Sciences

Reminder: Midterm Next Week!

- Please see the post on the discussion board for more info.

Summary:

- Material through Friday is fair game.
- Expect the problems to look like the exercises we do in class and in lab.
- Please arrive *to the room you are assigned by last name* by 7:05.

There are two major list implementations

- Array-based lists store references to elements in contiguous blocks of memory.

- Linked lists can store elements anywhere, but each element must store a reference to the *next* element in the list.

So what's the difference, really?

- Python lists are flexible and useful, but …
  - They allocate large blocks of contiguous memory, which becomes increasingly difficult as memory is in use
  - Some operations can be expensive.
    - Stack that uses list - add/remove items at the end vs. the front...

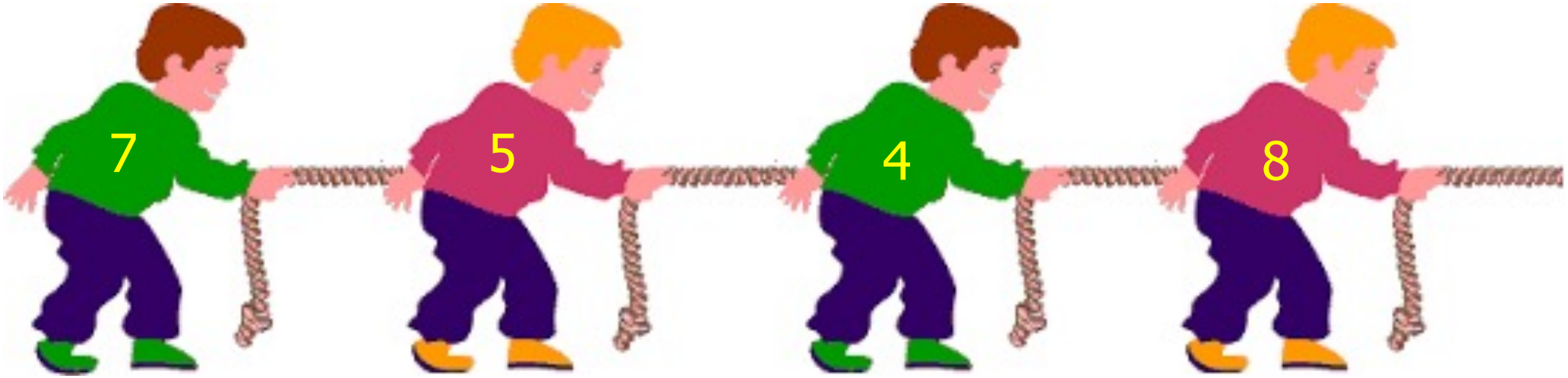
- Linked list nodes reserve just enough memory for the object value they refer to, a reference to it, and a reference to the next node in the list.
  - … but there are memory access drawbacks! We'll talk about this in CSC258.

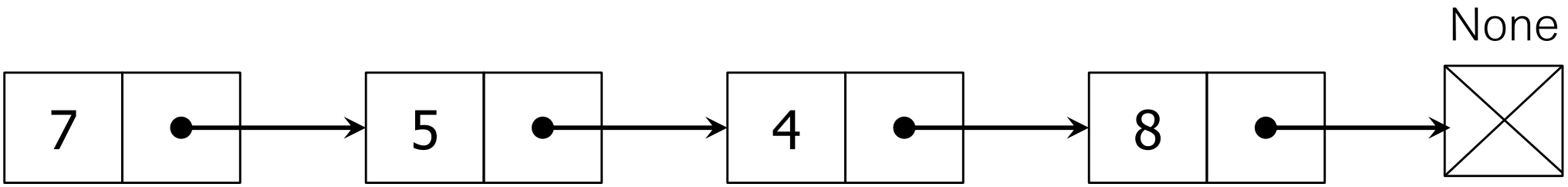
Our Goals for W5

1. Work with linked lists by implementing same operations as Python's built-in `list`.

2. Analyze the running time of our linked list methods and compare them to the array-based `list`.

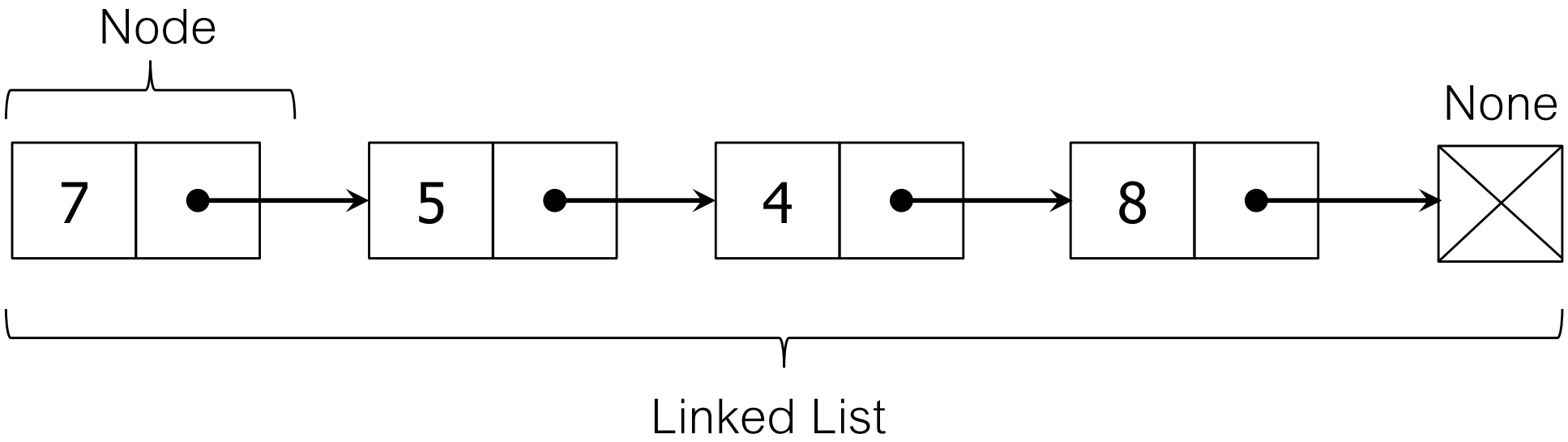
How to look at a linked list ...





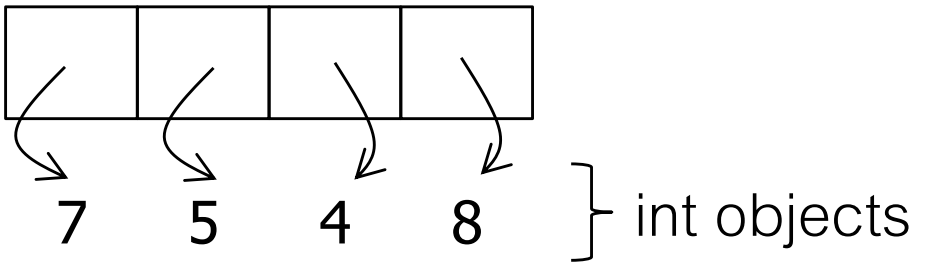
Linked List

- There are two useful, but different, ways of thinking of linked list nodes:
  - 1. As a list made up of an item (value) and a sub-list (rest)
  - 2. As objects (nodes), each containing a value and a reference to another similar node object (the “next link in the chain”)

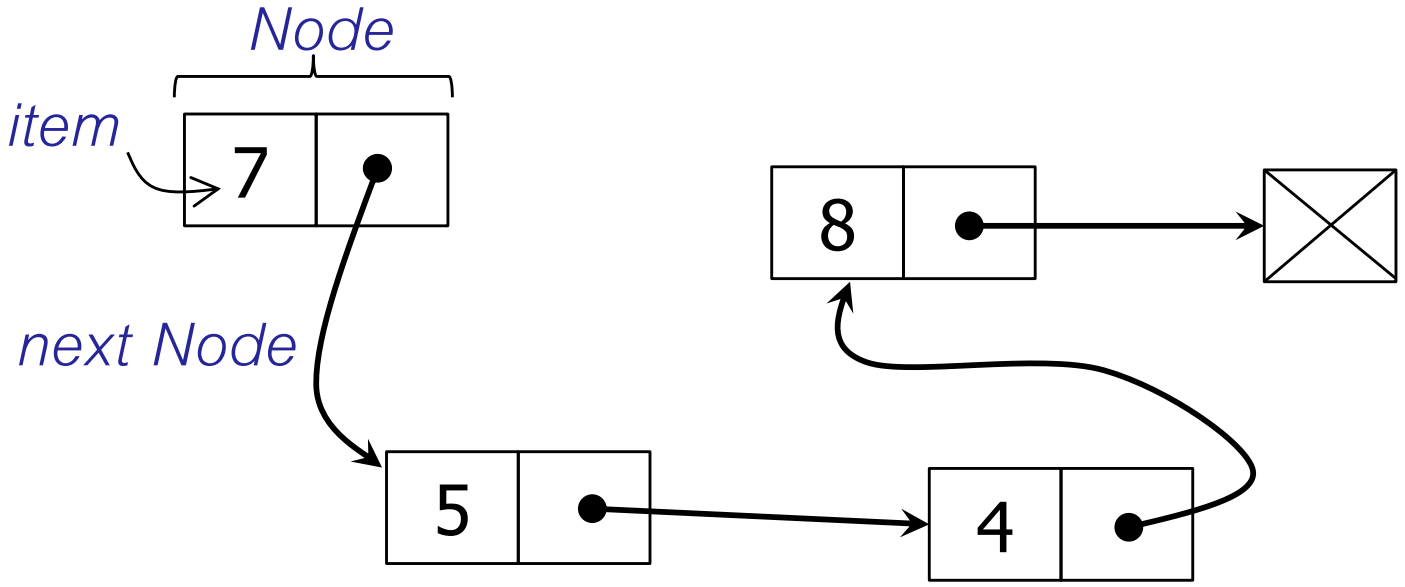


Linked List Nodes

Python list = [7, 5, 4, 8]



Linked list:

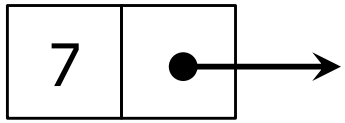


- Get in the habit of drawing diagrams to visualize things better ...
- Let's design a linked list node, then a separate "wrapper" to represent the linked list as a whole ...

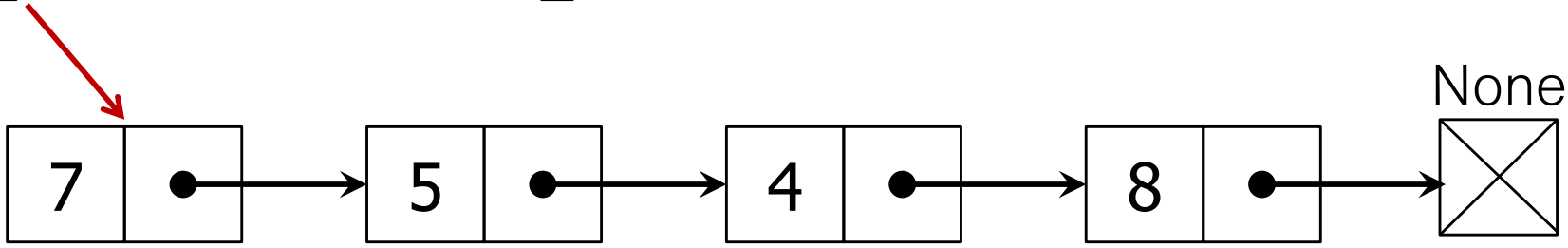
Code Summary

Data structures:

```
class _Node:
    item: Any
    next: Optional[_Node]
```




```
class LinkedList:
    _first: Optional[_Node]
```



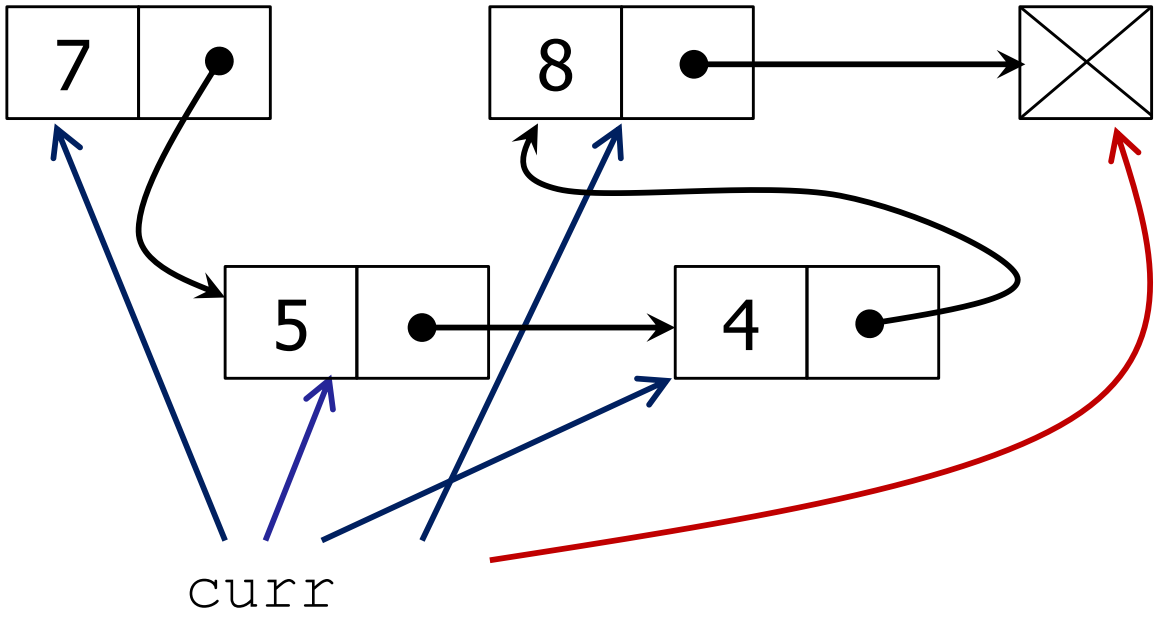
Traversing (Walking) a list

- Make a reference to (at least one) node, and move it along the list:

```
curr = self._first
while <some condition here...>:
    # do something here ...
    curr = curr.next
```

very common pattern

What if curr is None?



__contains__

- Check (possibly) every node

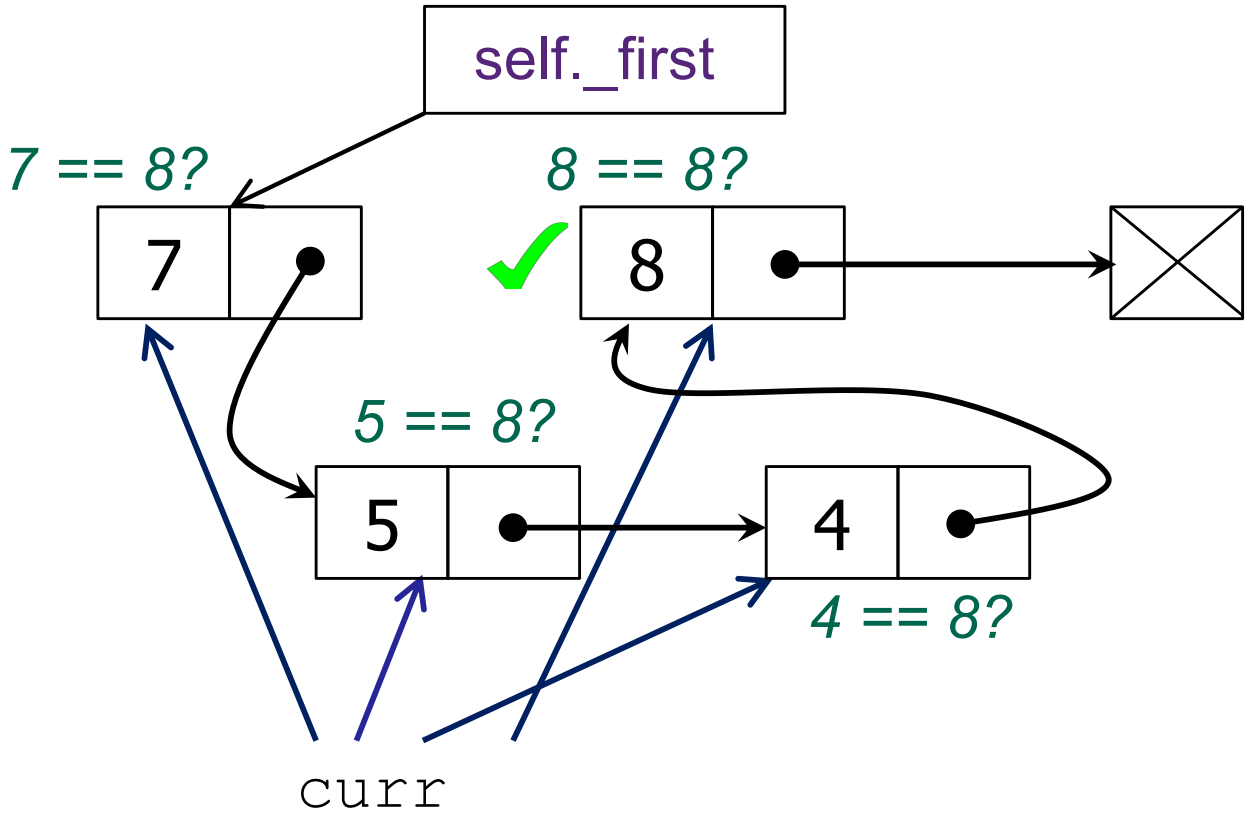
```
curr = self._first
while <some condition here...>:
    # do something here ...
    curr = curr.next
```

Question1:

Does the linked list contain 8?

Question2:

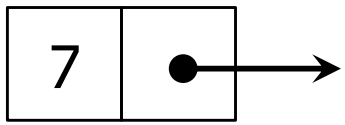
Does the linked list contain 3?



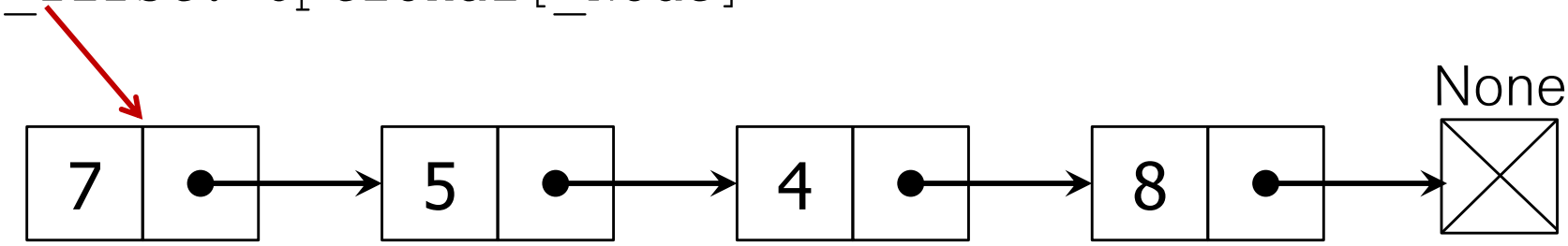
So far ..

Data structures:

```
class _Node:
    item: Any
    next: Optional[_Node]
```




```
class LinkedList:
    _first: Optional[_Node]
```



Traversal idea:

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

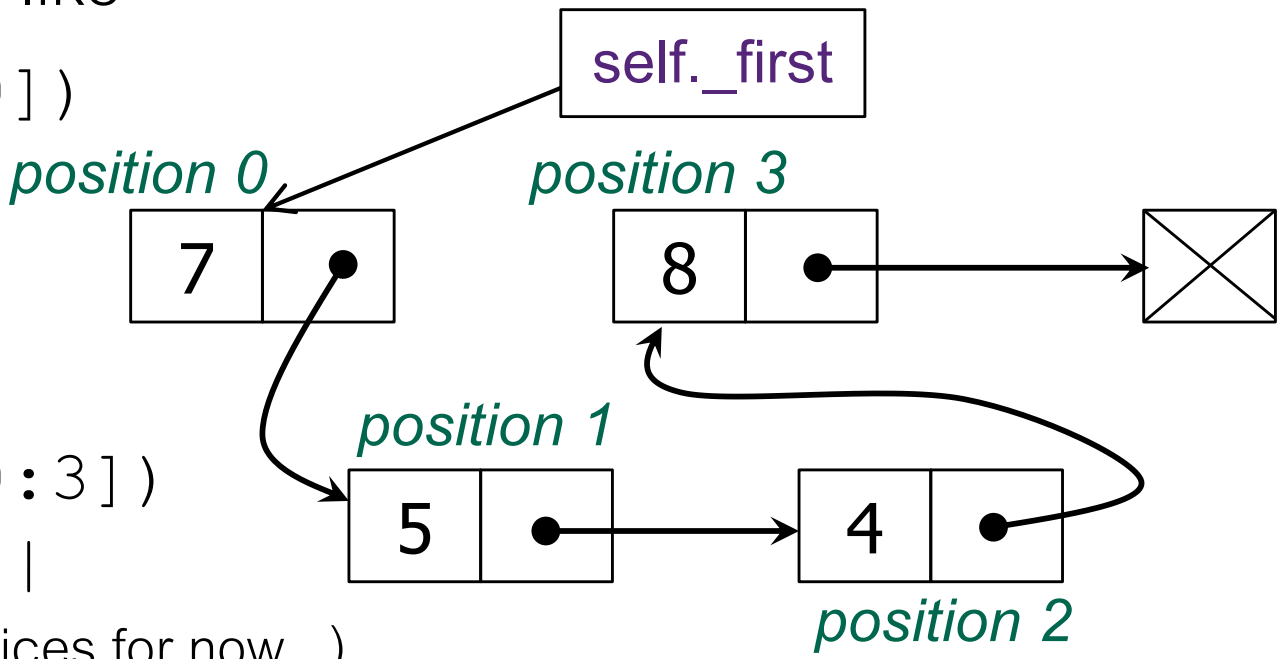
Worksheet 1

- Practice with traversing a list!
  - Think of what we've learnt so far: how to advance in the list, stopping condition, etc..
  - Having things like __eq__ and __getitem__ can be useful!

__getitem__

- Should enable things like

```
>>> print(lnk[0])
7
```



... or even this:

```
>>> print(lnk[0:3])
7 -> 5 -> 4 -> |
```

(we'll ignore support for slices for now...)

- What corner cases do we have to be careful about?
- How do we handle them?

Takeaways

- Code templates are useful.
- Code templates aren't everything.

- Writing a stopping condition is often *easier to understand* than writing a loop condition.