The Linked List

Representing sequences as elements linked together
Implementing Stacks with Linked Lists
Implementing Queues with Linked Lists

Why was the Queue implementation so complex?

- we were using the list provided by Python, which reserves a block of memory, and then manages the memory and provides access
  - Python lists are efficient for their intended use
- But to make sure it was efficient for what we wanted to do, we had to take control of the memory management
  - we had to stop Python auto-managing the list space

How else could we do it, without relying on a Python list?

- our own 'array' ? (i.e. reserve a block of memory, and place sequential items next to each other)
- free space?

Other array-based implementations

Any other array-based implementation is going to run into the same problems.

- How much space should we reserve?
- What happens when we fill that space?
- Can we cope with empty cells in the block?

Python's list structure is already well implemented, and hides a lot of the complexity.

Anything we write would have to be just as complex, and probably less efficient ...

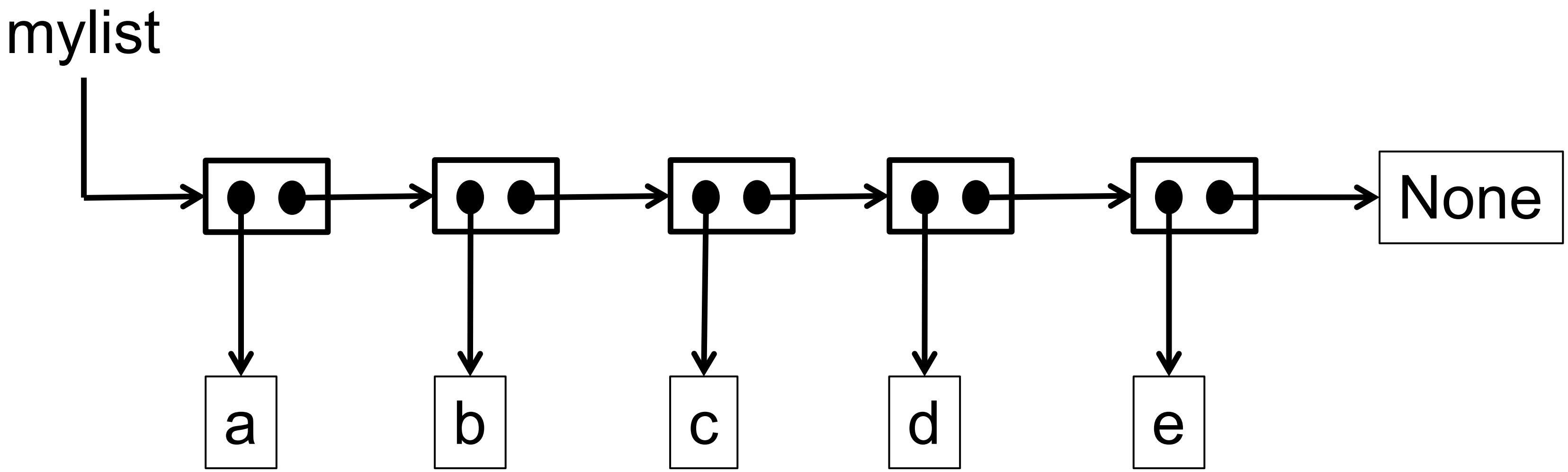
Free storage of list elements

Remember that a list is a sequence of *references* to items, and each item is stored individually by Python

We will store each *reference* individually
 - don't require them to be stored in consecutive memory

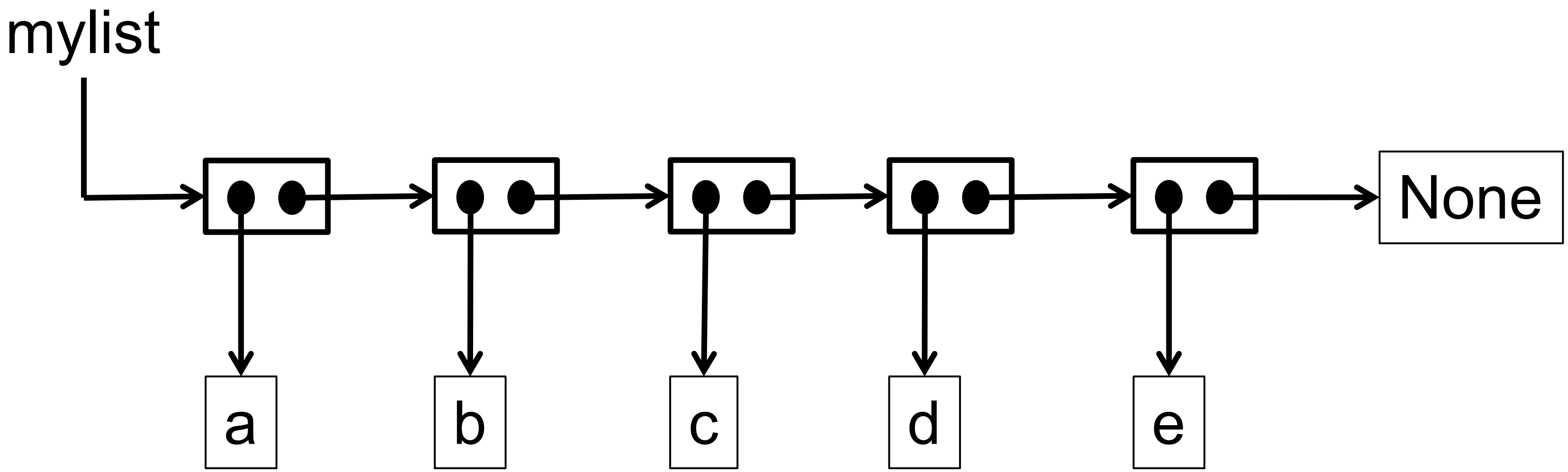
Let Python decide where to put them.

BUT
- with each reference, also store a reference to the next one in the list.



No space management issues – as long as there is any memory left for Python to use anywhere, we can create a new element object, and a new 'list node' object.

But we have to do the work to read the element
- we can't now rely on the efficient list lookup



How should we design the classes?
What operations can we implement?

```python
class SLLNode:
    def __init__(self, item, nextnode):
        self.element = item         #any object
        self.next = nextnode        #an SLLNode


class SLinkedList:
    def __init__(self):
        self.first = None           #an SLLNode
        self.size = 0               #an integer

    def add_first(self, item):     #add at front of list

    def get_first(self):           #report the first element

    def remove_first(self):        #remove the first element

    def length(self):              #report the number of elements
```

1st version

```
def test_linkedlist():
    mylist = SLinkedList()
    mylist.add_first(‘d')
    mylist.add_first('c')
    mylist.add_first(‘e')
    mylist.remove_first()
    mylist.add_first(‘b’)
    mylist.add_first(‘a’)
    print('mylist =', mylist)
    print('length =', mylist.length())
    print('first =', mylist.get_first())
    print('first (removed) =', mylist.remove_first())
    print('mylist now =', mylist)
    print('length =', mylist.length())
    mylist.remove_first()
    mylist.remove_first()
    mylist.remove_first()
    print('length =', mylist.length())
    print('first (None?) =', mylist.get_first())
    print('first removed (None?) =', mylist.remove_first())
    mylist.add_first('f')
    print('mylist (f) =', mylist)
    print('length =', mylist.length())
```

```
add_first(self, element):
get_first(self):
remove_first(self):
```

```python
def add_first(self, element):
    node = SLLNode(element, self.first)
    self.first = node
    self.size = self.size + 1
```

O(1)

```python
def get_first(self):
    if self.size == 0:
        return None
    return self.first.element
```

O(1)

```python
def remove_first(self):
    if self.size == 0:
        return None
    item = self.first.element
    self.first = self.first.next
    self.size = self.size - 1
    return item
```

O(1)

Implementing the Stack ADT

Since the singly-linked list behaves like a Stack, it is easy to implement the Stack ADT using a SLinkedList …

Stack methods:	LinkedList methods:
push(item)	add_first(item)
pop()	remove_first()
top()	get_first()

```python
class Stack:
    def __init__(self):
        self.body = SLinkedList()
        #front of list is top of stack

    def push(self, element):
        self.body.add_first(element)

    def pop(self):
        return self.body.remove_first()

    def top(self):
        return self.body.get_first()

    def length(self):
        return self.body.length()
```

basic implementation
- no private variables
- no __str__() method

```python
class Stack:
    def __init__(self):
        self.first = None
        self.size = 0

    def push(self, element):
        self.first = SLLNode(element, self.first)
        self.size = self.size + 1

    def pop(self):
        if self.size == 0:
            return None
        item = self.first.element
        self.first = None
        size = size - 1
        return item

    def top(self):
        if self.size == 0:
            return None
        return self.first.element

    #...
```

basic implementation
- no private variables
- no __str__() method

Alternative – implementing linked list behaviour directly into a Stack class …

Testing the LinkedList implementation

Use all the test methods and other functions we wrote to test or use the previous Stack implementation.

If we have implemented these test methods properly, then we do not make any reference to the internals of the Stack class, and so all previous methods should still work

```
def postfix(string):
    """ Evaluate postfix string, using a stack.
        Elements must be separated by spaces.
    """

    tokenlist = string.split()
    stack = Stack()
    for token in tokenlist:
        if token in ["+", "-", "*", "/"]:
            second = stack.pop()
            first = stack.pop()
            if token == "+":
               stack.push(first + second)
            elif token == "-":
               stack.push(first - second)
            elif token == "*":
               stack.push(first * second)
            else:
               stack.push(first / second)
        else:
            stack.push(int(token))
    return stack.pop()
```

Complexity of operations

```
class Stack:
    def __init__(self):
        self.body = SLinkedList()

    def push(self, element):
        self.body.add_first(element)

    def pop(self):
        return self.body.remove_first()

    def top(self):
        return self.body.get_first()

    def length(self):
        return self.body.length()
```

O(1)

O(1)

O(1)

O(1)

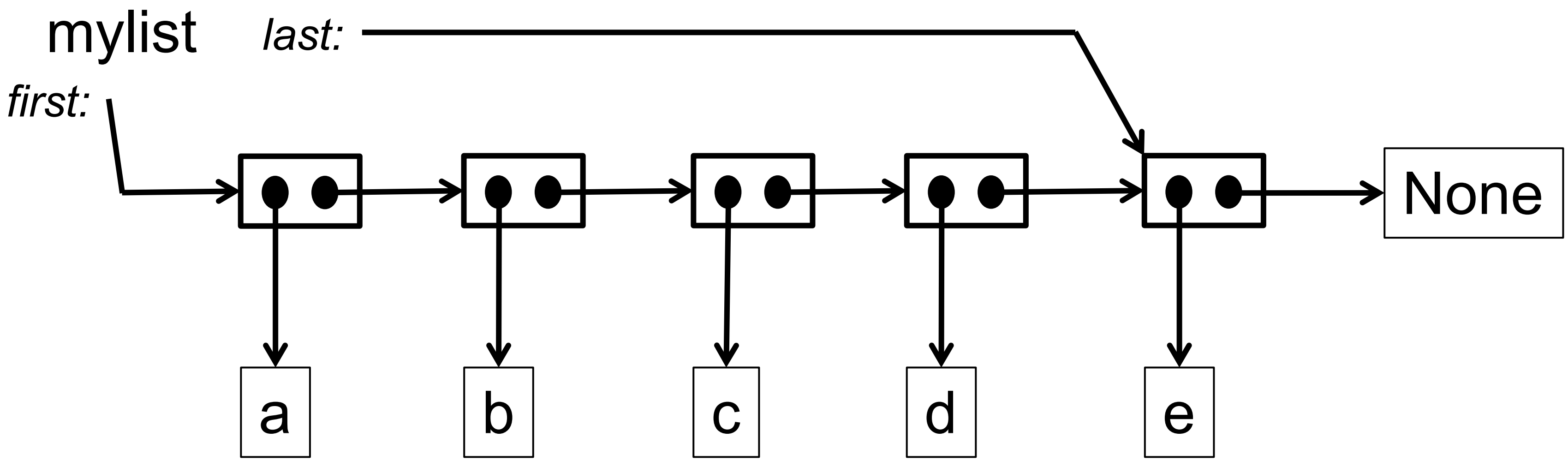
Note: genuine O(1) for each operation, not just on average – i.e. no hidden processes copying list elements into new space, and no occasional unexpected delays

Implementing the Queue ADT

Queue methods:	LinkedList methods:
enqueue(item)	?
dequeue()	?
front()	?

The Queue requires operations at both ends of the sequence, but the LinkedList only gives us access to the front …

Can we modify the LinkedList implementation?


mylist
last:
first:
None
a
b
c
d
e

```
add_last(self, element):
get_last(self):
```

Implementing the Queue ADT (2)

Queue methods:	LinkedList methods:
enqueue(item)	add_last(item)
dequeue()	remove_first()
front()	get_first()

Enqueue (i.e. add) at the end of the linked list
Dequeue (i.e. remove) at the front of the linked list

Implementing a Queue with a singly linked list

```
class QueueSLL:
    def __init__(self):
        self.body = SLinkedList()

    def enqueue(self, element):
        self.body.add_last(element)

    def dequeue(self):
        return self.body.remove_first()

    def first(self):
        return self.body.get_first()

    def length(self):
        return self.body.length()
```

all operations are O(1)

Next lecture ...

No lecture on Wednesday 11th October
Next lecture is Friday 13th october