Linked lists

When we discussed the implementation of stacks, we discovered that the only way to ensure constant time for each operation without knowing in advance the size of the stack is to work with small node objects that are linked together.

Let's now generalize this idea and see how to implement a sequence with many useful operations like insertions and deletions. This data structures is called a *linked list*. Some data structure books and some libraries will simply call it "list," but since Python already has a `list` data structure (which is *not* a linked list), we will always explicitely say "linked list."

Basic ideas

As in the stack implementation, we need a small node object that stores an element, and also links to another node object:

```
class Node:
  def __init__(self, el, next=None):
    self.el = el
    self.next = next
  def __repr__(self):
    return "<" + repr(self.el) + ">"
```

We have added a `repr`-method to help with debugging.

By creating nodes and linking them together, we can create a short linked list with three strings:

```
>>> a = Node("apples")
>>> a = Node("oranges", a)
>>> a = Node("strawberries", a)
>>> a
<'strawberries'>
>>> a.next
<'oranges'>
>>> a.next.next
<'apples'>
>>> a.next.next.next is None
True
```

Let's write a function that will display all the elements of a list. One approach is to do it recursively:

```
def display(a):
  if a is not None:
    print(a.el)
    display(a.next)
```

This is the output using the linked list we made above:

```
>>> display(a)
strawberries
oranges
apples
```

It is natural to use recursive functions to work with linked lists, since linked lists can be defined recursively: a linked list is either empty (that is, `None`), or consists of a `Node` whose `next` field points to a linked list.

As usual with recursive functions, we have to worry about stack overflow. With this function, we wouldn't be able to display a linked list with more than 1000 elements—that's not a lot.

Let's solve the problem by rewriting the function using a loop, without recursion:

```
def display(a):
  while a is not None:
    print(a.el)
    a = a.next
```

Note how the local variable `a` walks through the linked list while printing out its elements.

A LinkedList class

Let's get more serious and make a class to manage a linked list:

```
class LinkedList:
  def __init__(self):
    self._front = None

  def first(self):
    if self._front is None:
      raise EmptyListError
    return self._front

  def is_empty(self):
    return self._front is None

  def __repr__(self):
    if self.is_empty():
      return "[]"
    res = "["
    p = self._front
    while p is not None:
      res += repr(p.el)
      if p.next is not None:
        res += ", "
      p = p.next
    res += "]"
    return res
```

The constructor creates an empty list. The `LinkedList` object stores only a reference to the first element of the list, called the front. The `first()` method returns this front node. We have added a `repr`-conversion method that walks through the linked list and builds a string representation.

Let's now look at all the operations we will need:

prepend(el). This method adds a new node at the front of the list. We need to create a new `Node` object storing the element. The `next` of this node should be the old linked list, while the new node becomes the new front node. This results in the following one-liner:

```
  def prepend(self, el):
    self._front = Node(el, self._front)
```

remove_first(). This method removes the front node, thus making the linked list one element shorter.

A first attempt gives another one-liner:

```
  def remove_first(self):
    self._front = self._front.next
```

However, we missed some error-handling here: if the list is empty, we should raise an `EmptyListError`. So our final code is this:

```
def remove_first(self):
  if self._front is None:
    raise EmptyListError
  self._front = self._front.next
```

insert_after(n, el). This method inserts a new node into the linked list just *after* the given node `n`. Again, we need to create a new node storing `el`. We will need to change `n.next` to point to this new node, while the new node will point to the old `n.next`. This gives the following one-liner:

```
def insert_after(self, n, el):
  n.next = Node(el, n.next)
```

Note that we need no special error handling here: this code will work correctly even if `n` is the last node of the linked list.

remove_after(n). This method deletes the node *after* node `n` from the linked list, making the list one element shorter. Here, we need to check if `n` is the last node, and obtain the following code:

```
def remove_after(self, n):
  if n.next is None:
    raise ValueError(n)
  n.next = n.next.next
```

before(n). This method returns the node just *before* the given node `n`. This is tricky: there is no way we can go from `n` to the node before it, so the only way to do this is to start at the front of the list and walk through the list until we find a node whose `next` field is equal to `n`.

```
def before(self, n):
  p = self._front
  while p.next != n:
    p = p.next
  return p
```

last(). This method returns the *last* node of the linked list. Again, there is no easy way to do this: we must walk through the list to find the last node:

```
def last(self):
  p = self._front
  while p.next != null:
    p = p.next
  return p
```

Length of linked list. Similarly, to compute the length of a linked list, we must walk through the list and count node objects. Here is the magic method that makes the Python function `len` work:

```
def __len__(self):
  if self.is_empty():
    return 0
  p = self._front
  count = 0
```

```
    while p is not None:
      count += 1
      p = p.next
    return count
```

Linked list with fast append

Most operations on linked lists take constant time, but appending at the end requires linear time: we have to walk through the list to find the end before we can add a new node there.

If we want to support `append` in constant time, then the `LinkedList` object must store a reference to the last node:

```
class LinkedList:
  def __init__(self):
    self._front = None
    self._rear = None
```

Now, `append` is in principle fast (and easy):

```
  def append(self, x):
    if self._front is None:
      self._front = _Node(x, None)
      self._rear = self._front
    else:
      self._rear.next = _Node(x, None)
      self._rear = self._rear.next
```

But we have to add quite a bit of code to our class to make sure that the `_rear` field is updated correctly whenever we make a change to the list. For instance, `prepend` must also update `_rear` when the list was empty.

Linked queues

A linked list with fast append directly provides all the operations we need for a queue. Example code `linkedqueue.py` contains a full queue implementation with this technique. All operations work in constant time.