Outline

Using the ListNode Class

Ideas about Linked List Implementation

We have a pretty good feeling how linked structures can be used to represent sequences by now.

We need to be very careful with the link manipulation so that items don't get lost or the structure corrupted.

This is a perfect place to employ the idea of ADT:
we can encapsulate all of the details of the linked structure and manipulate that structure through some high-level operations that `insert` and `delete` items.

Using the ListNode Class

API

We will borrow a subset of Python list API (*Application Programming Interface*).

Application Programming Interface is the set of values, operations, and objects provided by a code library or framework.

Using the ListNode Class

Using the ListNode Class

For a true ADT to build we would need to add `get_item`, `set_item`, `get_link`, and `set_link` to `ListNode`.

An alternative: let's create a class `LList` that will use class `ListNode`, i.e. an Abstract Data Type which will provide the necessary interface operations for its objects to behave like lists, and will be `ListNode`'s only "customer".

Since no other class will use ListNode objects, we don't provide public accessors or mutators (`get_item`, `get_link`, `set_item`, `set_link`) for (private) `ListNode` attributes.
Rather, we allow `LList` to access the attributes directly via dot-notation.

Properties of the LList Class

Thoughts about LList class

`LList` class will maintain its data as a linked sequence of `ListNodes`.

An `LList` object should have an instance variable pointing to the first node in its sequence, called head.

It is also convenient to keep track of the number of items in the list.

Properties of the LList Class

Class Invariants

A Class Invariant of a class is a condition which must be true for the concrete representation of every instance (object) of that class. For the `LList` class, these are:

- `self.size` is the number of nodes currently in the list.
- If `self.size == 0` then `self.head` is `None`; otherwise `self.head` is a reference to the first `ListNode` in the list.
- The last `ListNode` (at position `self.size - 1`) has its link set to `None`, and all other `ListNode` links refer to the next `ListNode` in the list.

Methods of the LList Class

__INIT__

```
def __init__(self, seq=()):
   """ creates an LList
   post:  Creates an LList containing items in seq"""

   if seq == ():
      self.head = None
   else:
      self.head = ListNode(seq[0], None)
      last = self.head
      for item in seq[1:]:
         last.link = ListNode(item, None)
         last = last.link
   self.size = len(seq)
```

Methods of the LList Class

__LEN__

```
def __len__(self):
    """ post:  returns number of items in the list """
    return self.size
```

Methods of the LList Class

_find

This method will be called from other methods as needed.

```
def _find(self, position):
   """private method that returns node that is at location
position in the list
   pre:  0<= position < self.size
   post:  returns the ListNode at the specified position
in the list"""

   assert 0 <= position < self.size
   node = self.head
   # move forward until we reach the specified node
   for i in range(position):
      node = node.link
   return node
```

Methods of the LList Class

APPEND

```
def append(self, x):
   """ appends x onto the end of list
   post:  x is appended onto the end of the list"""

   # create a new node containing x
   newNode = ListNode(x)     # non-empty list
   if self.head is not None:
      node = self._find(self.size - 1)
      node.link = newNode
   else:           # empty list
      self.head = newNode
   self.size += 1
```

Methods of the LList Class

__GETITEM__

Indexing - when the square brackets are used to access an item in the list.

```
def __getitem__(self, position):
   """ return data item at the location position
   pre:  0 <= position < self.size
   post:  returns data item at the specified position"""

   node = self._find(position)
   return node.item
```

Methods of the LList Class

__SETITEM__

Indexing - when the square brackets are used on the left-hand side of an assignment statement.

```
def __setitem__(self, position, value):
   """ set data item at the location position to value
   pre:  0 <= position < self.size
   post:  sets the data item at the specified position to
value"""

   node = self._find(position)
   node.item = value
```

Methods of the LList Class

__delitem__

```
def __delitem__(self, position):
   """ delete item at location position from the list
   pre:  0 <= position < self.size
   post:  the item at the specified position is removed
from the list"""

   assert 0 <= position < self.size
   self._delete(position)
```

Methods of the LList Class

_delete

```
def _delete(self, position):
   """ private method to delete item at location position
   pre:  0 <= position < self.size
   post:  the item at the specified position is removed from the
list and the item is returned (for use with pop)"""
   if position == 0:
      item = self.head.item
      self.head = self.head.link
   else:
      prev_node = self._find(position - 1)
      item = prev_node.link.item
      prev_node.link = prev_node.link.link
   self.size -= 1
   return item
```

Methods of the LList Class

pop

```
def pop(self, i=None):
   """ returns and removes item at position i from list, the
default is to return and remove the last item
   pre:  self.size> 0 and (i is None or (0 <= i < self.size))
   post:  if i is Nonve, the last items is removed and returned;
otherwise the ith item is removed and returned"""

   assert self.size > 0 and (i is None or (0 <= i <
self.size))
   if i is None:
      i = self.size - 1
   return self._delete(i)
```

Methods of the LList Class

INSERT

```
def insert(self, i, x):
   """inserts a at position i in the list
   pre:  0 <= i < self.size
   post:  x is inserted into the list a position i and old
elemends from position i...oldsize-1 are at positions
1+1...newsize-1"""

   assert 0 <= i <= self.size
   if i == 0:
      self.head = ListNode(x, self.head)
   else:
      node = self._find(i - 1)
      node.link = ListNode(x, node.link)
   self.size += 1
```

A Common Problem For any Container Class: Traversal

Iteration is an Abstraction of Traversal

Container classes can provide efficient access to their contents in various ways:

- random access indexed: (arrays, Python lists, dictionaries)
- sequential access: Linked Lists

A Common Problem For any Container Class: Traversal

Traversal Depends on Structure

To process a container class, each item must be visited exactly once. Different structures will do this differently.

- random access indexed:

```
n = len(lst)
for i in range(n):
    print(lst[i])
```

- sequential access: Linked Lists

```
node = myLList.head
while node is not None:
    print(node.item)
    node = node.link
```

A Common Problem For any Container Class: Traversal

Iteration is Traversal Without Seeing Internal Structure

Dilema for implementing containers: traversing items is a useful operation for virtually any container, but doing so efficiently seems to require exploiting the internal structure of a container.

A Design Pattern is a strategy which occurs repeatedly in object-oriented design.

The iterator is one of the common design patterns. It provides each container class with an associated iterator class, whose behavior is simply to produce each item in some sequence.

Different designers choose slightly different APIs for iterators.

Iterators in Python

The Interface of an Iterator: next()

```
>>> from LList import *
>>> myList=[1,2,3]
>>> it=iter(myList)
>>> next(it)
1
>>> next(it)
2
>>> next(it)
3
>>> next(it)
Traceback (most recent call last):
File "<pyshell>", line 1, in <module>
it.next()
StopIteration
```

Iterators in Python

The Interface of an Iterator: the StopIteration exception

Here is how we can use this:

```
it = iter(myContainter)
while True:
    try:
        a = next(it)
    except StopIteration:
        break
    print(a)
1
2
3
```

Iterators in Python

The Interface of an Iterator: `in`

Another way:

```
for a in myList:
    print(a)
```

```
1
2
3
```

Adding an Iterator to LList

An Iterator Class for LList

```
class LListIterator(object):
 def __init__(self, head):
    self.currnode = head
 def __next__(self):
    if self.currnode is None:
       raise StopIteration
    else:
       item = self.currnode.item
       self.currnode = self.currnode.link
       return item
```

Adding an Iterator to LList

__iter__ Method for LList Class

```
def __iter__(self):
 return LListIterator(self.head)
```

Adding an Iterator to LList

Python for loop

```
>>> from LList import *
>>> nums = LList([1, 2, 3, 4])
>>> for item in nums:
        print(item)
1
2
3
4
```

Iterating With A Python Generator

A Generator Object

A Generator Object has the same interface as an iterator.

- It is used whenever a computation needs to be stopped to return a partial result.
  (Just as an iterator stops after each item when traversing a list, and returns that item.)
- It continues the computation in steps when called repeatedly.
  (Just as an iterator continues its traversal of a container, returning successive items.)

Iterating With A Python Generator

A Generator Definition

A Generator Definition combines properties of a function definition with those of the `__init__` method of a class.

- It has the format of a function definition.
- Instead of `return` it uses `yield`, to indicate where a partial result is returned and the computation frozen until the next call.
- Like a constructor (`__init__`), it returns a generator object, which behaves according to the body of the definition.

Iterating With A Python Generator

Example: Generating A Sequence of Squares

```
def squares():
    num = 1
    while True:
        yield num * num
        num += 1
>>> seq = squares()
>>> next(seq)
1
>>> next(seq)
4
>>> next(seq)
9
```

Iterating With A Python Generator

LList Iterator Reimplemented as Generator

```
class LList(object):
  ...
  def __iter__(self):
    node = self.head
    while node is not None:
      yield node.item
      node = node.link
```

Trade-offs When Storing Sequential Information

Costs and Benefits of Array Storage

- Fast random access.
- Slow insertion and deletion.
- Efficient memory usage for homogeneous data (no links to store).

Trade-offs When Storing Sequential Information

Costs and Benefits of Linked Storage

- Slow random access.
- Faster insertion and deletion.
- Requires more memory (link information). If each data item is small this may double the storage required.

In-class work

Working with LList class

Use LList.py and write a program that will do the following:
1) create a linked list for the sequence $[1, \ldots n]$, where value for n is given by the user.
2) insert three numbers (your choice), provided by the user into the list (your choice of positions, but they should be different)
3) Delete two numbers from the list (also your choice for the different positions)
4) Find the sum of all values in the linked list.

creating a Generator

Define a generator, that will be generating Fibonacci numbers.