CSC148H Week 7

February 23, 2015

Announcements

Still Being Marked/To Be Marked:

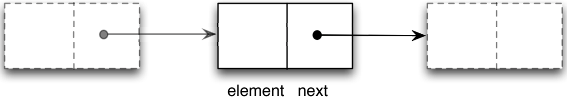
- Assignment 1
- Term test 1

Upcoming:

- Assignment 2 due March 5 at 22:00 (submit through MarkUs)
- Quiz 2 at 10:00 in class March 6

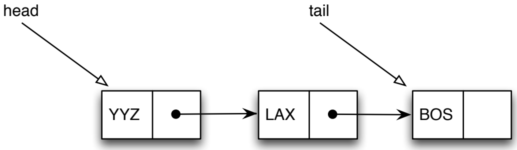
Linked Lists

- We'll implement our own list abstract data type (ADT) called a **linked list**
- Like a tree, a linked list is a recursive data structure
- A (singly) linked list contains a series of nodes
- Where each node has
  - A cargo node which contains a reference to an object that is an element of the sequence, and
  - A reference to the next node of the linked list



Linked Lists - Head and Tail

- The first and last node of a linked list are known as the **head** and **tail**.
- By starting at the head, and moving from one node to another by following each nodes `next` reference, we can reach the tail of the list.
- We can identify the tail as the node having `None` as its next reference.
- This process is commonly known as **traversing**
- Because the next reference of a node can be viewed as a **link** or **pointer** to another node, traversing a list is also known as **link hopping** or **pointer hopping**.



Linked List Methods

We'll work through writing some linked list methods.

- `__init__`: create our linked list
- `__repr__`: represent our linked list
- `prepend`: add a new object to the start of the list
- `remove_first`: remove the head object
- `__contains__`: check if the linked list contains a specified object, often refereed to as a "search" method

Linked List Methods

```
class LinkedList:
    '''Linked list class'''

    def __init__(self: 'LinkedList', head: object=None,
                 rest: 'LinkedList'=None) -> None:
        '''Create a new LinkedList.
        head - first element of linked list
        rest - linked list of remaining elements'''

    def __repr__(self: 'LinkedList') -> str:
        '''Return str representation of LinkedList'''

    def prepend(self: 'LinkedList', newhead: object) -> None:
        '''Add new head to the front of the linked list.'''

    def remove_first(self: 'LinkedList') -> None:
        '''Remove head from LinkedList'''

    def __contains__(self: 'LinkedList', value: object) -> bool:
        '''Return True iff LinkedList contains value'''
```

Linked List Methods, Constructor

- When a linked list is first initialized it has no nodes, so the head is set to `None`.
- We can also by default set the rest of the linked list to `None`.

```
class LinkedList:
    '''Linked list class'''

    def __init__(self: 'LinkedList', head: object=None,
                 rest: 'LinkedList'=None) -> None:
        '''Create a new LinkedList.
        head - first element of linked list
        rest - linked list of remaining elements
        '''
        # a linked list is empty if and only if it has no head
        self.empty = head is None
        if not self.empty:
            self.head = head
            if rest is None:
                self.rest = LinkedList()
            else:
                self.rest = rest
```

Linked List Methods, Representation Method

```
class LinkedList:
    '''Linked list class'''

  def __repr__(self: 'LinkedList') -> str:
      '''Return str representation of LinkedList'''
      if self.empty:
          return 'LinkedList()'
      else:
          return 'LinkedList({}, {})'.format(
            repr(self.head), repr(self.rest))
```

Linked List Methods, Contains Method

```python
class LinkedList:
    '''Linked list class'''

    def __contains__(self: 'LinkedList', newhead: object) -> None:
        '''Return True iff LinkedList contains value'''
        if self.empty:
            return False

        # list has at least one element
        return self.head == value or rest.__contains__(value)
```

Writing Linked List Methods

- We've covered the `__init__`, `__repr__` and `contains` methods
- Now how do we address `prepend` and `remove_first`?

Writing Linked List Methods

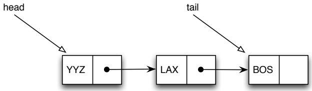
It's often useful to think in terms of three specific cases:

- What to do if the linked list is empty
- What to do if it has one node
- What to do if it has more than one node

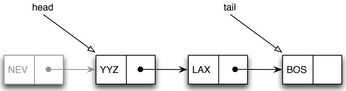
You may not have to write three separate cases for each method, but always test that these three cases are covered!

prepend

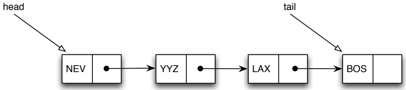
- You can insert a node anywhere in the list, but the *simplest* way is to insert a new node at the **head**.
- By doing so, we need to point the new node at the old head.



(a) Original List



(b) List befre inserting NEV



(c) List after inserting NEV

Figure : Prepending a node

prepend...

What to do if the linked list is empty?

- Set `head` to the new object
- Set `rest` to the empty linked list

What to do if the linked list is not empty?

- Make a new linked list `n` that contains the `head` and `rest` of the original linked list
- Set `head` to the new object
- Set `rest` to `n`

prepend

```
class LinkedList:
    '''Linked list class'''

    def prepend(self: 'LinkedList', newhead: object) -> None:
        '''Add new head to the front of the linked list.'''
        if not self.empty:
            temp = LinkedList(self.head, self.rest)
        else:    # non-empty
            temp = LinkedList()

        self.head = newhead
        self.rest = temp
        self.empty = False
```

remove first...

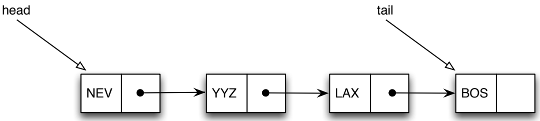
What to do if the linked list is empty?

- Prevent this from happening by raising an error

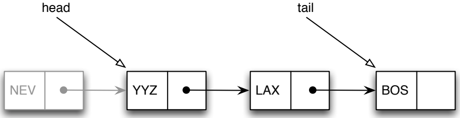
What to do if the linked list is not empty?

- Essentially the reverse of prepending an object (inserting a new element at the head.)

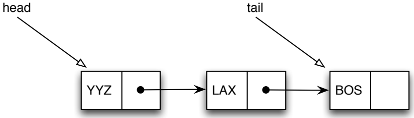
remove first



(a) Original List



(b) List before removing first and "linking out" the old head



(c) List after removing first

Figure : Removing first node

remove first

```python
class LinkedList:
    '''Linked list class'''

    def remove_first(self: 'LinkedList') -> None:
        '''Remove head from LinkedList.'''
        if self.empty:
            raise Exception("Can't remove head from empty list")
        if self.rest.empty:
            self.empty = True
            del self.head
            del self.rest
        else: # has more than one element
            self.head = self.rest.head
            self.rest = self.rest.rest
```

append...

What to do if the linked list is empty?

- Then you can call on the `prepend` method to add the to the new object to the front
- Set `rest` to the empty linked list

What to do if the linked list is not empty?

- Need to *append* to the `rest`

append

```
class LinkedList:
    '''Linked list class'''

    def append(self: 'LinkedList', newlast: object) -> None:
        '''Add newlast to end of LinkedList'''
        # must get to end of list before appending
        if self.empty:
           self.prepend(newlast)
        else:
           self.rest.append(newlast)
```

Implementating Linked Lists with other ADTs

- Stacks: last in, first out
- Queues: first in, first out
- Recall: a node consists of an *element* and a reference to the *next* node

Implementating Linked Lists with other ADTs

```python
class LinkedList:
    '''Implementation of a singly linked list'''

    # Nested node class
    class _Node:
    '''A nonpublic singly linked list class'''

        def __init__(self, element, next):
            self._element = element
            self._next = next # reference to the next node
```

Stack Implementation of a Singly Linked Lists

- Last in, first out
- Methods: `__init__`, `size`, `is_empty`, `pop`, `push`, `top`

Stack Implementation of a Singly Linked Lists

- Let's start with the constructor

```
class LinkedList:
    '''Implementation of a singly linked list'''

    def __init__(self):
        '''Create an empty stack'''
        self._head = None
        self._size = 0
```

Stack Implementation of a Singly Linked Lists

```python
class LinkedList:
    '''Implementation of a singly linked list'''

    def __init__(self):
        '''Create an empty stack'''
        self._head = None
        self._size = 0

    def _len(self):
        '''Return the number of elements in the stack'''
        return self._size

    def is_empty(self):
        '''Return True if the stack is empty'''
        return self._size == 0
```

Stack Implementation of a Singly Linked Lists

```
class LinkedList:
    '''Implementation of a singly linked list'''

    # Nested node class
    class _Node:
    '''A nonpublic singly linked list class'''

        def __init__(self, element, next):
            self._element = element
            self._next = next # reference to the next node

    # Stack methods
     def push(self, e):
         '''Add element e to the top of the stack'''
         self._head = self._Node(e, self._head)
         self._size += 1
```

Stack Implementation of a Singly Linked Lists

```
class LinkedList:
    '''Implementation of a singly linked list'''

    # Nested node class
    class _Node:
    '''A nonpublic singly linked list class'''

        def __init__(self, element, next):
            self._element = element
            self._next = next # reference to the next node

    # Stack methods
    def top(self):
        '''Return (but do not remove) the element at the top of the stack.
        Raise Empty exception if the stack is empty.'''
        if self.is_empty():
            raise Empty('Stack is empty')
        return self._head._element
```

Stack Implementation of a Singly Linked Lists

```
class LinkedList:
    '''Implementation of a singly linked list'''

    # Nested node class
    class _Node:
    '''A nonpublic singly linked list class'''

        def __init__(self, element, next):
            self._element = element
            self._next = next # reference to the next node

    def pop(self):
        '''Remove and return the top element of the queue.
        Raise Empty exception if the queue is empty.'''
        if self.is_empty():
            raise Empty('Stack is empty')
        remove = self._head._element
        self._head = self._head._next
        self._size -= 1
        return remove
```

Queue Implementation of a Singly Linked Lists

- First in, first out
- Methods: `__init__`, `size`, `is_empty`, `dequeue`, `enqueue`, `front`

Queue Implementation of a Singly Linked Lists

- Let's begin with the constructor again
- Notice the new instance variable, `self._tail`

```
class LinkedList:
    '''Implementation of a singly linked list'''

    # Queue methods
    def __init__(self):
        '''Create an empty stack'''
        self._head = None
        self._tail= None
        self._size = 0
```

Queue Implementation of a Singly Linked Lists

```
class LinkedList:
    '''Implementation of a singly linked list'''

    def _len(self):
        '''Return the number of elements in the stack'''
        return self._size

    def is_empty(self):
        '''Return True if the stack is empty'''
        return self._size == 0
```

Queue Implementation of a Singly Linked Lists

```
class LinkedList:
    '''Implementation of a singly linked list'''

    # Nested node class
    class _Node:
    '''A nonpublic singly linked list class'''

        def __init__(self, element, next):
            self._element = element
            self._next = next # reference to the next node

    # Queue methods
    def first(self):
        '''Return (but do not remove) the element at the front of the queue.
        Raise Empty exception if the queue is empty.'''
        if self.is_empty():
            raise Empty('Stack is empty')
        return self._head._element
```

Queue Implementation of a Singly Linked Lists

```
class LinkedList:

    # Nested node class
    class _Node:
    '''A nonpublic singly linked list class'''

        def __init__(self, element, next):
            self._element = element
            self._next = next # reference to the next node

    # Queue methods
    def dequeue(self):
        '''Remove and return the first element of the queue.
        Raise Empty exception if the queue is empty.'''
        if self.is_empty():
            raise Empty('Stack is empty')
        remove = self._head._element
        self._head = self._head._next
        self._size -= 1

        if self.is_empty():
            self._tail = None
        return remove
```

Queue Implementation of a Singly Linked Lists

```
class LinkedList:

    # Nested node class
    class _Node:
    '''A nonpublic singly linked list class'''

        def __init__(self, element, next):
            self._element = element
            self._next = next # reference to the next node

    # Queue methods
    def enqueue(self, e):
        '''Add element e to the back of the queue'''
        newest = self._Node(e, None)
        if self.is_empty():
            self._head = newest
        else:
            self._tail._next = newest
        self._tail = newest
        self._size += 1
```

Iterables

- You can use "for ... in" to iterate over lists, dictionaries, strings, files, etc
- All the values are stored in memory

```
x_iter = [1, 2, 3]

def example(x);
    for i in x:
        print(i)

>>> example(x_iter)
>>> 1
>>> 2
>>> 3
```

Generators

- Generators are also iterables
  - Generator functions create **generator iterators**. That's the last time you'll see the term **generator iterator**, though, since they're almost always referred to as "generators".
- A generator "generates" values but you can only iterate over them **once**
- A generator function is defined like a normal function, but whenever it needs to generate a value, it does so with the **yield** keyword rather than **return**

Generator Example

```
x_gen = (x*x for x in range(3))

def example(x):
    for i in x:
        yield(i)
>>> example(x_gen)
0
1
4
>>> example(x_gen)
>>>
```

Recall: you can not perform `for i in x_gen` a second time since generators can only be used once

Yield

- **Yield** is a keyword that is used like **return**, except the function will return a **generator**.

```
def gen_yield_ex():
     x = range (3)
     for i in x:
         yield i*i

>>> my_generator = gen_yield_ex()      # create a generator
<generator object gen_yield_ex at 0x105a319d8>
>>> for i in my_generator:
     print(i)

0
1
4
```

if/else ternary expression

The **ternary** operator is a way to concisely say:
If test, then a, else b,
with the value of the statement being the value of a or b.

In python, we use the form:

```
a if test else b
```

In other languages:

Language	**Form**
C, C++, Java, Perl, PHP, Ruby	test ? a : b
Python	a if test else b

if/else ternary expression

```
if b:
  result = x
else:
  result = y
```

is the same as the expression

```
result = x if b else y
```

```
>>> lst = [1, 2, 3, 11, 12, 13]
>>> ['even' if e % 2 == 0 else 'odd' for e in lst]
['odd', 'even', 'odd', 'odd', 'even', 'odd']
```