Lecture 17: Mutable Linked Lists

Brian Hou
July 20, 2016

Announcements

- Homework 6 is due today at 11:59pm
- Project 3 is due 7/26 at 11:59pm
  - Earn 1 EC point for completing it by 7/25
- Quiz 5 tomorrow at the beginning of lecture
  - May cover mutability, OOP I (Monday)
- Project 1 revisions due 7/27 at 11:59pm

Roadmap

Introduction
Functions
Data
Mutability
Objects
Interpretation
Paradigms
Applications

- This week (Objects), the goals are:
  - To learn the paradigm of *object-oriented programming*
  - To study applications of, and problems that be solved using, OOP

Practical OOP

Checking Types (and Accounts) (demo)

- We often check the type of an object to determine what operations it permits
- The **type** built-in function returns the class that its argument is an instance of
- The **isinstance** built-in function returns whether its first argument (object) is an instance of the second argument (class) *or a subclass*
- isinstance(obj, cls) is usually preferred over type(obj) == cls

```
>>> a = Account('Brian')
>>> ch = CheckingAccount('Brian')

>>> type(a) == Account
True
>>> type(ch) == Account
False
>>> type(ch) == CheckingAccount
True

>>> isinstance(a, Account)
True
>>> isinstance(ch, Account)
True
>>> isinstance(a, CheckingAccount)
False
>>> isinstance(ch, CheckingAccount)
True
```

Python's Magic Methods (demo)

- How does the Python interpreter display values?
  - First, it evaluates the expression to some value
  - Then, it calls repr on that value and prints that string
- How do magic methods work?
- Are integers objects too? (Yep!)
- Are ____ objects too? (Yep!)

```
>>> x = Rational(3, 5)
>>> y = Rational(1, 3)
>>> y
Rational(1, 3)
>>> repr(y)
'Rational(1, 3)'
>>> print(repr(y))
Rational(1, 3)
>>> x * y
Rational(1, 5)
>>> x.__mul__(y)
Rational(1, 5)
```

Linked Lists

The Link Class

```
empty = 'X'

def link(first, rest=empty):
    return [first, rest]

def first(lnk):
    return lnk[0]

def rest(lnk):
    return lnk[1]


>>> link_adt = link(1,
                    link(2,
                    link(3)))
>>> first(rest(link_adt))
2
```

```
class Link:
    empty = ()

    def __init__(self, first,
                 rest=empty):
        self.first = first
        self.rest = rest


>>> link_cls = Link(1,
                    Link(2,
                    Link(3)))
>>> link_cls.rest.first
2
```

Mutable Linked Lists (demo)

• Instances of user-defined classes are mutable *by default*

```
class Link:
    empty = ()

    def __init__(self, first, rest=empty):
        self.first = first
        self.rest = rest

    def __repr__(self):
        if self.rest is Link.empty:
            return 'Link({0})'.format(
                self.first)
        else:
            return 'Link({0}, {1})'.format(
                self.first, repr(self.rest))
```

Linked Lists are Sequences (demo)

```
class Link:
    empty = ()

    ...

    def __getitem__(self, i):
        if i == 0:
            return self.first
        elif self.rest is Link.empty:
            raise IndexError('...')
        else:
            return self.rest[i - 1]

    def __len__(self):
        return 1 + len(self.rest)
```

Sneaky recursive call: equivalent to self.rest.__getitem__(i-1)

Another sneaky recursive call: equivalent to self.rest.__len__()

Where's the base case??

The __setitem__ Magic Method (demo)

```
>>> s = Link(1, Link(2, Link(3)))
>>> s[1] = 3
>>> s
Link(1, Link(3, Link(3)))


class Link:
    ...
    def __setitem__(self, i, val):
        if i == 0:
            self.first = val
        elif self.rest is Link.empty:
            raise IndexError('...')
        else:
            self.rest[i - 1] = val
```

Mutating Map (demo)

```
>>> s = Link(1, Link(2, Link(3)))
>>> s.map(lambda x: x * x)
>>> s
Link(1, Link(4, Link(9)))


class Link:
    ...
    def map(self, f):
        for i in range(len(self)):
            self[i] = f(self[i])
```

Runtime?

Mutating Map

```
>>> s = Link(1, Link(2, Link(3)))
>>> s.map(lambda x: x * x)
>>> s
Link(1, Link(4, Link(9)))

class Link:
    ...
    def __getitem__(self, i):
        if i == 0:
            return self.first
        else:
            return self.rest[i - 1]

    def map(self, f):
        for i in range(len(self)):
            self[i] = f(self[i])
```

```
self[0] = f(self[0])

self[1] = f(self[1])

self[2] = f(self[2])

...

self[n-1] = f(self[n-1])
```

$\Theta(n^2)$

Mutating Map (Improved) (demo)

```
>>> s = Link(1, Link(2, Link(3)))
>>> s.map(lambda x: x * x)
>>> s
Link(1, Link(4, Link(9)))

class Link:
    ...
    def map(self, f):
        self.first = f(self.first)
        if self.rest is not Link.empty:
            self.rest.map(f)
```

Runtime?

$\Theta(n)$

contains and in (demo)

```
class Link:
    ...
    def __contains__(self, e):
        return self.first == e or e in self.rest
```

```
>>> s = Link(1, Link(2, Link(3)))
>>> 2 in s
True
>>> 4 in s
False
```

Break!

Environments

Environment Frames

- An environment is a sequence of frames
  - Each frame has some data (bindings) and a parent, which points to another frame
- A linked list is a sequence of values
  - Each link has some data (first) and a rest, which points to another link
- An environment is just a special case of a linked list!

Environment Frames (demo)

- An environment is a sequence of frames
  - Each frame has some data (**bindings**) and a **parent**, which points to another **frame**
- A linked list is a sequence of values
  - Each link has some data (**first**) and a **rest**, which points to another **link**
- An environment is just a special case of a linked list!

The Call Stack (demo)

- A *stack* is a data structure that permits two operations
  - Add to the top of a stack ("push")
  - Remove from the top of a stack ("pop")

- Two new Link operations required: insert_front and remove_front
- A *call stack* keeps track of frames that are currently open
  - Calling a function adds a new frame to the stack
  - Returning from a function removes that frame from the stack
  - The current frame is always on the top of the stack

Python

Brython

- What if we could have Python functions use the environment frames and the call stack that we just defined?
- Two important parts:
  - What should happen when **defining** a Brython function?
  - What should happen when **calling** a Brython function?

A different Brython: http://brython.info/

Function Definitions

- What happens in a function definition?
  - Determine the current frame of execution: this is the function's parent frame
  - Bind the function name to the function value

Function Calls (demo)

- What happens in a function call?
  - Create a brand new call frame (using the function parent as the parent of that frame) and insert it into the stack
  - Bind function's parameters to arguments
  - Execute the function in the environment of the call frame
    - Remember: the current frame is at the top of the stack
  - After executing the function, remove the frame from the stack

Summary

- Linked lists are one way to store sequential data
- An object-based implementation of the linked list abstraction allows for easy mutability
  - No more crazy nonlocal stuff!
- Implementing magic methods lets us hook into convenient Python syntax and built-in functions
- Linked lists can be used to implement some of the core ideas of this course!