Monday, October 14
Announcements

• Homework 5 is due Tuesday 10/15 @ 11:59pm
• Project 3 is due Thursday 10/24 @ 11:59pm
• Midterm 2 is on Monday 10/28 7pm–9pm
Special Method Names
Special Method Names in Python

Certain names are special (or "magic") because they have built-in behavior.

These names always start and end with two underscores.

- `__init__`: Method invoked automatically when an object is constructed.
- `__len__`: Method invoked by the built-in `len` function.
- `__getitem__`: Method invoked for element selection: `sequence[index]`
- `__repr__`: Method invoked to display an object as a string.

```python
>>> s = (3, 4, 5)
>>> len(s)
3
>>> s[2]
5
>>> s
(3, 4, 5)

>>> s = (3, 4, 5)
>>> s.__len__()
3
>>> s.__getitem__(2)
5
>>> print(s.__repr__())
(3, 4, 5)
```
Recursive List Class
Closure Property of Data

A tuple can contain another tuple as an element.

Pairs are sufficient to represent sequences of arbitrary length.

Recursive list representation of the sequence 1, 2, 3, 4:

```
→ 1 ← → 2 ← → 3 ← → 4
```

Recursive lists are recursive: the rest of the list is a list.

Now, we can implement the same behavior using a class called Rlist:

Abstract data type (old): \(\text{rlist}(1, \text{rlist}(2, \text{rlist}(3, \text{rlist}(4, \text{empty\_rlist}))))\)

Rlist class (new): \(\text{Rlist}(1, \text{Rlist}(2, \text{Rlist}(3, \text{Rlist}(4))))\)
Recursive List Class

class Rlist:

class EmptyList:
    def __len__(self):
        return 0

empty = EmptyList()

def __init__(self, first, rest=empty):
    assert type(rest) is Rlist or rest is Rlist.empty
    self.first = first
    self.rest = rest

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

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

Methods can be recursive too!

There's the base case!

Calls this method with a special name

This element selection syntax

Yes, this call is recursive
Recursive List Processing
Recursive Operations on Recursive Lists

Recursive list processing almost always involves a recursive call on the rest of the list.

```python
>>> s = Rlist(1, Rlist(2, Rlist(3)))

>>> s.rest
Rlist(2, Rlist(3))

>>> extend_rlist(s.rest, s)
Rlist(2, Rlist(3, Rlist(1, Rlist(2, Rlist(3))))))

def extend_rlist(s1, s2):
    if s1 is Rlist.empty:
        return s2
    else:
        return Rlist(s1.first, extend_rlist(s1.rest, s2))
```
We want operations on all elements of a list, not just an element at a time.

\[
\begin{align*}
double_rlist(s) & \quad \text{Double } s.first, \text{ then } double_rlist(s.rest) \\
map_rlist(s, fn) & \quad \text{Apply } fn \text{ to } s.first, \text{ then } map_rlist(s.rest, fn) \\
filter_rlist(s, fn) & \quad \text{Either keep } s.first \text{ or not, then } filter_rlist(s.rest, fn)
\end{align*}
\]

In all of these functions, the base case is the empty list.

(Demo)
Trees
Tree Structured Data

Nested sequences form hierarchical structures: tree-structured data

\(((1, 2), (3, 4), 5)\)

*In every tree, a vast forest*
Recursive Tree Processing

Tree operations typically make recursive calls on branches.

\[
\text{count}\_\text{leaves}(t) = \begin{cases} 
1 & \text{if } t \text{ is a leaf, otherwise sum } \text{count}\_\text{leaves}(\text{branch}) \\
\end{cases}
\]

\[
\text{map}\_\text{tree}(t, \text{fn}) = \begin{cases} 
\text{fn}(t) & \text{if } t \text{ is a leaf, otherwise combine } \text{map}\_\text{tree}(\text{branch, fn}) \\
\end{cases}
\]

In these functions, the base case is a leaf.

(Demo)
Trees with Internal Entries
Trees with Internal Entries

Trees can have values at their roots as well as their leaves.
Trees can have values at their roots as well as their leaves.

```python
class Tree:
    def __init__(self, entry, left=None, right=None):
        self.entry = entry
        self.left = left
        self.right = right

def fib_tree(n):
    if n == 1:
        return Tree(0)
    if n == 2:
        return Tree(1)
    left = fib_tree(n-2)
    right = fib_tree(n-1)
    return Tree(left.entry + right.entry, left, right)
```
(Demo)
Trees with Internal Entries

(Demo)
Memoization
Memoization

**Idea:** Remember the results that have been computed before

def memo(f):
    cache = {}
    def memoized(n):
        if n not in cache:
            cache[n] = f(n)
        return cache[n]
    return memoized

Keys are arguments that map to return values

Same behavior as f, if f is a pure function

(Demo)
Memoized Tree Recursion

$\text{fib\_tree}(35)$

Distinct trees with memoization: $35$
Distinct trees without memoization: $18,454,929$