Who am I? What am I doing here?

- First two weeks of class (Chapter 1):
  - FUNCTIONS
  - Computational processes, role of functions
- Past 2 weeks of class (Chapter 2):
  - DATA
  - Real-world phenomena are complex – Mining Twitter data!
- Today and the next 2 weeks (Chapter 3):
  - PROGRAMS and their INTERPRETATION
Next 2 weeks

• A Python program is just a collection of text
• This text only has meaning through interpretation
• Programming languages like Python are useful because we can define an interpreter, a program that carries out Python’s evaluation and execution procedures

An interpreter, which determines the meaning of expressions, is really just another program.

We are not only users of languages designed by others, we are designers of languages.
Interpreters...

• Writing our own interpreters will be exciting! We will cover this next week.
• First, though, we need to learn a few background tools and techniques
• Today’s lecture focuses on several ideas that will help us later create our interpreters
Announcements

• Do homework!
• Potluck next Friday, July 26 6-8pm, in the Woz Lounge (same place as last time)
  • Don’t make any other plans for Friday!
  • Bring some food, enjoy other people’s food!
  • Come mingle with fellow students and the teaching staff! No project or midterm due that week!
Speeding up computation

```python
def fib_iter(n):
    if n == 0:
        return 0
    fib_n, fib_n_1 = 1, 0
    k = 1
    while k < n:
        fib_n, fib_n_1 = fib_n_1 + fib_n, fib_n
        k += 1
    return fib_n

def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    return fib(n - 1) + fib(n - 2)
```
Speeding up computation

- fib_iter seems to be much faster when the input is large!
- Why?
  - The recursive function calls fib many times (about 2 fib recursive calls are generated for each call to fib). When you have a LOT of function calls, then computation will take much longer (think: a new frame has to be created for each call)
- How can we speed up the recursive version?
Memoization

Tree recursive functions often compute the same thing many times

Idea: Remember the results that have been computed before

```python
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
Memoized Tree Recursion

Calls to `fib` with memoization: 35
Calls to `fib` without memoization: 18,454,929
When does memo speed computation up?

• memo_factorial(5) – not sped up the first time we call it
• What if we called memo_factorial(5) again?
• Memoization speeds computation up when the function is called more than once, perhaps through recursion; otherwise, no effect other than minor assignments to the memo dictionary
• The memoized version of fib computes more efficiently
• We will discuss a more precise definition for “computes more efficiency” tomorrow
Closure Property of Data

A tuple can contain another tuple as an element.

Pairs are sufficient to represent sequences.

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

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

Nested pairs (old):   (1, (2, (3, (4, None)))))
Rlist class (new):   Rlist(1, Rlist(2, Rlist(3, Rlist(4)))))
```
Recursive List Class

Methods can be recursive as well!

class Rlist(object):

class EmptyList(object):
    def __len__(self):
        return 0

empty = EmptyList()

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

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

def __getitem__(self, i):
    if i == 0:
        return self.first
    return self.rest[i - 1]
Recursive Operations on Rlists

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

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

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

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

```python
def extend_rlist(s1, s2):
    if s1 is Rlist.empty:
        return s2
    return Rlist(s1.first, extend_rlist(s1.rest, s2))
```
Map and Filter on Rlists

We want operations on a whole list, not an element at a time.

```python
def map_rlist(s, fn):
    if s is Rlist.empty:
        return s
    return Rlist(fn(s.first), map_rlist(s.rest, fn))

def filter_rlist(s, fn):
    if s is Rlist.empty:
        return s
    rest = filter_rlist(s.rest, fn)
    if fn(s.first):
        return Rlist(s.first, rest)
    return rest
```
Break!
Sets

A built-in Python container type

- Set literals are enclosed in braces
- Duplicate elements are removed on construction
- Sets are unordered, just like dictionary entries

```python
>>> s = {3, 2, 1, 4, 4}
>>> s
{1, 2, 3, 4}
>>> 3 in s
True
>>> len(s)
4
>>> s.union({1, 5})
{1, 2, 3, 4, 5}
>>> s.intersection({6, 5, 4, 3})
{3, 4}
```
Implementing Sets

What we should be able to do with a set:

- Membership testing: Is a value an element of a set?
- Union: Return a set with all elements in set1 or set2
- Intersection: Return a set with any elements in set1 and set2
- Adjunction: Return a set with all elements in s and a value v

<table>
<thead>
<tr>
<th>Union</th>
<th>Intersection</th>
<th>Adjunction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 3 4 2 5</td>
<td>1 3 2 5 4</td>
<td>1 3 1 2 4</td>
</tr>
</tbody>
</table>
Sets as Unordered Sequences

**Proposal 1:** A set is represented by a recursive list that contains no duplicate items

```python
def empty(s):
    return s is Rlist.empty

def set_contains(s, v):
    if empty(s):
        return False
    elif s.first == v:
        return True
    return set_contains(s.rest, v)
```
Sets as Unordered Sequences

def adjoin_set(s, v):
    if set_contains(s, v):
        return s
    return Rlist(v, s)

def intersect_set(set1, set2):
    f = lambda v: set_contains(set2, v)
    return filter_rlist(set1, f)

def union_set(set1, set2):
    f = lambda v: not set_contains(set2, v)
    set1_not_set2 = filter_rlist(set1, f)
    return extend_rlist(set1_not_set2, set2)

We will talk about how “efficient” these operations are next class!
Sets as Ordered Sequences

**Proposal 2**: A set is represented by a recursive list with unique elements ordered from least to greatest

def set_contains2(s, v):
    if empty(s) or s.first > v:
        return False
    elif s.first == v:
        return True
    return set_contains(s.rest, v)
Set Intersection Using Ordered Sequences

This algorithm assumes that elements are in order.

```python
def intersect_set2(set1, set2):
    if empty(set1) or empty(set2):
        return Rlist.empty
    e1, e2 = set1.first, set2.first
    if e1 == e2:
        rest = intersect_set2(set1.rest, set2.rest)
        return Rlist(e1, rest)
    elif e1 < e2:
        return intersect_set2(set1.rest, set2)
    elif e2 < e1:
        return intersect_set2(set1, set2.rest)
```
Tree Structured Data

Nested Sequences are Hierarchical Structures.

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

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

Tree operations typically make recursive calls on branches

def count_leaves(tree):
    if type(tree) != tuple:
        return 1
    return sum(map(count_leaves, tree))

def map_tree(tree, fn):
    if type(tree) != tuple:
        return fn(tree)
    return tuple(map_tree(branch, fn)
                  for branch in tree)
Trees with Internal Node Values

Trees can have values at internal nodes as well as their leaves.

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

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

Trees can have values at internal nodes as well as their leaves.

```
fib(5)
  /     /
fib(3)  fib(4)
  /     /     /
fib(1) fib(2) fib(3)
 /     /     /
1  fib(0) fib(1)
     /     /
     0 1
```

```
fib(5)
  /     /
fib(3)  fib(4)
  /     /     /
fib(1) fib(2) fib(3)
 /     /     /
1  fib(0) fib(1)
     /     /
     0 1
```