Announcements
Box-and-Pointer Notation
The Closure Property of Data Types

• A method for combining data values satisfies the closure property if:
  
  The result of combination can itself be combined using the same method

• Closure is powerful because it permits us to create hierarchical structures

• Hierarchical structures are made up of parts, which themselves are made up of parts, and so on

Lists can contain lists as elements (in addition to anything else)
Box-and-Pointer Notation in Environment Diagrams

Lists are represented as a row of index-labeled adjacent boxes, one per element. Each box either contains a primitive value or points to a compound value.

pair = [1, 2]
Box-and-Pointer Notation in Environment Diagrams

Lists are represented as a row of index-labeled adjacent boxes, one per element. Each box either contains a primitive value or points to a compound value.

```
1 pair = [1, 2]
2
3 nested_list = [[1, 2], [],
4     [3, False, None],
5     [4, lambda: 5]]
```
Slicing

(Demo)
Slicing Creates New Values

```python
digits = [1, 8, 2, 8]
start = digits[:1]
middle = digits[1:3]
end = digits[2:]
full = digits[:]
```

Interactive Diagram
Processing Container Values
Sequence Aggregation

Several built-in functions take iterable arguments and aggregate them into a value

- **sum(iterable[, start]) -> value**
  
  Return the sum of an iterable of numbers (NOT strings) plus the value of parameter 'start' (which defaults to 0). When the iterable is empty, return start.

- **max(iterable[, key=func]) -> value**
  
  With a single iterable argument, return its largest item. With two or more arguments, return the largest argument.

- **all(iterable) -> bool**
  
  Return True if bool(x) is True for all values x in the iterable. If the iterable is empty, return True.
Trees
Tree Abstraction

**Recursive description** *(wooden trees):*
A **tree** has a **root** and a list of **branches**
Each branch is a **tree**
A tree with zero branches is called a **leaf**

**Relative description** *(family trees):*
Each location in a tree is called a **node**
Each **node** has a **label value**
One node can be the **parent/child** of another

*People often refer to values by their locations: "each parent is the sum of its children"*
Implementing the Tree Abstraction

```python
def tree(label, branches=[]):
    return [label] + branches

def label(tree):
    return tree[0]

def branches(tree):
    return tree[1:]

>>> tree(3, [tree(1),
...    tree(2, [tree(1),
...      tree(1)])])
[3, [1], [2, [1], [1]]]
```

- A tree has a label value and a list of branches
Implementing the Tree Abstraction

```python
def tree(label, branches=[]):
    for branch in branches:
        assert is_tree(branch)
    return [label] + list(branches)

def label(tree):
    return tree[0]

def branches(tree):
    return tree[1:]

def is_tree(tree):
    if type(tree) != list or len(tree) < 1:
        return False
    for branch in branches(tree):
        if not is_tree(branch):
            return False
    return True

def is_leaf(tree):
    return not branches(tree)  # (Demo)

>>> tree(3, [tree(1), ...
      ... tree(2, [tree(1), ...
            ... [3, [1], [2, [1], [1]]])])
```

- A tree has a label value and a list of branches

```
  3
 /   \\   \\
 1    2
   /\
  1  1
```

Verifies the tree definition

- Verifies that tree is bound to a list
- Creates a list from a sequence of branches

Verifies the tree definition
Tree Processing

(Demo)
Tree Processing Uses Recursion

Processing a leaf is often the base case of a tree processing function.

The recursive case typically makes a recursive call on each branch, then aggregates:

```python
def count_leaves(t):
    '''Count the leaves of a tree.'''
    if is_leaf(t):
        return 1
    else:
        branch_counts = [count_leaves(b) for b in branches(t)]
        return sum(branch_counts)

(Demo)```
Discussion Question

Implement `leaves`, which returns a list of the leaf labels of a tree.

*Hint:* If you sum a list of lists, you get a list containing the elements of those lists.

```python
>>> sum([[1], [2, 3], [4]], [])
[1, 2, 3, 4]
>>> sum([[1], []])
[1]
>>> sum([[1], [2]], [])
[1, 2]
```

```python
def leaves(tree):
    """Return a list containing the leaves of tree."
    if is_leaf(tree):
        return [label(tree)]
    else:
        return sum([leaves(b) for b in branches(tree)], [])
```

```text
branches(tree)
leaves(tree)
[branches(b) for b in branches(tree)]
[leaves(b) for b in branches(tree)]
```
Creating Trees

A function that creates a tree from another tree is typically also recursive

```python
def increment_leaves(t):
    """Return a tree like t but with leaf values incremented.""
    if is_leaf(t):
        return tree(label(t) + 1)
    else:
        bs = [increment_leaves(b) for b in branches(t)]
        return tree(label(t), bs)

def increment(t):
    """Return a tree like t but with all node values incremented.""
    return tree(label(t) + 1, [increment(b) for b in branches(t)])
```
Example: Printing Trees

(Demo)