Trees
Announcements
Data Abstraction
Data Abstraction

A small set of functions enforce an abstraction barrier between *representation* and *use*

- How data are represented (as some underlying list, dictionary, etc.)
- How data are manipulated (as whole values with named parts)

E.g., refer to the parts of a line (affine function) called \( f \):

- \texttt{slope(f)} instead of \texttt{f[0]} or \texttt{f['slope']}
- \texttt{y\_intercept(f)} instead of \texttt{f[1]} or \texttt{f['y\_intercept']}

Why? Code becomes easier to read & revise; later you could represent a line \( f \) as a Python function or as two points instead of a \([\text{slope}, \text{intercept}]\) pair without changing code that uses lines.
Trees
Tree Abstraction

Recursive description (wooden trees):
A tree has a root label and a list of branches. Each branch is a tree. A tree with zero branches is called a leaf. A tree starts at the root.

Relative description (family trees):
Each location in a tree is called a node. Each node has a label that can be any value. One node can be the parent/child of another. The top node is the root node.

People often refer to labels by their locations: "each parent is the sum of its children"
Using the Tree Abstraction

For a tree $t$, you can **only:**

- Get the label for the root of the tree: $\text{label}(t)$
- Get the list of branches for the tree: $\text{branches}(t)$
- Get the branch at index $i$, which is a tree: $\text{branches}(t)[i]$
- Determine whether the tree is a leaf: $\text{is_leaf}(t)$
- Treat $t$ as a value: $\text{return } t, f(t), [t], s = t$, etc.

(Demo)
Implementing the Tree Abstraction

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

def label(tree):
    return tree[0]

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

- A tree has a root **label** and a list of **branches**
- Each branch is a tree

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

---

- A **tree** has a root **label** and a list of **branches**.
- Each branch is a tree.

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

```python
def is_leaf(tree):
    return not branches(tree)  
```

(Demo)
Tree Processing
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)
```
Writing Recursive Functions

Make sure you can answer the following before you start writing code:

• What recursive calls will you make?

• What type of values do they return?

• What do the possible return values mean?

• How can you use those return values to complete your implementation?
**Example: Largest Label**

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 largest_label(t):
    """Return the largest label in tree t."""
    if is_leaf(t):
        return ____________
    else:
        return ____________
```

```python
    return max([largest_label(b) for b in branches(t)] + [label(t)]
```
Example: Largest Label

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 above_root(t):
    
    def process(u):
        if ____________:
            print(_________)

        for b in branches(u):
            process(b)

    process(t)
```