Trees

Announcements

Dictionaries

```
{'Dem': 0}
```

Dictionary Comprehensions

```
{<key exp>: <value exp> for <name> in <iter exp> if <filter exp>}
```

```
Short version: {<key exp>: <value exp> for <name> in <iter exp>}
```

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)

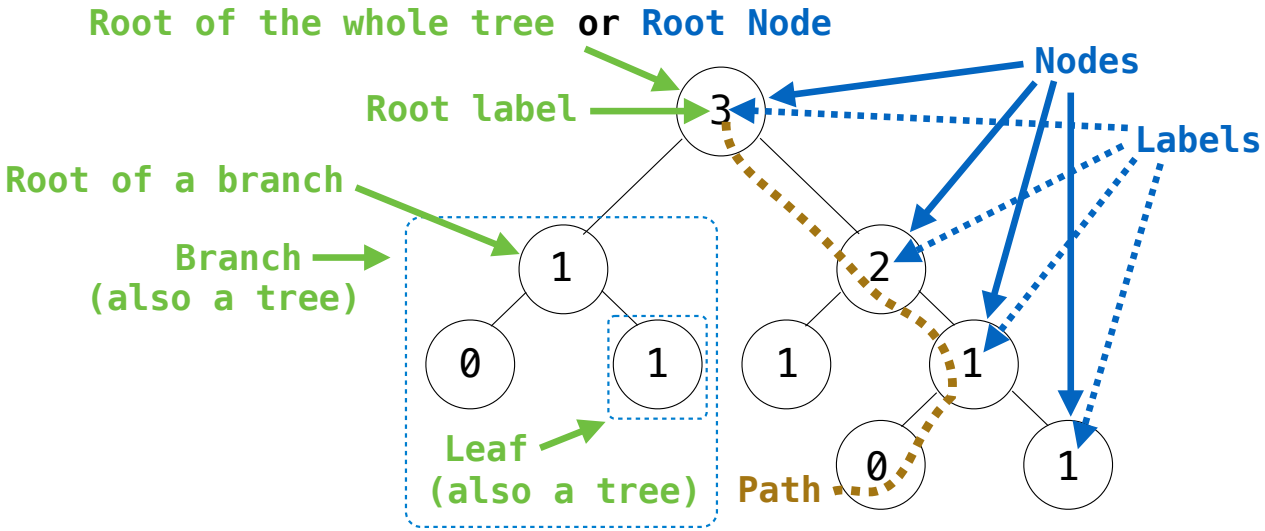
E.g., refer to the parts of a line (affine function) called f:

- `slope(f)` instead of `f[0]` or `f['slope']`
- `y_intercept(f)` instead of `f[1]` or `f['y_intercept']`

Why? Code becomes easier to read & revise

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**:

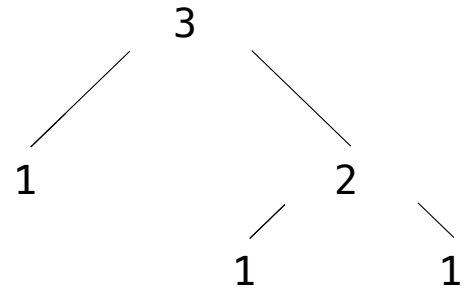
- Get the label for the root of the tree: `label(t)`
- Get the list of branches for the tree: `branches(t)`
- Determine whether the tree is a leaf: `is_leaf(t)`
- Treat t as a value: `return t`, `f(t)`, `[t]`, `s = t`, etc.

(Demo)

Implementing the Tree Abstraction

```
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

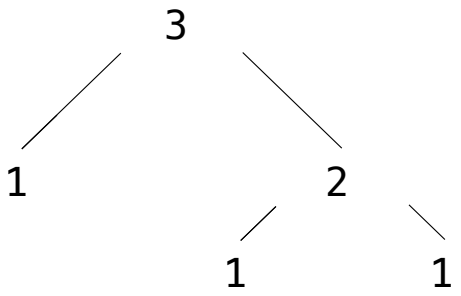
```
def tree(label, branches=[]):
    for branch in branches:
        assert is_tree(branch)
    return [label] + list(branches)

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
```

Verifies the tree definition

Creates a list from a sequence of branches

Verifies that tree is bound to a list

- 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]]]
```

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

(Demo)

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

```
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)

Example: Summing Paths

(Demo)

Example: Counting Paths

Count Paths that have a Total Label Sum

```
def count_paths(t, total):
    """Return the number of paths from the root to any node in tree t
    for which the labels along the path sum to total.

    >>> t = tree(3, [tree(-1), tree(1, [tree(2, [tree(1)]), tree(3)]), tree(1, [tree(-1)])])
    >>> count_paths(t, 3)
    2
    >>> count_paths(t, 4)
    2
    >>> count_paths(t, 5)
    0
    >>> count_paths(t, 6)
    1
    >>> count_paths(t, 7)
    2
    """
    if label(t) == total:
        found = 1
    else:
        found = 0
    return found + sum([count_paths(b, total - label(t)) for b in branches(t)])
```