

# 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)

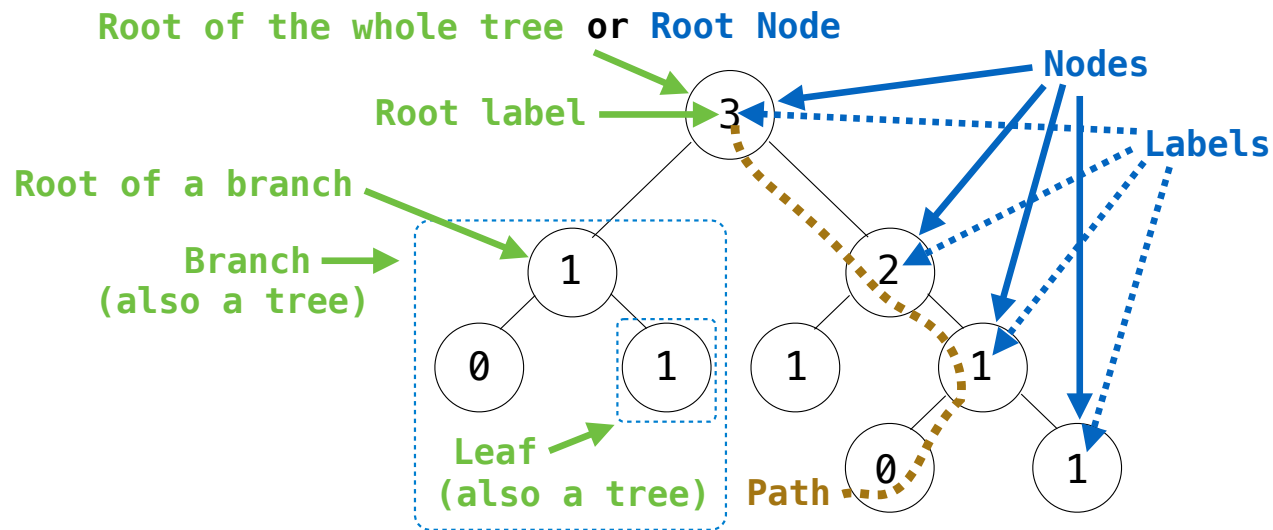
E.g., refer to the parts of a line (affine function) called `f`:

- `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; later you could represent a line `f` as a Python function or as two points instead of a `[slope, 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: `label(t)`
- Get the list of branches for the tree: `branches(t)`
- Get the branch at index `i`, which is a tree: `branches(t)[i]`
- 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

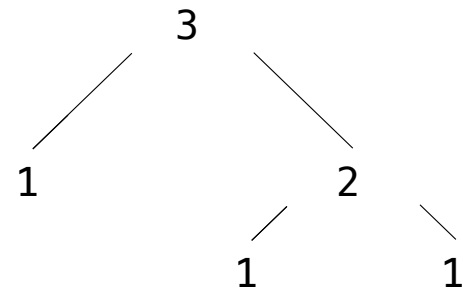
---

```
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

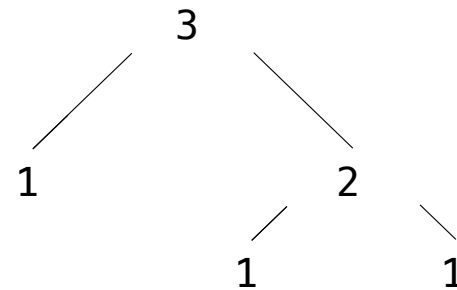
```
def tree(label, branches=[]):  
    for branch in branches:  
        assert is_tree(branch)  
    return [label] + list(branches)  
  
def label(tree):  
    return tree[0]  
  
def branches(tree):  
    return tree[1:]  
  
def is_tree(tree):  
    if type(tree) != list or len(tree) < 1:  
        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

## 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)
```

## 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

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

```
def above_root(t):  
    """Print all the labels of t that are larger than the root label."""  
    def process(u):  
        if label(u) > label(t):  
            print(label(u))  
        for b in branches(u):  
            process(b)  
    process(t)
```