Data Abstraction

Announcements

Lists, Slices, & Recursion

A List is a First Element and the Rest of the List

For any list **s**, the expression **s[1:]** is called a *slice* from index 1 to the end (or 1 onward)

- The value of s[1:] is a list whose length is one less than the length of s
- It contains all of the elements of s except s[0]
- Slicing s doesn't affect s

```
>>> s = [2, 3, 6, 4]
>>> s[1:]
[3, 6, 4]
>>> s
[2, 3, 6, 4]
```

In a list **s**, the first element is **s[0]** and the rest of the elements are **s[1:]**.

Recursion Example: Sum

Implement **sum_list**, which takes a list of numbers s and returns their sum. If a list is empty, the sum of its elements is 0.

```
def sum_list(s):
    """Sum the elements of list s.

    >>> sum([2, 4, 1, 3])
    10
    """

    if len(s) == 0:

        return 0

    else:

        return __s[0]__ + __sum_list(s[1:])__
```

Recursive idea: The sum of the elements of a list is the result of adding the first element to the sum of the rest of the elements

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>}`

Example: Multiples

Implement **multiples**, which takes two lists of positive numbers **s** and **factors.** It returns a dictionary in which each element of factors is a key, and the value for each key is a list of the elements of **s** that are mulitples of the key.

```
def multiples(s, factors):
    """Create a dictionary where each factor is a key and each value
    is the elements of s that are multiples of the key.

    >>> multiples([3, 4, 5, 6, 7, 8], [2, 3])
    {2: [4, 6, 8], 3: [3, 6]}
    >>> multiples([1, 2, 3, 4, 5], [2, 5, 8])
    {2: [2, 4], 5: [5], 8: []}
    """

    return {x: [y for y in ____s____ if _y % x == 0_] for x in _factors_}
```

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']`

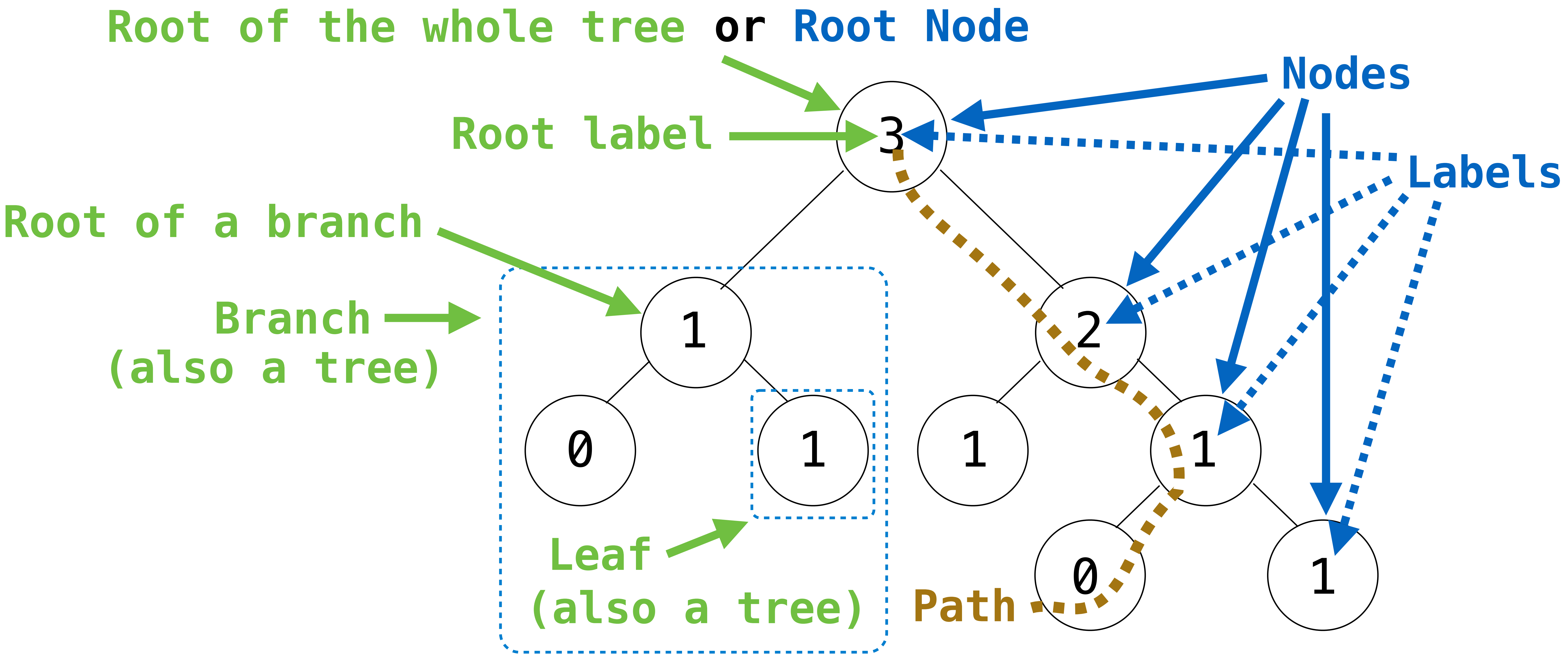
Why? Code becomes easier to read & revise.

(Demo)

Break: 5 minutes

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

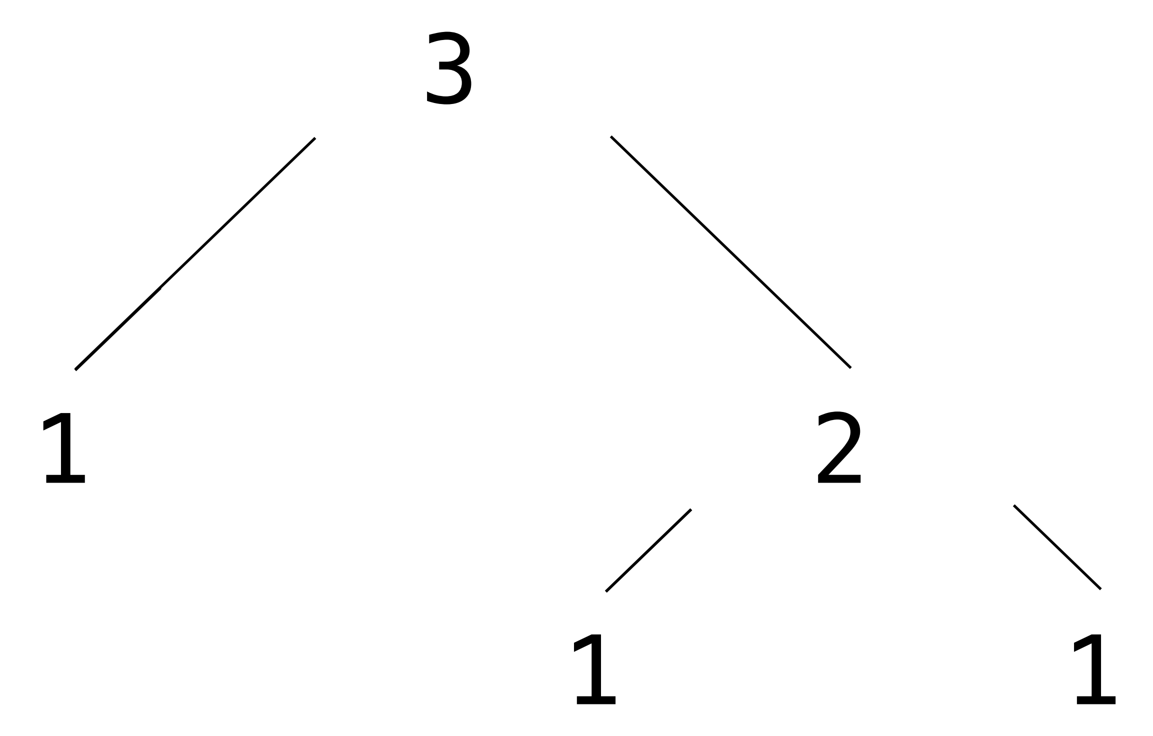
One node can be the **parent/child** of another

The top node is the **root node**

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

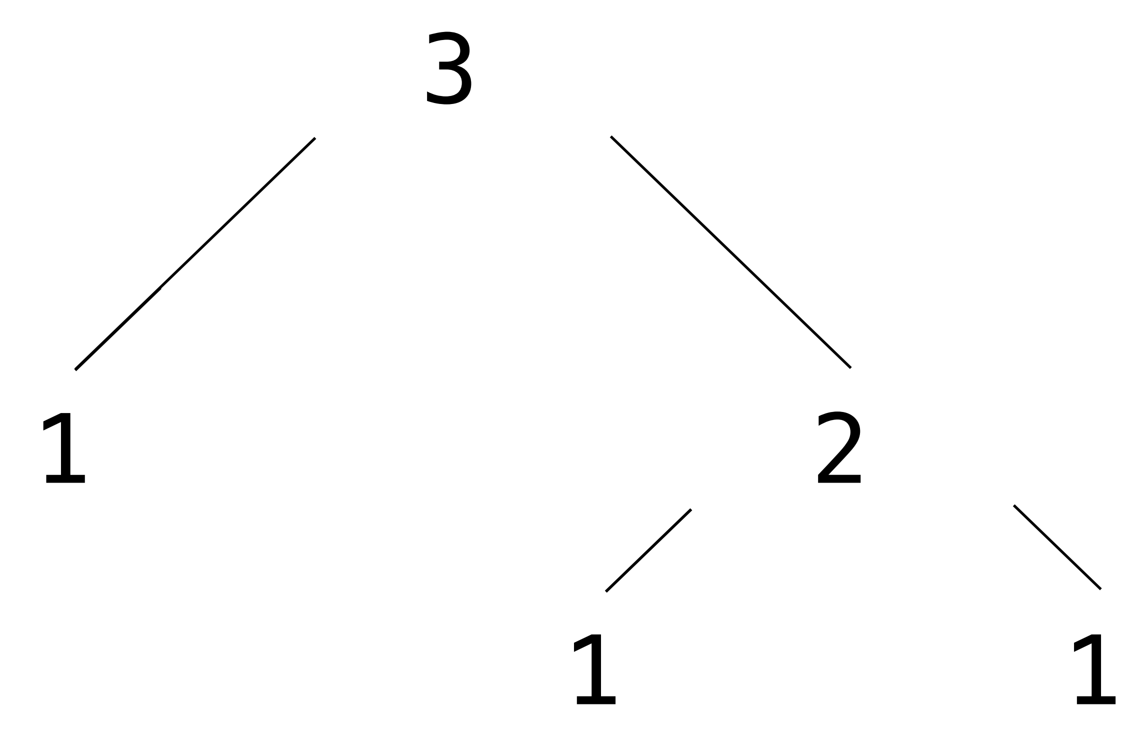
```
def tree(label, branches=[]):
    for branch in branches:
        assert is_tree(branch)
    return [label] + list(branches)

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



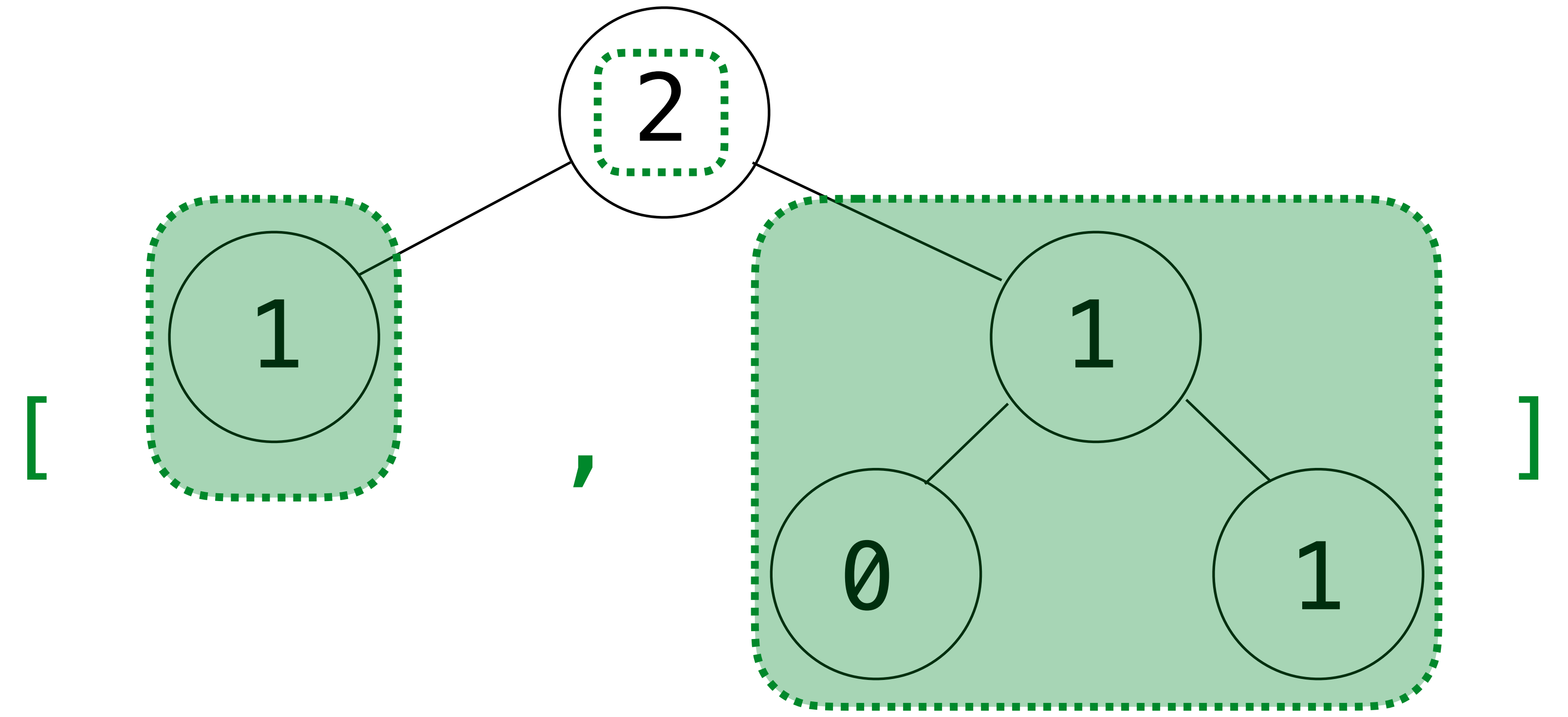

```
>>> tree(3, [tree(1),
...          tree(2, [tree(1),
...                   tree(1)])])
[3, [1], [2, [1], [1]]]

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

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)[0]`
- Determine whether the tree is a leaf: `is_leaf(t)`
- Treat t as a value: `return t`, `f(t)`, `[t]`, `s = t`, etc.

An example tree t:



(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)])
```