Trees

Announcements

The Tree Class

Tree-Structured Data

Objects with parts (such as an instance with multiple attributes) can have parts of parts.

In the world: a person has hands, each hand has fingers, & each finger has joints.

In programs: a dataset has data tables, each table has columns, each column has numbers.

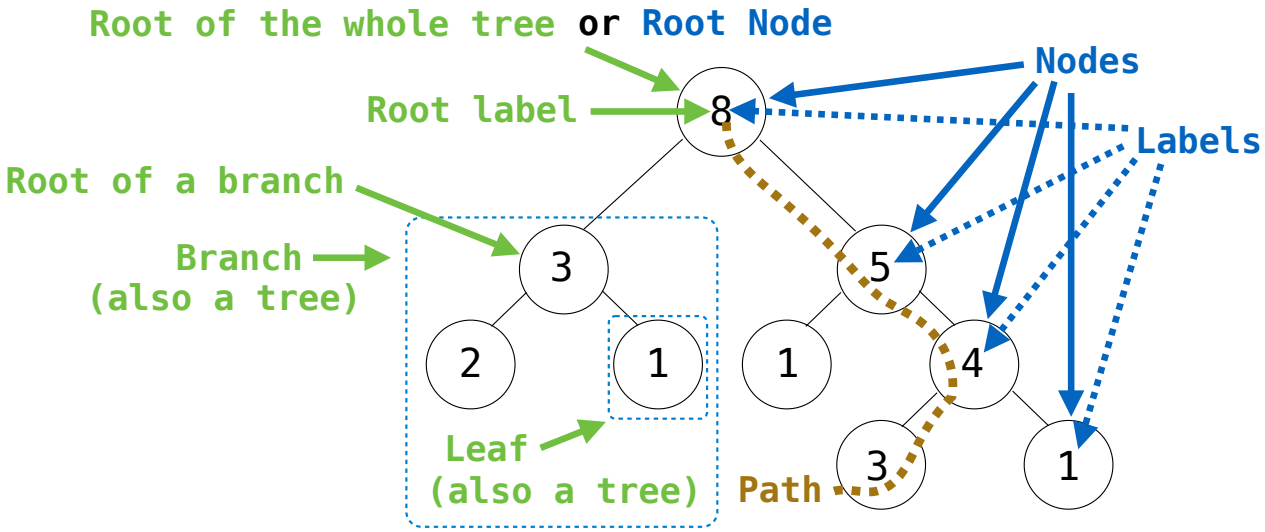
When the parts have the same type as the whole object, the object is *tree structured*.

In the world: an employee has reports, which are employees (& might have reports as well).

In programs: an expression has sub-expressions, which are expressions.

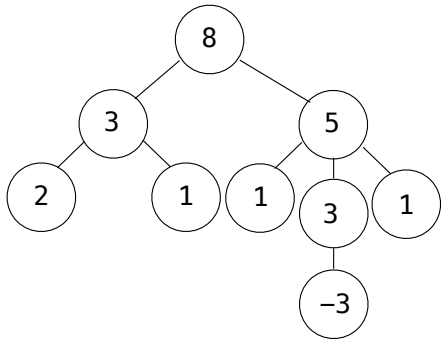
Recursion is commonly used to process tree-structured data.

Tree Terminology



(2 + 1) + (1 + (3 + 1))

(2 + 1) + (1 + abs(−3) + 1)



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**

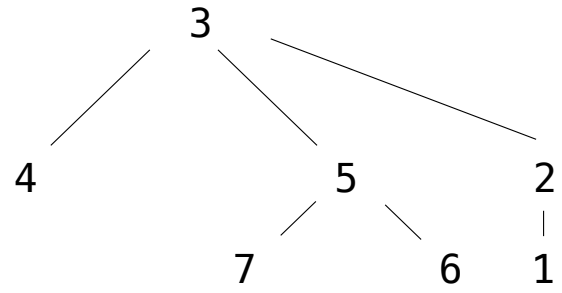
People often refer to labels by their locations: "*each parent is the sum of its children*"

The Tree Class

```
class Tree:
    """A tree has a label and a list of branches."""
    def __init__(self, label, branches=[]):
        self.label = label
        for branch in branches:
            assert isinstance(branch, Tree)
        self.branches = list(branches)

    def is_leaf(self):
        return not self.branches
```

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




```
>>> t
Tree(3, [Tree(4), Tree(5, [Tree(7), Tree(6)]), Tree(2, [Tree(1)])])
>>> print(t)
3
  4
  5
    7
    6
  2
    1
```

```
t = Tree(3, [Tree(4),
             Tree(5, [Tree(7),
                      Tree(6)]),
             Tree(2, [Tree(1)])])
```

Processing Trees

(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 t.is_leaf()
        return 1
    else:
        branch_counts = [count_leaves(b) for b in t.branches]
        return sum(branch_counts)
```

Creating Trees

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

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


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

Example: Counting Paths in a Tree

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 t.label == total :

        found = ________1________

    else:

        found = 0

    return found + ____sum____([ count_paths(b, total - t.label) for b in t.branches])
```