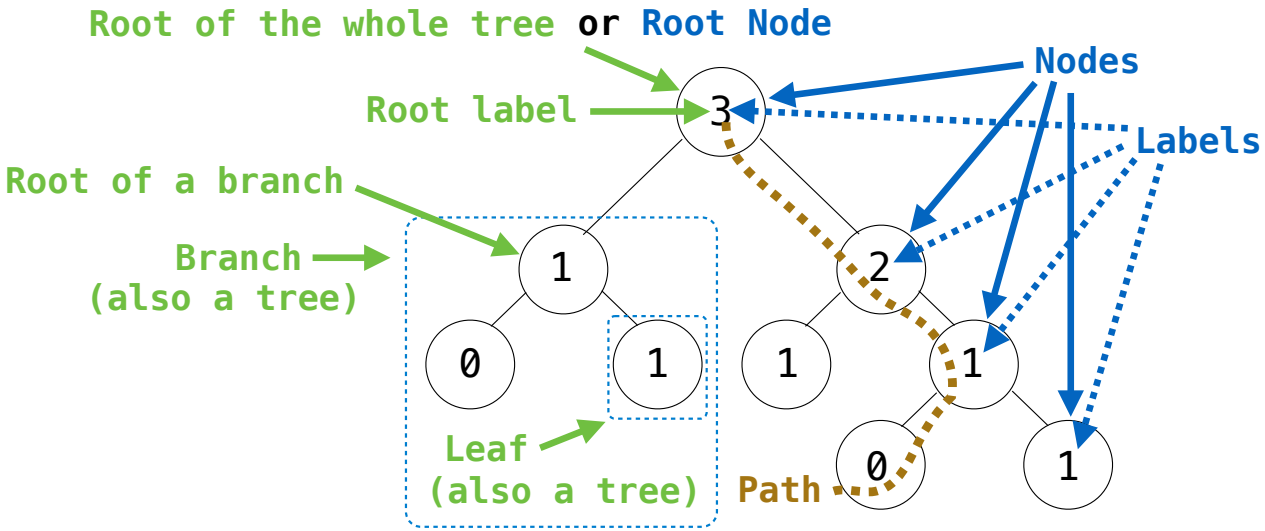
Trees

Announcements

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**

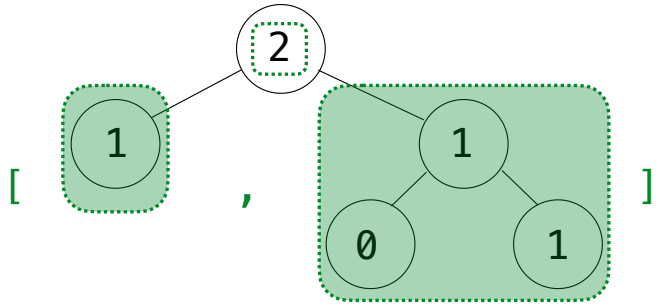
People often refer to labels by their locations: "*each parent is the sum of its children*"

Using the Tree Abstraction

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

- Get the label for the root of the tree: `t.label`
- Get the list of branches for the tree: `t.branches`
- Get the branch at index i, which is a tree: `t.branches[i]`
- 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)

https://composingprograms.com/3ed/draw/

Tree Data Class

```
@dataclass
class Tree[T]:
    label: T
    branches: list[Tree[T]] = field(default_factory=list)  # branches defaults to []
```

The type variable T describes the type of the labels in the tree

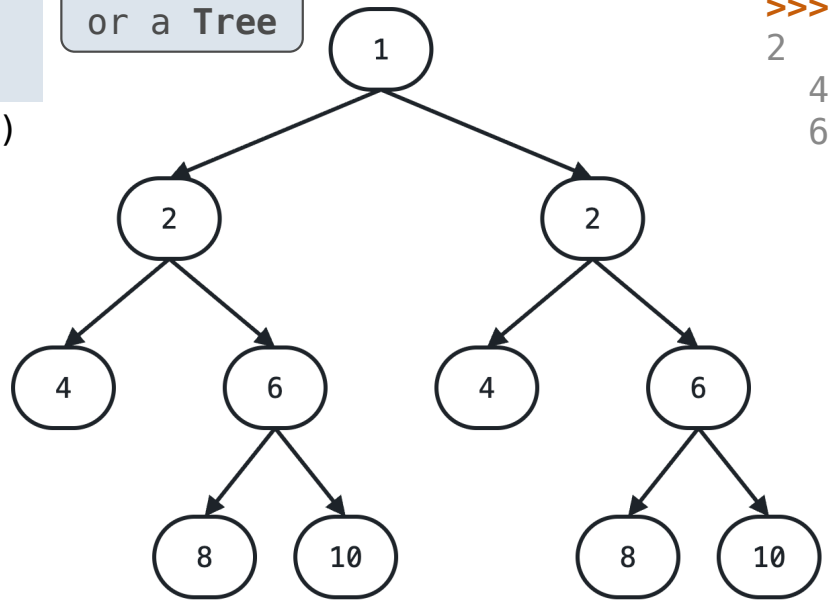
```
def is_leaf(t: Tree) -> bool:
    return not t.branches
```

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

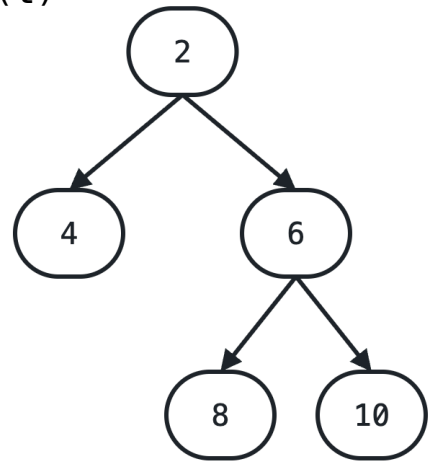
```
>>> print(Tree(1, [t, t]))
1
  2
    4
    6
      8
      10
  2
    4
    6
      8
      10
```

This is a **Tree[int]** or a **Tree**

```
t = Tree(2, [Tree(4), Tree(6, [Tree(8), Tree(10)])])
```




```
>>> print(t)
2
  4
  6
    8
    10
```



Trees vs Linked Lists

```
@dataclass
class Tree[T]:
    label: T
    branches: list[Tree[T]] = field(default_factory=list)  # branches defaults to []
```

```
def is_leaf(t: Tree) -> bool:
    return not t.branches
```

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

- A **linked list** is either empty **or** a **first** value and the **rest**
- The rest is a linked list

```
type LinkedList[T] = Link[T] | tuple[()]

@dataclass
class Link[T]:
    first: T
    rest: LinkedList[T] = ()  # rest defaults to ()
```

What's similar?

- A Tree in which each node has at most one child is just like a linked list

What's different?

- Trees can have many branches
- Trees can't be empty (course-specific decision)

Tree Processing

Writing Recursive Functions

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

- What **small initial choice** can I make?
  - For trees, often: which branch to explore?
- What **recursive call for each option**?
- How can you **combine the results** of those recursive calls?
  - 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? E.g.,
    - Look to see if any option evaluated to true
    - Add up the results from each option

Tree Processing Uses Recursion

Small, initial choice: which branch's leaves to count?

Recursive call for each option: for each branch b, `count_leaves(b)`

Number of leaves on branch b

Combine results: add up all of the counts

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?

```
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 t.branches]
        return sum(branch_counts)
```

Processing a leaf is often the base case

Example: Largest Label

Small, initial choice: which branch to look for the largest label on?

Recursive call for each option: for each branch b, largest_label(b)

Combine results:

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?

```
def largest_label(t):
    """Return the largest label in tree t."""
    if is_leaf(t):
        return __________
    else:
        return ____([_________________ for b in t.branches] + ___________)
```

Example: Largest Label

Small, initial choice: which branch to look for the largest label on?

Recursive call for each option: for each branch b, largest_label(b)

A label that's the largest one from branch b

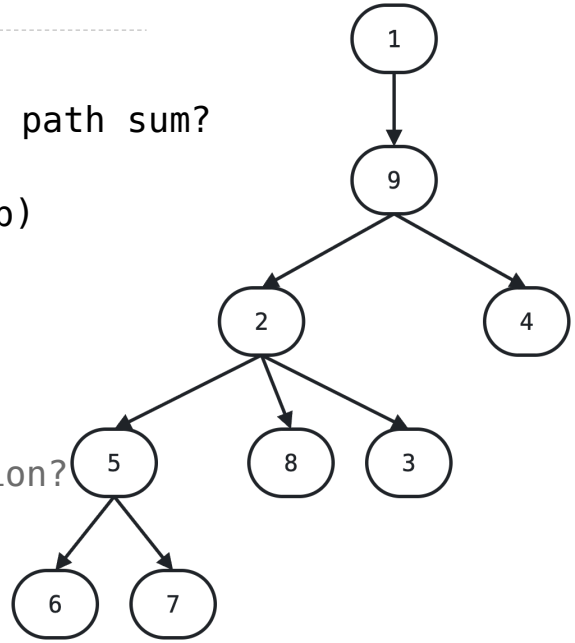
Combine results: Return the largest of these and the root label

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?

```
def largest_label(t):
    """Return the largest label in tree t."""
    if is_leaf(t):
        return t.label
    else:
        return max([largest_label(b) for b in t.branches] + [t.label])
```

What would happen if we got rid of this?

Example: Largest Path Sum



Small, initial choice: which branch contains the rest of the largest path sum?

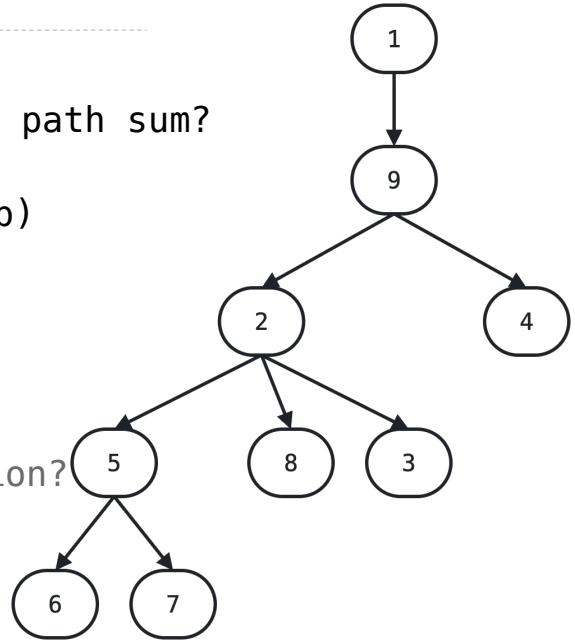
Recursive call for each option: for each branch b, largest_path_sum(b)

Combine results:

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?

```
def largest_path_sum(t):
    """Return the largest sum of the labels along a path from the root to a leaf."""
    if is_leaf(t):
        return __________
    else:
        return ____________________________________________________________
```

Example: Largest Path Sum



Small, initial choice: which branch contains the rest of the largest path sum?

Recursive call for each option: for each branch b, largest_path_sum(b)

Combine results:

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?

```
def largest_path_sum(t):
    """Return the largest sum of the labels along a path from the root to a leaf."""
    if is_leaf(t):
        return t.label
    else:
        return t.label + max([largest_path_sum(b) for b in t.branches])
```

Example: Above Root

Small, initial choice: Which branch to look at for labels to print?

Recursive call for each option: For each branch b, process(b)

Combine results: Don't need to combine the recursive return call!
Do need to print this label, if it's larger than the root

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

Example: Above Root

Small, initial choice: Which branch to look at for labels to print?

Recursive call for each option: For each branch b, process(b)

Combine results: Don't need to combine the recursive return call!
Do need to print this label, if it's larger than the root

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