Announcements
Dictionaries

{'Dem': 0}
Limitations on Dictionaries
Limitations on Dictionaries

Dictionaries are **unordered** collections of key-value pairs
Limitations on Dictionaries

Dictionaries are *unordered* collections of key-value pairs

Dictionary keys do have two restrictions:
Limitations on Dictionaries

Dictionaries are unordered collections of key–value pairs.

Dictionary keys do have two restrictions:

• A key of a dictionary cannot be a list or a dictionary (or any mutable type)
Limitations on Dictionaries

Dictionaries are unordered collections of key–value pairs

Dictionary keys do have two restrictions:

• A key of a dictionary cannot be a list or a dictionary (or any mutable type)

• Two keys cannot be equal; There can be at most one value for a given key
Limitations on Dictionaries

Dictionaries are unordered collections of key-value pairs

Dictionary keys do have two restrictions:

• A key of a dictionary cannot be a list or a dictionary (or any mutable type)

• Two keys cannot be equal; There can be at most one value for a given key

This first restriction is tied to Python's underlying implementation of dictionaries
Limitations on Dictionaries

Dictionaries are unordered collections of key–value pairs

Dictionary keys do have two restrictions:

• A key of a dictionary cannot be a list or a dictionary (or any mutable type)

• Two keys cannot be equal; There can be at most one value for a given key

This first restriction is tied to Python's underlying implementation of dictionaries

The second restriction is part of the dictionary abstraction
Limitations on Dictionaries

Dictionaries are unordered collections of key-value pairs

Dictionary keys do have two restrictions:

- A key of a dictionary cannot be a list or a dictionary (or any mutable type)

- Two keys cannot be equal; There can be at most one value for a given key

This first restriction is tied to Python's underlying implementation of dictionaries

The second restriction is part of the dictionary abstraction

If you want to associate multiple values with a key, store them all in a sequence value
Box-and-Pointer Notation
The Closure Property of Data Types
The Closure Property of Data Types

• A method for combining data values satisfies the closure property if:
  The result of combination can itself be combined using the same method
The Closure Property of Data Types

• A method for combining data values satisfies the closure property if:
  The result of combination can itself be combined using the same method

• Closure is powerful because it permits us to create hierarchical structures
The Closure Property of Data Types

- A method for combining data values satisfies the closure property if:
  The result of combination can itself be combined using the same method
- Closure is powerful because it permits us to create hierarchical structures
- Hierarchical structures are made up of parts, which themselves are made up of parts, and so on
### The Closure Property of Data Types

- A method for combining data values satisfies the *closure property* if:
  - The result of combination can itself be combined using the same method

- Closure is powerful because it permits us to create hierarchical structures

- Hierarchical structures are made up of parts, which themselves are made up of parts, and so on

Lists can contain lists as elements (in addition to anything else)
Box-and-Pointer Notation in Environment Diagrams
Box-and-Pointer Notation in Environment Diagrams

Lists are represented as a row of index-labeled adjacent boxes, one per element.
Box-and-Pointer Notation in Environment Diagrams

Lists are represented as a row of index-labeled adjacent boxes, one per element.
Each box either contains a primitive value or points to a compound value.

Interactive Diagram
Box-and-Pointer Notation in Environment Diagrams

Lists are represented as a row of index-labeled adjacent boxes, one per element.
Each box either contains a primitive value or points to a compound value.

pair = [1, 2]
Box-and-Pointer Notation in Environment Diagrams

Lists are represented as a row of index-labeled adjacent boxes, one per element. Each box either contains a primitive value or points to a compound value.

pair = [1, 2]
Box-and-Pointer Notation in Environment Diagrams

Lists are represented as a row of index-labeled adjacent boxes, one per element. Each box either contains a primitive value or points to a compound value.

```
1 pair = [1, 2]
2
3 nested_list = [[1, 2], [],
4     [[3, False, None],
5         [4, lambda: 5]]]
```
Slicing

(Demo)
Slicing Creates New Values

Interactive Diagram
Slicing Creates New Values

```python
1  digits = [1, 8, 2, 8]
2  start =  digits[:1]
3  middle = digits[1:3]
4  end =  digits[2:]
```
Slicing Creates New Values

```python
1  digits = [1, 8, 2, 8]
2  start = digits[:1]
3  middle = digits[1:3]
4  end = digits[2:]
```

Interactive Diagram
Processing Container Values
Sequence Aggregation
Sequence Aggregation

Several built-in functions take iterable arguments and aggregate them into a value
Sequence Aggregation

Several built-in functions take iterable arguments and aggregate them into a value

- **sum(iterable[, start]) -> value**

  Return the sum of an iterable of numbers (NOT strings) plus the value of parameter 'start' (which defaults to 0). When the iterable is empty, return start.
Sequence Aggregation

Several built-in functions take iterable arguments and aggregate them into a value

- **sum**(iterable[, start]) -> value

  Return the sum of an iterable of numbers (NOT strings) plus the value of parameter 'start' (which defaults to 0). When the iterable is empty, return start.

- **max**(iterable[, key=func]) -> value
  max(a, b, c, ...[, key=func]) -> value

  With a single iterable argument, return its largest item. With two or more arguments, return the largest argument.
Sequence Aggregation

Several built-in functions take iterable arguments and aggregate them into a value

- **sum(iterable[, start])** -> value

  Return the sum of an iterable of numbers (NOT strings) plus the value of parameter 'start' (which defaults to 0). When the iterable is empty, return start.

- **max(iterable[, key=func])** -> value
  max(a, b, c, ...[, key=func]) -> value

  With a single iterable argument, return its largest item. With two or more arguments, return the largest argument.

- **all(iterable)** -> bool

  Return True if bool(x) is True for all values x in the iterable. If the iterable is empty, return True.
Sequence Aggregation

Several built-in functions take iterable arguments and aggregate them into a value

• `sum(iterable[, start])` -> value

  Return the sum of an iterable of numbers (NOT strings) plus the value of parameter 'start' (which defaults to 0). When the iterable is empty, return start.

• `max(iterable[, key=func])` -> value
  `max(a, b, c, ...[, key=func])` -> value

  With a single iterable argument, return its largest item. With two or more arguments, return the largest argument.

• `all(iterable)` -> bool

  Return True if `bool(x)` is True for all values x in the iterable. If the iterable is empty, return True.

  (Demo)
Trees
Tree Abstraction
Tree Abstraction

Recursive description (wooden trees):

Relative description (family trees):
Tree Abstraction

Recursive description (wooden trees):
A tree has a root value and a list of branches

Relative description (family trees):
Tree Abstraction

Recursive description (wooden trees):
A tree has a root value and a list of branches.

Relative description (family trees):
Recursive description (wooden trees): A tree has a root value and a list of branches

Relative description (family trees):
**Tree Abstraction**

**Recursive description** (wooden trees):
A tree has a root value and a list of branches.
Each branch is a tree.

**Relative description** (family trees):
Root value

```
      3
     / \
    1   2
   /   /   /
  0   1   1
     /   /
    0   1
```
**Recursive description (wooden trees):**

A tree has a root value and a list of branches.

Each branch is a tree.

A tree with zero branches is called a leaf.

**Relative description (family trees):**
Tree Abstraction

Recursive description (wooden trees): A tree has a root value and a list of branches. Each branch is a tree. A tree with zero branches is called a leaf.

Relative description (family trees):
Tree Abstraction

**Recursive description** (wooden trees):
A tree has a root value and a list of branches.
Each branch is a tree.
A tree with zero branches is called a leaf.

**Relative description** (family trees):
Each location in a tree is called a node.
Tree Abstraction

Recursive description (wooden trees):
A **tree** has a **root** value and a list of **branches**.
Each branch is a **tree**.
A tree with zero branches is called a **leaf**.

Relative description (family trees):
Each location in a tree is called a **node**.
**Tree Abstraction**

**Recursive description (wooden trees):**
A tree has a root value and a list of branches.
Each branch is a tree.
A tree with zero branches is called a leaf.

**Relative description (family trees):**
Each location in a tree is called a node.
Each node has a value.
**Recursive description (wooden trees):**
A **tree** has a root value and a list of **branches**
Each branch is a **tree**
A tree with zero branches is called a **leaf**

**Relative description (family trees):**
Each location in a tree is called a **node**
Each **node** has a **value**
Recursive description (wooden trees):
A **tree** has a **root** value and a list of **branches**
Each branch is a **tree**
A tree with zero branches is called a **leaf**

Relative description (family trees):
Each location in a tree is called a **node**
Each **node** has a **value**
One node can be the **parent/child** of another
Tree Abstraction

Recursive description (wooden trees):
A tree has a root value and a list of branches.
Each branch is a tree.
A tree with zero branches is called a leaf.

Relative description (family trees):
Each location in a tree is called a node.
Each node has a value.
One node can be the parent/child of another.

People often refer to values by their locations: "each parent is the sum of its children"
Implementing the Tree Abstraction
Implementing the Tree Abstraction

• A tree has a root value and a list of branches
• Each branch is a tree
Implementing the Tree Abstraction

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

```
3
/  \   /
1    2 1
```

Implementing the Tree Abstraction

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

```python
>>> tree(3, [tree(1),
...    tree(2, [tree(1),
...        tree(1)])])
```
Implementing the Tree Abstraction

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

```python
>>> tree(3, [tree(1),
    ...
    tree(2, [tree(1),
    ...
    tree(1)])])
[3, [1], [2, [1], [1]]]
```
Implementing the Tree Abstraction

```python
def tree(root, branches=[]):
```

- A tree has a root value 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(root, branches=[]):
    return [root] + branches

• A tree has a root value and a list of branches
• Each branch is a tree

```python
def tree(root, branches=[]):
    return [root] + branches
```

```
>>> tree(3, [tree(1),
    ...    tree(2, [tree(1),
    ...        tree(1)]))])
[3, [1], [2, [1], [1]]]
```
Implementing the Tree Abstraction

def tree(root, branches=[]):
    return [root] + branches

def root(tree):

• A tree has a root value 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

```python
def tree(root, branches=[]):
    return [root] + branches

def root(tree):
    return tree[0]
```

- A tree has a root value 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

```python
def tree(root, branches=[]):
    return [root] + branches

def root(tree):
    return tree[0]

def branches(tree):
```

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

```python
>>> tree(3, [tree(1),
...    tree(2, [tree(1),
...      tree(1)])])
[3, [1], [2, [1], [1]]]
```
Implementing the Tree Abstraction

```python
def tree(root, branches=[]):
    return [root] + branches

def root(tree):
    return tree[0]

def branches(tree):
    return tree[1:]

>>> tree(3, [tree(1),
    ... tree(2, [tree(1),
    ... tree(2, [tree(1)])])
    [3, [1], [2, [1], [1]]]
```
Implementing the Tree Abstraction

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

def root(tree):
    return tree[0]

def branches(tree):
    return tree[1:]
```

A tree has a root value 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

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

def root(tree):
    return tree[0]

def branches(tree):
    return tree[1:]
```

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

```python
>>> tree(3, [tree(1),
... tree(2, [tree(1),
... tree(1)])])
[3, [1], [2, [1], [1]]]
```
Implementing the Tree Abstraction

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

def root(tree):
    return tree[0]

def branches(tree):
    return tree[1:]

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

- A tree has a root value and a list of branches
- Each branch is a tree
Implementing the Tree Abstraction

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

def root(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

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

```python
>>> tree(3, [tree(1), ...
    ...      tree(2, [tree(1), ...
    ...          [3, [1], [2, [1], [1]]])
```
Implementing the Tree Abstraction

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

def root(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

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

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

def root(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

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

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

def root(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
```
Implementing the Tree Abstraction

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

def root(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
```

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

A tree has a root value and a list of branches.
Each branch is a tree.

```python
def tree(root, branches=[]):
    Verifies the tree definition
    for branch in branches:
        assert is_tree(branch)
    return [root] + list(branches)

def root(tree):
    Creates a list from a sequence of branches
    return tree[0]

def branches(tree):
    Verifies that tree is bound to a list
    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
```

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

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

### Example

```
3
  1
  2
  1
```

```python
def tree(3, [tree(1), ...
```
Implementing the Tree Abstraction

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

def root(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

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

def is_leaf(tree):
    return not branches(tree)  # Demo
```
Tree Processing
Tree Processing

(Demo)
Tree Processing Uses Recursion
Tree Processing Uses Recursion

```python
def count_leaves(t):
    """Count the leaves of a tree."""
```
Tree Processing Uses Recursion

Processing a leaf is often the base case of a tree processing function

```python
def count_leaves(t):
    """Count the leaves of a tree."""
```
Tree Processing Uses Recursion

Processing a leaf is often the base case of a tree processing function

```python
def count_leaves(t):
    """Count the leaves of a tree."""
    if is_leaf(t):
        return 1
```
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
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:

```python
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)]
```
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.

```python
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)
```
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.

```python
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)
Discussion Question
Discussion Question

Implement `leaves`, which returns a list of the leaf values of a tree
Discussion Question

Implement `leaves`, which returns a list of the leaf values of a tree

```python
def leaves(tree):
    """Return a list containing the leaves of tree.

    >>> leaves(fib_tree(5))
    [1, 0, 1, 0, 1, 1, 0, 1]
    """
```
Discussion Question

Implement `leaves`, which returns a list of the leaf values of a tree

*Hint*: If you `sum` a list of lists, you get a list containing the elements of those lists

```python
def leaves(tree):
    """Return a list containing the leaves of tree."
    
    >>> leaves(fib_tree(5))
    [1, 0, 1, 0, 1, 1, 0, 1]
```

"""
**Discussion Question**

Implement `leaves`, which returns a list of the leaf values of a tree

*Hint*: If you `sum` a list of lists, you get a list containing the elements of those lists

```python
>>> sum([[1], [2, 3], [4]], [])
```

```python
def leaves(tree):
    '''Return a list containing the leaves of tree.

    >>> leaves(fib_tree(5))
    [1, 0, 1, 0, 1, 1, 0, 1]
    '''
```
Discussion Question

Implement `leaves`, which returns a list of the leaf values of a tree

*Hint*: If you `sum` a list of lists, you get a list containing the elements of those lists

```python
def leaves(tree):
    """Return a list containing the leaves of tree."
    return sum(tree, [])

>>> leaves(fib_tree(5))
[1, 0, 1, 0, 1, 1, 0, 1]
```
Discussion Question

Implement `leaves`, which returns a list of the leaf values of a tree

*Hint: If you `sum` a list of lists, you get a list containing the elements of those lists*

```python
>>> sum([[1], [2, 3], [4]], [])
[1, 2, 3, 4]
>>> sum([[1]], [])

```
Discussion Question

Implement `leaves`, which returns a list of the leaf values of a tree

*Hint*: If you `sum` a list of lists, you get a list containing the elements of those lists

```python
>>> sum([ [1], [2, 3], [4] ], [])
[1, 2, 3, 4]
>>> sum([ [1] ], [])
[1]
```

```python
def leaves(tree):
    """Return a list containing the leaves of tree."
    >>> leaves(fib_tree(5))
    [1, 0, 1, 1, 0, 1, 0, 1]
    """
```
Discussion Question

Implement \texttt{leaves}, which returns a list of the leaf values of a tree.

\textit{Hint:} If you \texttt{sum} a list of lists, you get a list containing the elements of those lists.

\begin{Verbatim}
>>> sum([[1], [2, 3], [4]], [])
[1, 2, 3, 4]
>>> sum([[1], []])
[1]
>>> sum([[1], [2]], [])
[1, 0, 1, 0, 1, 1, 0, 1]
\end{Verbatim}

```python

def leaves(tree):
    """Return a list containing the leaves of tree."""
    return sum(tree, [])

>>> leaves(fib_tree(5))
[1, 0, 1, 0, 1, 1, 0, 1]
```
**Discussion Question**

Implement `leaves`, which returns a list of the leaf values of a tree

*Hint:* If you `sum` a list of lists, you get a list containing the elements of those lists

```python
>>> sum([[1], [2, 3], [4]], [])
[1, 2, 3, 4]
>>> sum([[1]], [])
[1]
>>> sum([[1], [2]], [])
[[1], 2]
```

```python
def leaves(tree):
    """Return a list containing the leaves of tree."""
    if not tree:
        return []
    else:
        return sum(leaves(t) for t in tree) + [tree[-1][-1]]

>>> leaves(fib_tree(5))
[1, 0, 1, 0, 1, 1, 0, 1]
```
Discussion Question

Implement `leaves`, which returns a list of the leaf values of a tree.

*Hint*: If you `sum` a list of lists, you get a list containing the elements of those lists.

```python
>>> sum([[1], [2, 3], [4]], [])
[1, 2, 3, 4]
>>> sum([[1]], [])
[1]
>>> sum([[1], [2]], [])
[[1], 2]
```

```python
def leaves(tree):
    """Return a list containing the leaves of tree."
    >>> leaves(fib_tree(5))
    [1, 0, 1, 0, 1, 1, 0, 1]
    """
```
Discussion Question

Implement `leaves`, which returns a list of the leaf values of a tree

*Hint:* If you `sum` a list of lists, you get a list containing the elements of those lists

```python
>>> sum([[1], [2, 3], [4]], [])
[1, 2, 3, 4]
>>> sum([1], [])
[1]
>>> sum([[1]], [2], [])
[[1], 2]
```
Discussion Question

Implement `leaves`, which returns a list of the leaf values of a tree

*Hint*: If you `sum` a list of lists, you get a list containing the elements of those lists

```python
>>> sum([[1], [2, 3], [4]], [])
[1, 2, 3, 4]
```

```python
>>> sum([[1]], []
[1]
```

```python
>>> sum([[1]], [2], []
[[1], 2]
```

```python
def leaves(tree):
    """Return a list containing the leaves of tree."
    if is_leaf(tree):
        return [root(tree)]
    else:
        return sum(______________________________, [])
```

```python
>>> leaves(fib_tree(5))
[1, 0, 1, 0, 1, 1, 0, 1]
```

branches(tree)
leaves(tree)
[branches(b) for b in branches(tree)]
[leaves(b) for b in branches(tree)]

[branches(s) for s in leaves(tree)]
[leaves(s) for s in leaves(tree)]
```
Discussion Question

Implement `leaves`, which returns a list of the leaf values of a tree

*Hint:* If you `sum` a list of lists, you get a list containing the elements of those lists

```python
>>> sum([ [1], [2, 3], [4] ], [])
[1, 2, 3, 4]
>>> sum([ [1] ], [])
[1]
>>> sum([ [[1]], [2] ], [])
[[1], 2]
```

```python
def leaves(tree):
    """Return a list containing the leaves of tree."
    if is_leaf(tree):
        return [root(tree)]
    else:
        return sum(List of leaves for each branch, [])
        
branches(tree)
leaves(tree)
[branches(b) for b in branches(tree)]
[leaves(b) for b in branches(tree)]
```
**Discussion Question**

Implement `leaves`, which returns a list of the leaf values of a tree

*Hint*: If you `sum` a list of lists, you get a list containing the elements of those lists

```python
>>> sum([[1], [2, 3], [4]], [])
[1, 2, 3, 4]
>>> sum([[1]], [])
[1]
>>> sum([[1]], [2], [], []
[[1], 2]
```

```python
def leaves(tree):
    """Return a list containing the leaves of tree."
    if is_leaf(tree):
        return [root(tree)]
    else:
        return sum([list of leaves for each branch, []])
```

- `branches(tree)`
- `leaves(tree)`
- `[branches(b) for b in branches(tree)]`
- `[leaves(b) for b in branches(tree)]`
- `[b for b in branches(tree)]`
- `[s for s in leaves(tree)]`
- `[branches(s) for s in leaves(tree)]`
- `[leaves(s) for s in leaves(tree)]`
Creating Trees
Creating Trees

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

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

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

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

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

def increment(t):
    """Return a tree like t but with all node values incremented.""
```

Creating Trees

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

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

def increment(t):
    """Return a tree like t but with all node values incremented.""
    return tree(root(t) + 1, [increment(b) for b in branches(t)])
```
Example: Printing Trees

(Demo)