CS 61A Lecture 12

Monday, September 29

Announcements

Announcements

- Homework 3 due Wednesday 10/1 @ 11:59pm

Announcements

- Homework 3 due Wednesday 10/1 @ 11:59pm
  - Homework Party on Monday 9/29, time and place TBD

Announcements

- Homework 3 due Wednesday 10/1 @ 11:59pm
  - Homework Party on Monday 9/29, time and place TBD
- Optional Hog Contest due Wednesday 10/1 @ 11:59pm

Announcements

- Homework 3 due Wednesday 10/1 @ 11:59pm
  - Homework Party on Monday 9/29, time and place TBD
- Optional Hog Contest due Wednesday 10/1 @ 11:59pm
- Project 2 due Thursday 10/9 @ 11:59pm

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 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 the key to power in any means of combination 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 the key to power in any means of combination 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 the key to power in any means of combination 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

Box-and-Pointer Notation in Environment Diagrams

Interactive Diagram

Box-and-Pointer Notation in Environment Diagrams

Lists are represented as a row of index-labeled adjacent boxes, one per element

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

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

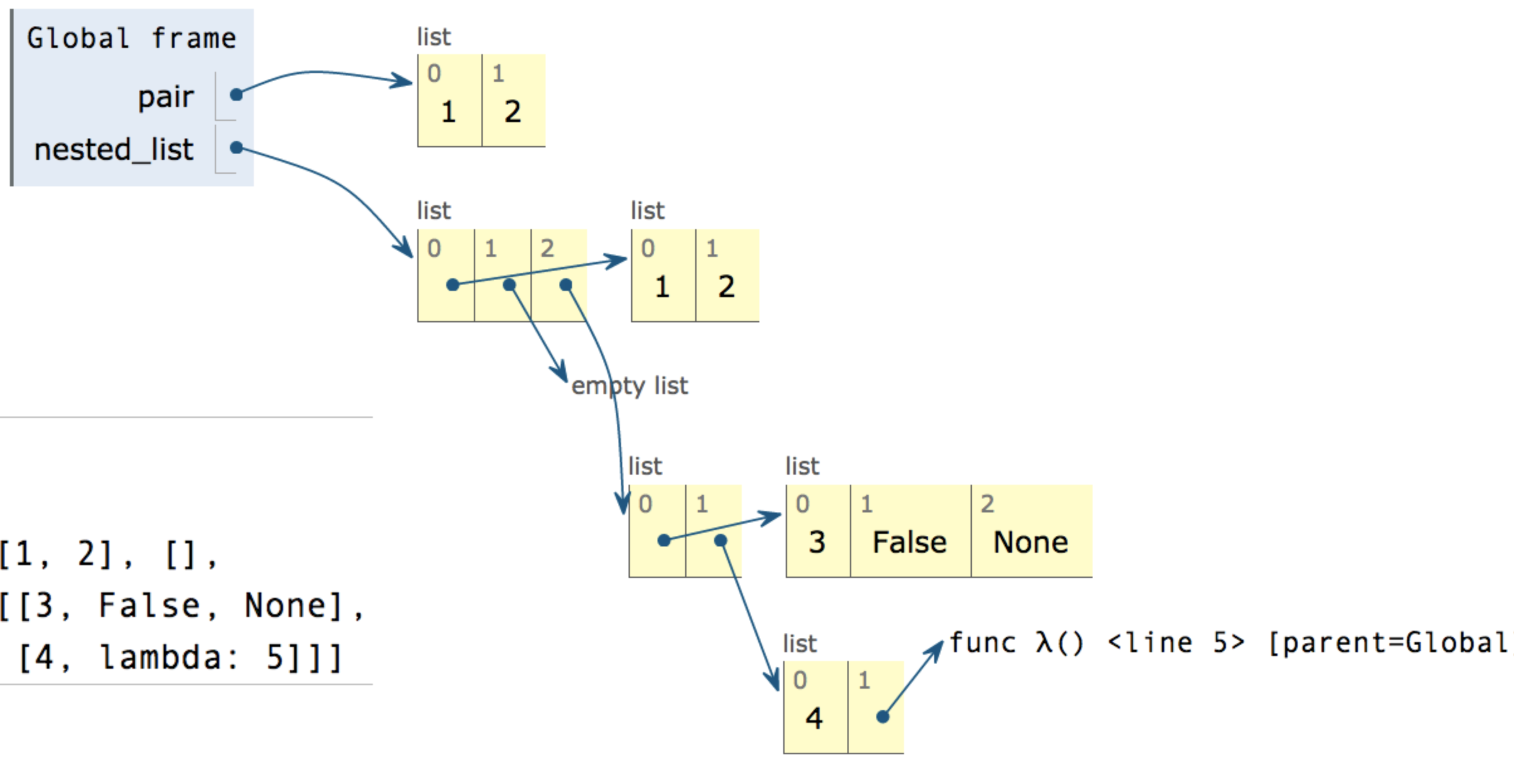
```
pair = [1, 2]

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

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]

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

Interactive Diagram

Trees

Trees are Nested Sequences

Trees are Nested Sequences

A **tree** is either a single value called a ***leaf*** or a sequence of ***trees***

Trees are Nested Sequences

A **tree** is either a single value called a ***leaf*** or a sequence of ***trees***

Typically, some type restriction is placed on the leaves. E.g., a tree of numbers:

Trees are Nested Sequences

A **tree** is either a single value called a ***leaf*** or a sequence of ***trees***

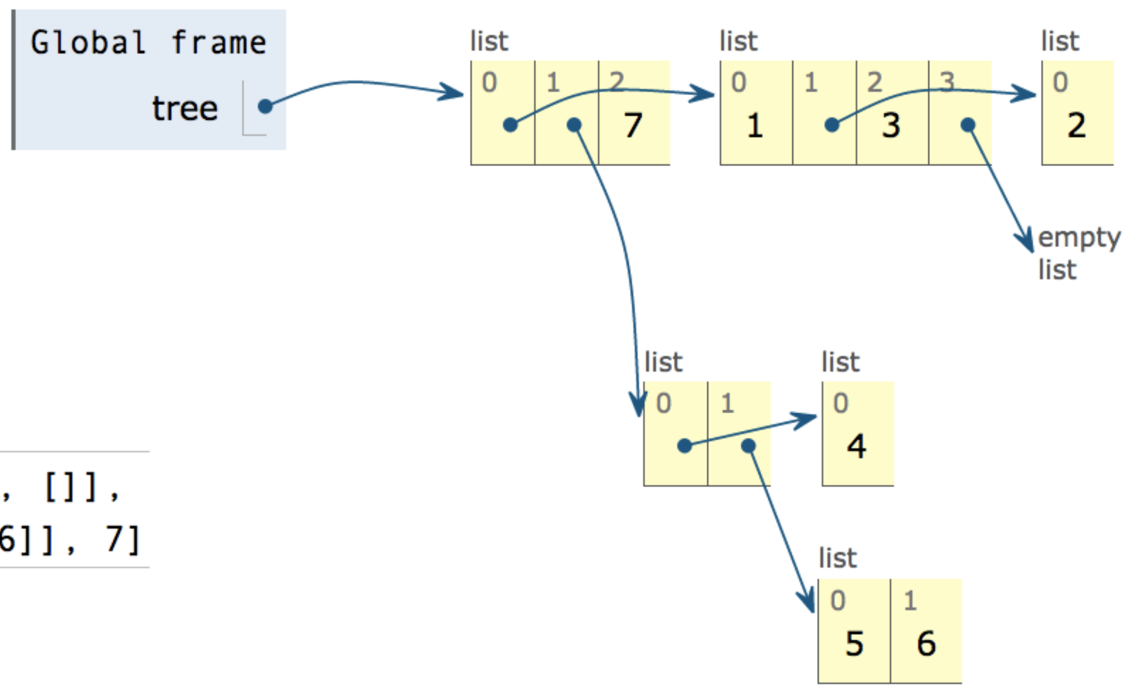
Typically, some type restriction is placed on the leaves. E.g., a tree of numbers:

```
tree = [[1, [2], 3, []],
        [[4], [5, 6]], 7]
```

Trees are Nested Sequences

A **tree** is either a single value called a ***leaf*** or a sequence of ***trees***

Typically, some type restriction is placed on the leaves. E.g., a tree of numbers:

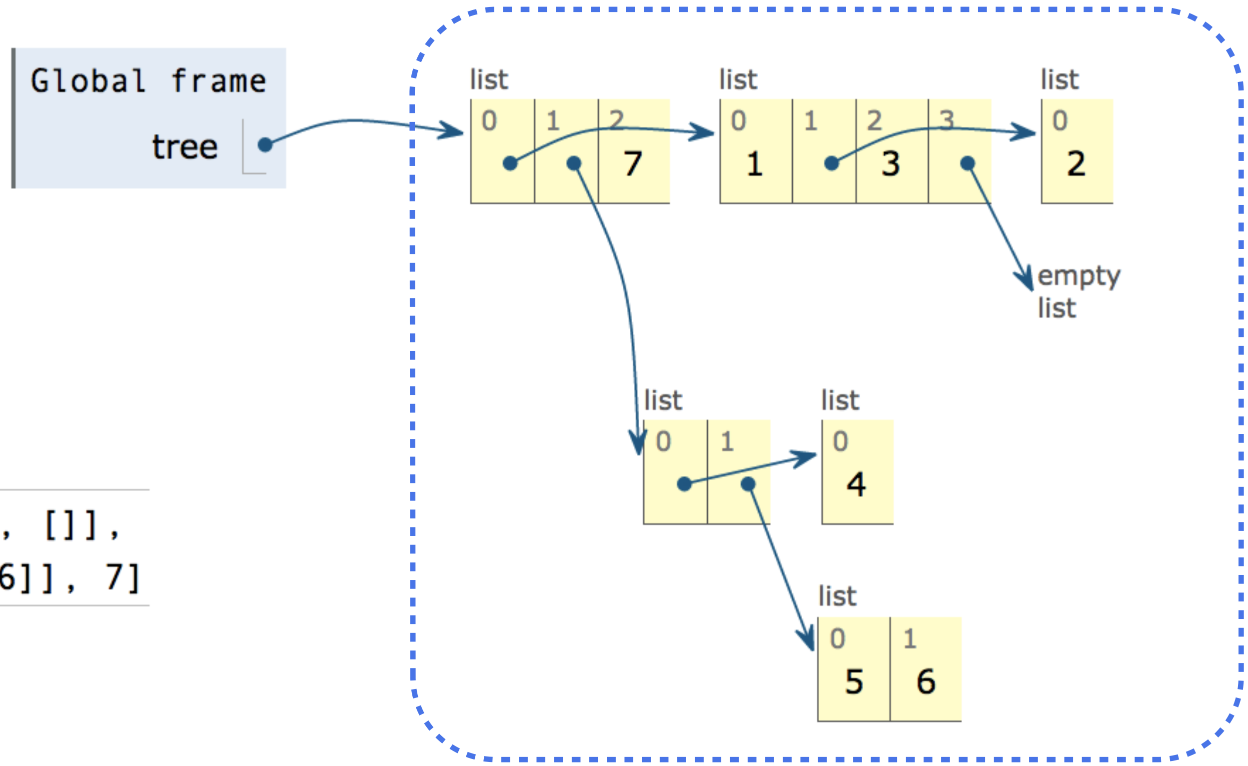



```
tree = [[1, [2], 3, []],
        [[4], [5, 6]], 7]
```

Trees are Nested Sequences

A **tree** is either a single value called a ***leaf*** or a sequence of ***trees***

Typically, some type restriction is placed on the leaves. E.g., a tree of numbers:

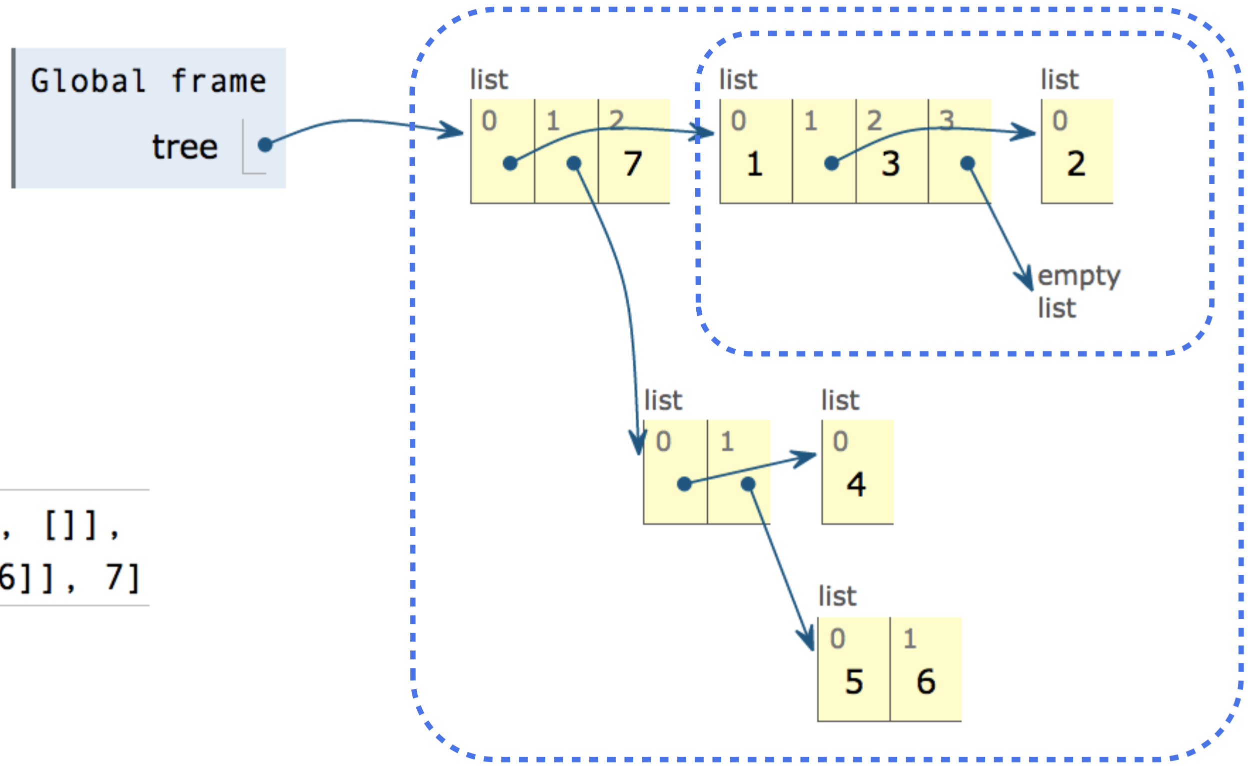



```
tree = [[1, [2], 3, []],
        [[4], [5, 6]], 7]
```

Trees are Nested Sequences

A **tree** is either a single value called a ***leaf*** or a sequence of ***trees***

Typically, some type restriction is placed on the leaves. E.g., a tree of numbers:

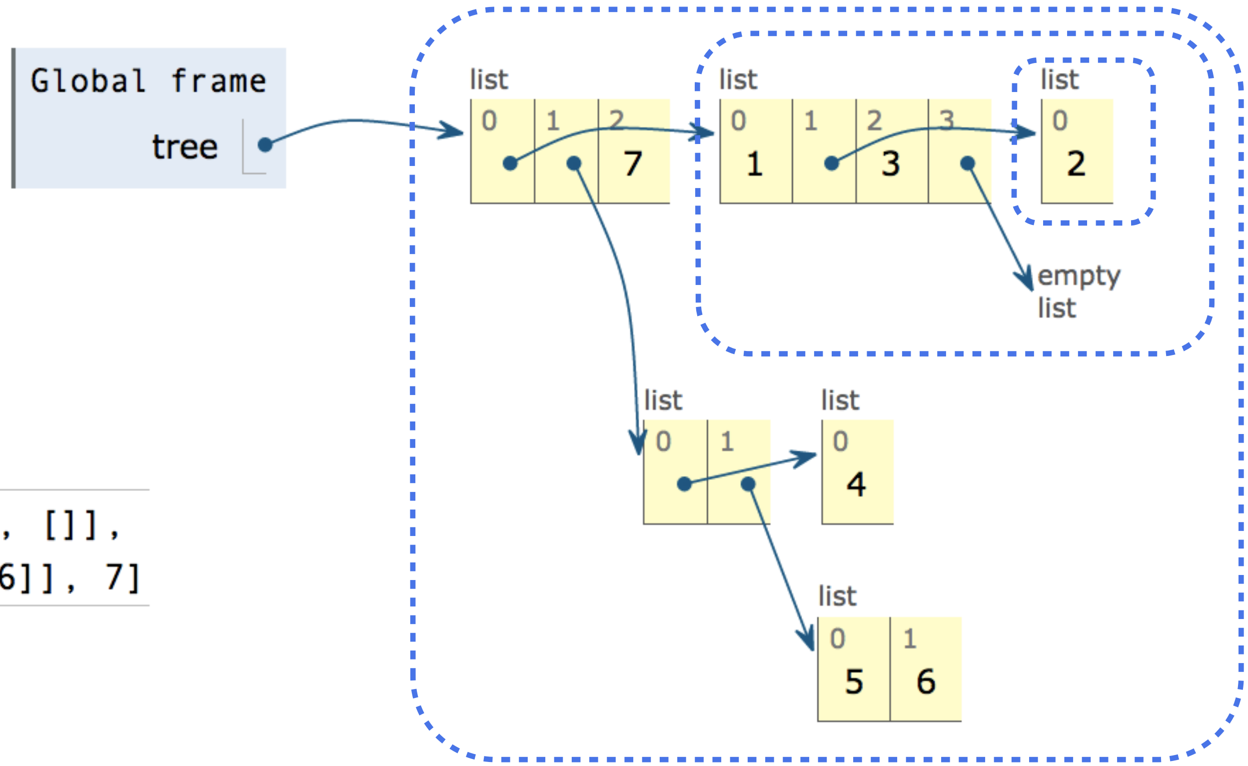



```
tree = [[1, [2], 3, []],
        [[4], [5, 6]], 7]
```

Trees are Nested Sequences

A **tree** is either a single value called a ***leaf*** or a sequence of ***trees***

Typically, some type restriction is placed on the leaves. E.g., a tree of numbers:

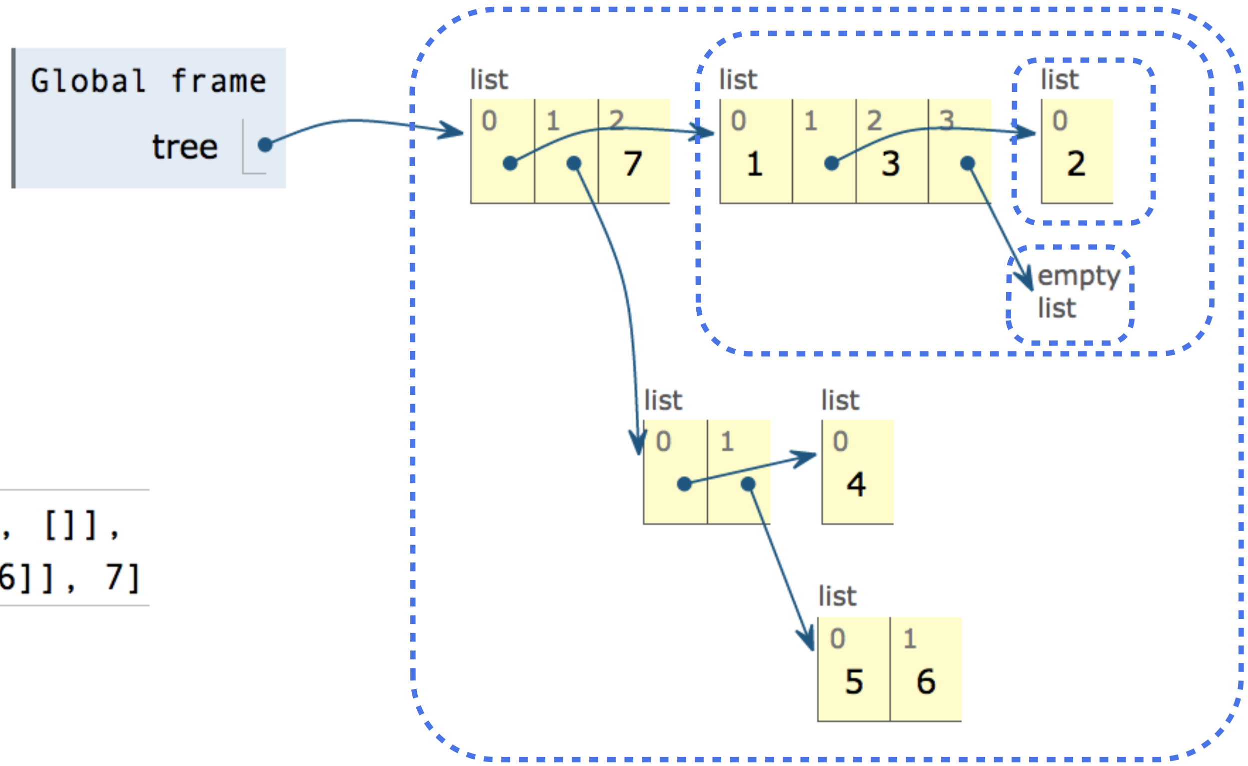



```
tree = [[1, [2], 3, []],
        [[4], [5, 6]], 7]
```

Trees are Nested Sequences

A **tree** is either a single value called a ***leaf*** or a sequence of ***trees***

Typically, some type restriction is placed on the leaves. E.g., a tree of numbers:

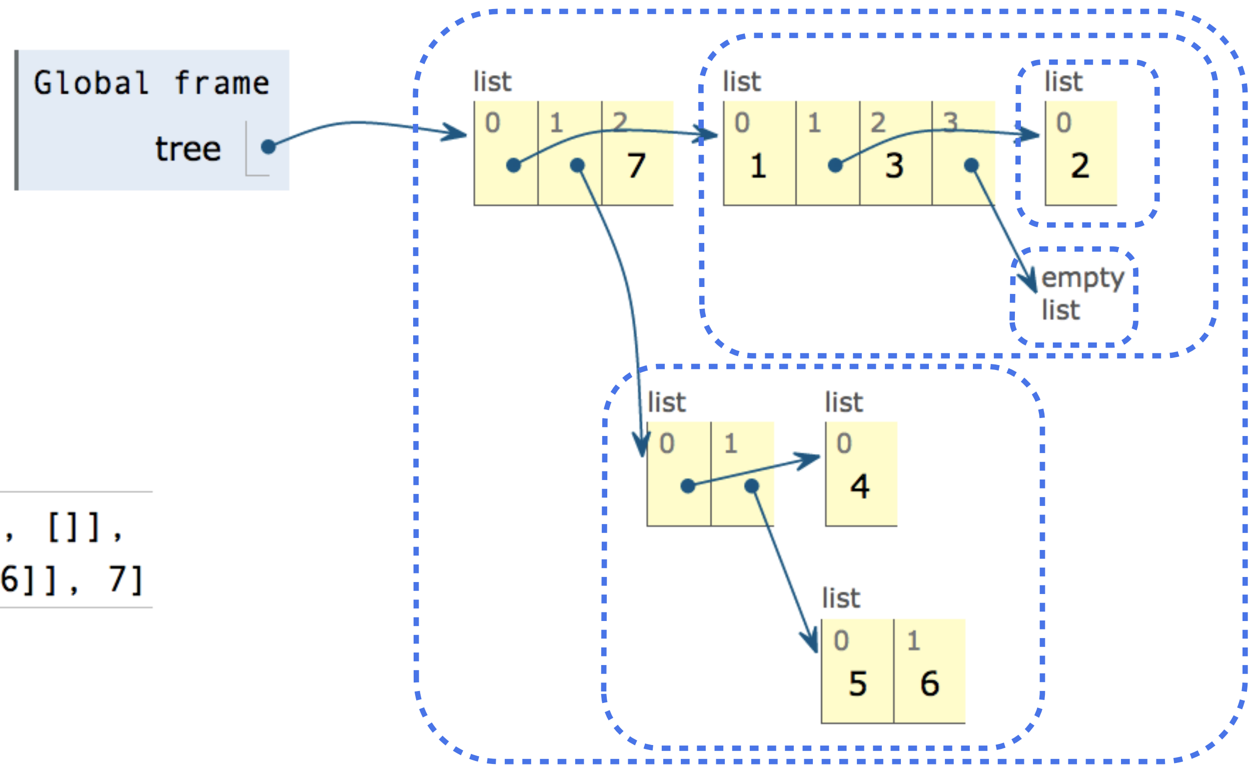



```
tree = [[1, [2], 3, []],
        [[4], [5, 6]], 7]
```

Trees are Nested Sequences

A **tree** is either a single value called a ***leaf*** or a sequence of ***trees***

Typically, some type restriction is placed on the leaves. E.g., a tree of numbers:

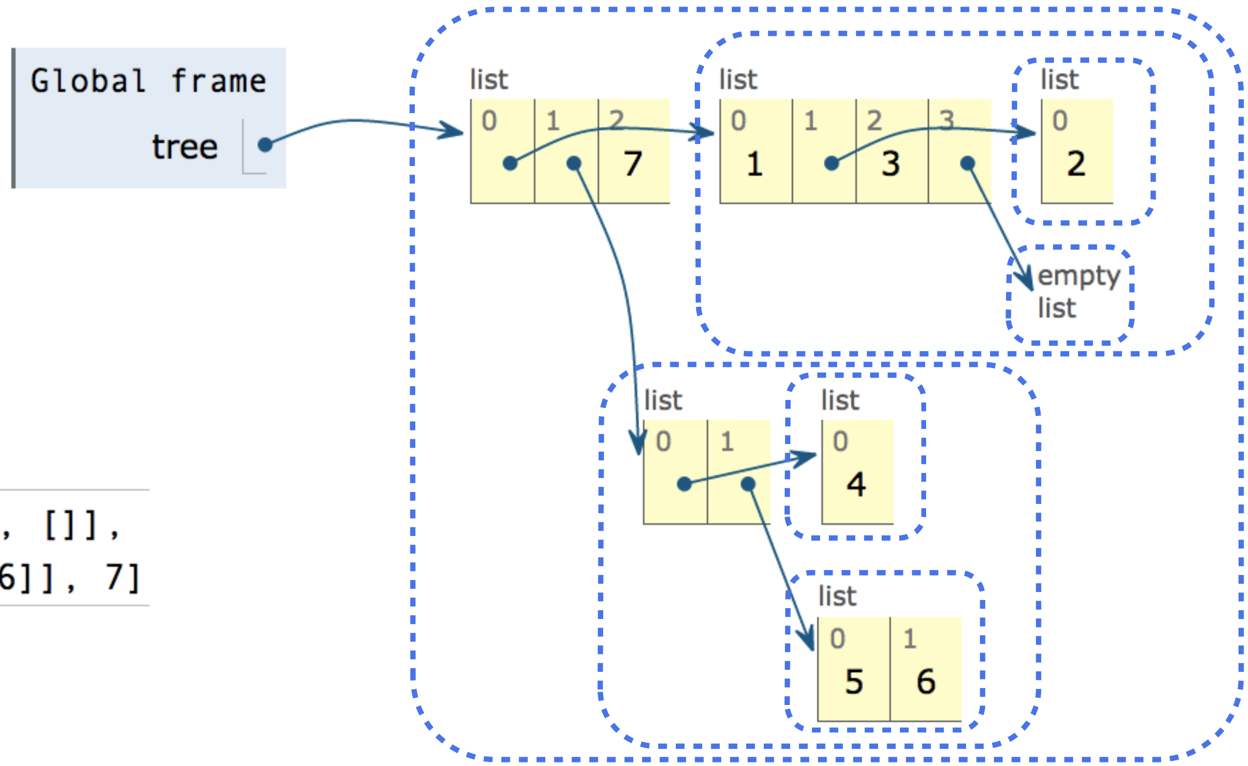



```
tree = [[1, [2], 3, []],
        [[4], [5, 6]], 7]
```

Trees are Nested Sequences

A **tree** is either a single value called a ***leaf*** or a sequence of ***trees***

Typically, some type restriction is placed on the leaves. E.g., a tree of numbers:

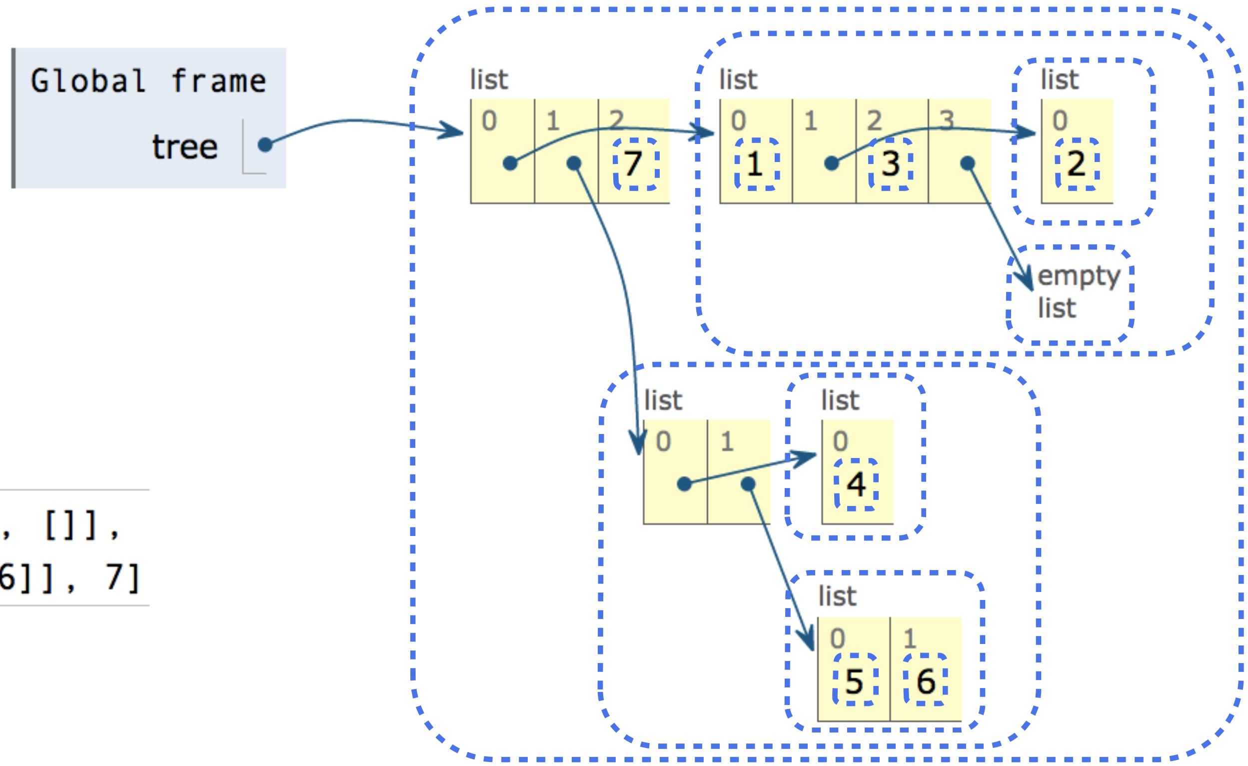



```
1  tree = [[1, [2], 3, []],
2            [[4], [5, 6]], 7]
```

Trees are Nested Sequences

A **tree** is either a single value called a ***leaf*** or a sequence of ***trees***

Typically, some type restriction is placed on the leaves. E.g., a tree of numbers:




```
1  tree = [[1, [2], 3, []],
2          [[4], [5, 6]], 7]
```

Tree Processing Uses Recursion

(Demo)

Tree Processing Uses Recursion

(Demo)

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

Tree Processing Uses Recursion

(Demo)

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

```
def count_leaves(tree):
    """Count the leaves of a tree."""
```

Tree Processing Uses Recursion

(Demo)

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

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

Tree Processing Uses Recursion

(Demo)

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

The recursive case often makes a recursive call on each branch and then aggregates

```
def count_leaves(tree):
    """Count the leaves of a tree."""
    if is_leaf(tree):
        return 1
    else:
        branch_counts = [count_leaves(b) for b in tree]
```

Tree Processing Uses Recursion

(Demo)

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

The recursive case often makes a recursive call on each branch and then aggregates

```
def count_leaves(tree):
    """Count the leaves of a tree."""
    if is_leaf(tree):
        return 1
    else:
        branch_counts = [count_leaves(b) for b in tree]
        return sum(branch_counts)
```

Discussion Question

Discussion Question

Complete the definition of flatten, which takes a tree and returns a list of its leaves

Discussion Question

Complete the definition of flatten, which takes a tree and returns a list of its leaves

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

    >>> tree = [[1, [2], 3, []], [[4], [5, 6]], 7]
    >>> flatten(tree)
    [1, 2, 3, 4, 5, 6, 7]
    """
```

Discussion Question

Complete the definition of flatten, which takes a tree and returns a list of its leaves

Hint: If you sum a sequence of lists, you get 1 list containing the elements of those lists

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

    >>> tree = [[1, [2], 3, []], [[4], [5, 6]], 7]
    >>> flatten(tree)
    [1, 2, 3, 4, 5, 6, 7]
    """
```

Discussion Question

Complete the definition of flatten, which takes a tree and returns a list of its leaves

Hint: If you sum a sequence of lists, you get 1 list containing the elements of those lists

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

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

    >>> tree = [[1, [2], 3, []], [[4], [5, 6]], 7]
    >>> flatten(tree)
    [1, 2, 3, 4, 5, 6, 7]
    """
```

Discussion Question

Complete the definition of flatten, which takes a tree and returns a list of its leaves

Hint: If you sum a sequence of lists, you get 1 list containing the elements of those lists

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

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

    >>> tree = [[1, [2], 3, []], [[4], [5, 6]], 7]
    >>> flatten(tree)
    [1, 2, 3, 4, 5, 6, 7]
    """
```

Discussion Question

Complete the definition of flatten, which takes a tree and returns a list of its leaves

Hint: If you sum a sequence of lists, you get 1 list containing the elements of those lists

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

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

    >>> tree = [[1, [2], 3, []], [[4], [5, 6]], 7]
    >>> flatten(tree)
    [1, 2, 3, 4, 5, 6, 7]
    """
```

Discussion Question

Complete the definition of flatten, which takes a tree and returns a list of its leaves

Hint: If you sum a sequence of lists, you get 1 list containing the elements of those lists

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

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

    >>> tree = [[1, [2], 3, []], [[4], [5, 6]], 7]
    >>> flatten(tree)
    [1, 2, 3, 4, 5, 6, 7]
    """
```

Discussion Question

Complete the definition of flatten, which takes a tree and returns a list of its leaves

Hint: If you sum a sequence of lists, you get 1 list containing the elements of those lists

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

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

    >>> tree = [[1, [2], 3, []], [[4], [5, 6]], 7]
    >>> flatten(tree)
    [1, 2, 3, 4, 5, 6, 7]
    """
```

Discussion Question

Complete the definition of flatten, which takes a tree and returns a list of its leaves

Hint: If you sum a sequence of lists, you get 1 list containing the elements of those lists

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

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

    >>> tree = [[1, [2], 3, []], [[4], [5, 6]], 7]
    >>> flatten(tree)
    [1, 2, 3, 4, 5, 6, 7]
    """
```

Discussion Question

Complete the definition of flatten, which takes a tree and returns a list of its leaves

Hint: If you sum a sequence of lists, you get 1 list containing the elements of those lists

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

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

    >>> tree = [[1, [2], 3, []], [[4], [5, 6]], 7]
    >>> flatten(tree)
    [1, 2, 3, 4, 5, 6, 7]
    """
    if is_leaf(tree):
        return [tree]
    else:
        return ______________________________

def is_leaf(tree):
    return type(tree) != list
```

Discussion Question

Complete the definition of flatten, which takes a tree and returns a list of its leaves

Hint: If you sum a sequence of lists, you get 1 list containing the elements of those lists

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

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

    >>> tree = [[1, [2], 3, []], [[4], [5, 6]], 7]
    >>> flatten(tree)
    [1, 2, 3, 4, 5, 6, 7]
    """
    if is_leaf(tree):
        return [tree]
    else:
        return sum([flatten(b) for b in tree], [])

def is_leaf(tree):
    return type(tree) != list
```

Sequence Operations

Membership & Slicing

Python sequences have operators for membership and slicing

Membership & Slicing

Python sequences have operators for membership and slicing

Membership.

Membership & Slicing

Python sequences have operators for membership and slicing

Membership.

```
>>> digits = [1, 8, 2, 8]
>>> 2 in digits
True
>>> 1828 not in digits
True
```

Membership & Slicing

Python sequences have operators for membership and slicing

Membership.

```
>>> digits = [1, 8, 2, 8]
>>> 2 in digits
True
>>> 1828 not in digits
True
```

Slicing.

Membership & Slicing

Python sequences have operators for membership and slicing

Membership.

```
>>> digits = [1, 8, 2, 8]
>>> 2 in digits
True
>>> 1828 not in digits
True
```

Slicing.

```
>>> digits[0:2]
[1, 8]
>>> digits[1:]
[8, 2, 8]
```

Membership & Slicing

Python sequences have operators for membership and slicing

Membership.

```
>>> digits = [1, 8, 2, 8]
>>> 2 in digits
True
>>> 1828 not in digits
True
```

Slicing.

```
>>> digits[0:2]
[1, 8]
>>> digits[1:]
[8, 2, 8]
```

Slicing creates a new object

Membership & Slicing

Python sequences have operators for membership and slicing

Membership.

```
>>> digits = [1, 8, 2, 8]
>>> 2 in digits
True
>>> 1828 not in digits
True
```

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

Slicing.

```
>>> digits[0:2]
[1, 8]
>>> digits[1:]
[8, 2, 8]
```

Slicing creates a new object

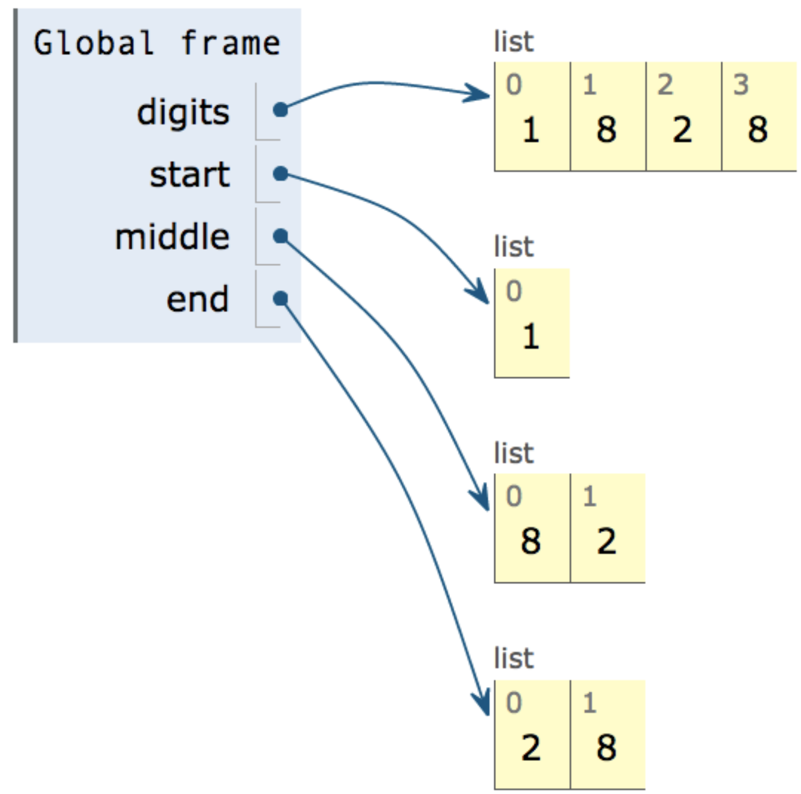
Membership & Slicing

Python sequences have operators for membership and slicing

Membership.

```
>>> digits = [1, 8, 2, 8]
>>> 2 in digits
True
>>> 1828 not in digits
True
```

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



Slicing.

```
>>> digits[0:2]
[1, 8]
>>> digits[1:]
[8, 2, 8]
```

Slicing creates a new object

Binary Trees

Binary Trees

Trees may also have restrictions on their structure

Binary Trees

Trees may also have restrictions on their structure

A **binary tree** is either a ***leaf*** or a sequence containing at most two ***binary trees***

Binary Trees

Trees may also have restrictions on their structure

A **binary tree** is either a ***leaf*** or a sequence containing at most two ***binary trees***

The process of transforming a tree into a binary tree is called ***binarization***

Binary Trees

Trees may also have restrictions on their structure

A **binary tree** is either a ***leaf*** or a sequence containing at most two ***binary trees***

The process of transforming a tree into a binary tree is called ***binarization***

```
def right_binarize(tree):
    """Construct a right-branching binary tree.
```

Binary Trees

Trees may also have restrictions on their structure

A **binary tree** is either a ***leaf*** or a sequence containing at most two ***binary trees***

The process of transforming a tree into a binary tree is called ***binarization***

```
def right_binarize(tree):
    """Construct a right-branching binary tree.

    >>> right_binarize([1, 2, 3, 4, 5, 6, 7])
    [1, [2, [3, [4, [5, [6, 7]]]]]]
    """
```

Binary Trees

Trees may also have restrictions on their structure

A **binary tree** is either a ***leaf*** or a sequence containing at most two ***binary trees***

The process of transforming a tree into a binary tree is called ***binarization***

```
def right_binarize(tree):
    """Construct a right-branching binary tree.

    >>> right_binarize([1, 2, 3, 4, 5, 6, 7])
    [1, [2, [3, [4, [5, [6, 7]]]]]]
    """
    if is_leaf(tree):
        return tree
```

Binary Trees

Trees may also have restrictions on their structure

A **binary tree** is either a ***leaf*** or a sequence containing at most two ***binary trees***

The process of transforming a tree into a binary tree is called ***binarization***

```
def right_binarize(tree):
    """Construct a right-branching binary tree.

    >>> right_binarize([1, 2, 3, 4, 5, 6, 7])
    [1, [2, [3, [4, [5, [6, 7]]]]]]
    """
    if is_leaf(tree):
        return tree
    if len(tree) > 2:
        tree = [tree[0], tree[1:]]
```

Binary Trees

Trees may also have restrictions on their structure

A **binary tree** is either a ***leaf*** or a sequence containing at most two ***binary trees***

The process of transforming a tree into a binary tree is called ***binarization***

```
def right_binarize(tree):
    """Construct a right-branching binary tree.

    >>> right_binarize([1, 2, 3, 4, 5, 6, 7])
    [1, [2, [3, [4, [5, [6, 7]]]]]]
    """
    if is_leaf(tree):
        return tree
    if len(tree) > 2:
        tree = [tree[0], tree[1:]]
```

All but the first branch are grouped into a new branch

Binary Trees

Trees may also have restrictions on their structure

A **binary tree** is either a ***leaf*** or a sequence containing at most two ***binary trees***

The process of transforming a tree into a binary tree is called ***binarization***

```
def right_binarize(tree):
    """Construct a right-branching binary tree.

    >>> right_binarize([1, 2, 3, 4, 5, 6, 7])
    [1, [2, [3, [4, [5, [6, 7]]]]]]
    """
    if is_leaf(tree):
        return tree
    if len(tree) > 2:
        tree = [tree[0], tree[1:]]
    return [right_binarize(b) for b in tree]
```

All but the first branch are grouped into a new branch

Binary Trees

Trees may also have restrictions on their structure

A **binary tree** is either a ***leaf*** or a sequence containing at most two ***binary trees***

The process of transforming a tree into a binary tree is called ***binarization***

```
def right_binarize(tree):
    """Construct a right-branching binary tree.

    >>> right_binarize([1, 2, 3, 4, 5, 6, 7])
    [1, [2, [3, [4, [5, [6, 7]]]]]]
    """
    if is_leaf(tree):
        return tree
    if len(tree) > 2:
        tree = [tree[0], tree[1:]]
    return [right_binarize(b) for b in tree]
```

All but the first branch are grouped into a new branch

(Demo)

Strings

Strings are an Abstraction

Strings are an Abstraction

Representing data:

```
'200'      '1.2e-5'      'False'      '(1, 2)'
```

Strings are an Abstraction

Representing data:

```
'200'      '1.2e-5'      'False'      '(1, 2)'
```

Representing language:

```
"""And, as imagination bodies forth
The forms of things to unknown, and the poet's pen
Turns them to shapes, and gives to airy nothing
A local habitation and a name.
"""
```

Strings are an Abstraction

Representing data:

```
'200'      '1.2e-5'      'False'      '(1, 2)'
```

Representing language:

```
"""And, as imagination bodies forth
The forms of things to unknown, and the poet's pen
Turns them to shapes, and gives to airy nothing
A local habitation and a name.
"""
```

Representing programs:

```
'curry = lambda f: lambda x: lambda y: f(x, y)'
```

Strings are an Abstraction

Representing data:

```
'200'      '1.2e-5'      'False'      '(1, 2)'
```

Representing language:

```
"""And, as imagination bodies forth
The forms of things to unknown, and the poet's pen
Turns them to shapes, and gives to airy nothing
A local habitation and a name.
"""
```

Representing programs:

```
'curry = lambda f: lambda x: lambda y: f(x, y)'
```

(Demo)

String Literals Have Three Forms

```
>>> 'I am string!'
'I am string!'

>>> "I've got an apostrophe"
"I've got an apostrophe"

>>> '您好'
'您好'
```

String Literals Have Three Forms

```
>>> 'I am string!'
'I am string!'

>>> "I've got an apostrophe"
"I've got an apostrophe"

>>> '您好'
'您好'
```

Single-quoted and double-quoted strings are equivalent

String Literals Have Three Forms

```
>>> 'I am string!'
'I am string!'

>>> "I've got an apostrophe"
"I've got an apostrophe"

>>> '您好'
'您好'


>>> """The Zen of Python
claims, Readability counts.
Read more: import this."""
'The Zen of Python\nclaims, Readability counts.\nRead more: import this.'
```

Single-quoted and double-quoted strings are equivalent

String Literals Have Three Forms

```
>>> 'I am string!'
'I am string!'

>>> "I've got an apostrophe"
"I've got an apostrophe"

>>> '您好'
'您好'


>>> """The Zen of Python
claims, Readability counts.
Read more: import this."""
'The Zen of Python\nclaims, Readability counts.\nRead more: import this.'
```

Single-quoted and double-quoted strings are equivalent

A backslash "escapes" the following character

String Literals Have Three Forms

```
>>> 'I am string!'
'I am string!'

>>> "I've got an apostrophe"
"I've got an apostrophe"

>>> '您好'
'您好'


>>> """The Zen of Python
claims, Readability counts.
Read more: import this."""
'The Zen of Python\nclaims, Readability counts.\nRead more: import this.'
```

Single-quoted and double-quoted strings are equivalent

A backslash "escapes" the following character

"Line feed" character represents a new line

Strings are Sequences

Strings are Sequences

Length and element selection are similar to all sequences

Strings are Sequences

Length and element selection are similar to all sequences

```
>>> city = 'Berkeley'
>>> len(city)
8
>>> city[3]
'k'
```

Strings are Sequences

Length and element selection are similar to all sequences

```
>>> city = 'Berkeley'
>>> len(city)
8
>>> city[3]
'k'
```

Careful: An element of a string is itself a string, but with only one element!

Strings are Sequences

Length and element selection are similar to all sequences

```
>>> city = 'Berkeley'
>>> len(city)
8
>>> city[3]
'k'
```

Careful: An element of a string is itself a string, but with only one element!

However, the "in" and "not in" operators match substrings

Strings are Sequences

Length and element selection are similar to all sequences

```
>>> city = 'Berkeley'
>>> len(city)
8
>>> city[3]
'k'
```

Careful: An element of a string is itself a string, but with only one element!

However, the "in" and "not in" operators match substrings

```
>>> 'here' in "Where's Waldo?"
True
>>> 234 in [1, 2, 3, 4, 5]
False
>>> [2, 3, 4] in [1, 2, 3, 4, 5]
False
```

Strings are Sequences

Length and element selection are similar to all sequences

```
>>> city = 'Berkeley'
>>> len(city)
8
>>> city[3]
'k'
```

Careful: An element of a string is itself a string, but with only one element!

However, the "in" and "not in" operators match substrings

```
>>> 'here' in "Where's Waldo?"
True
>>> 234 in [1, 2, 3, 4, 5]
False
>>> [2, 3, 4] in [1, 2, 3, 4, 5]
False
```

When working with strings, we usually care about whole words more than letters

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