Box-and-Pointer Notation

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

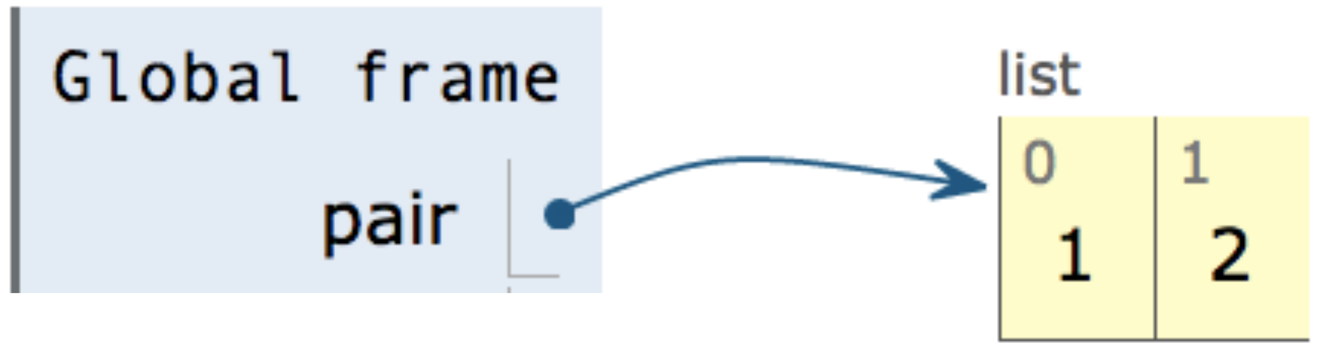
- Hierarchical structures are made up of parts, which themselves are made up of parts, and so on

Lists can contain lists as elements (in addition to anything else)

Box-and-Pointer Notation in Environment Diagrams

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

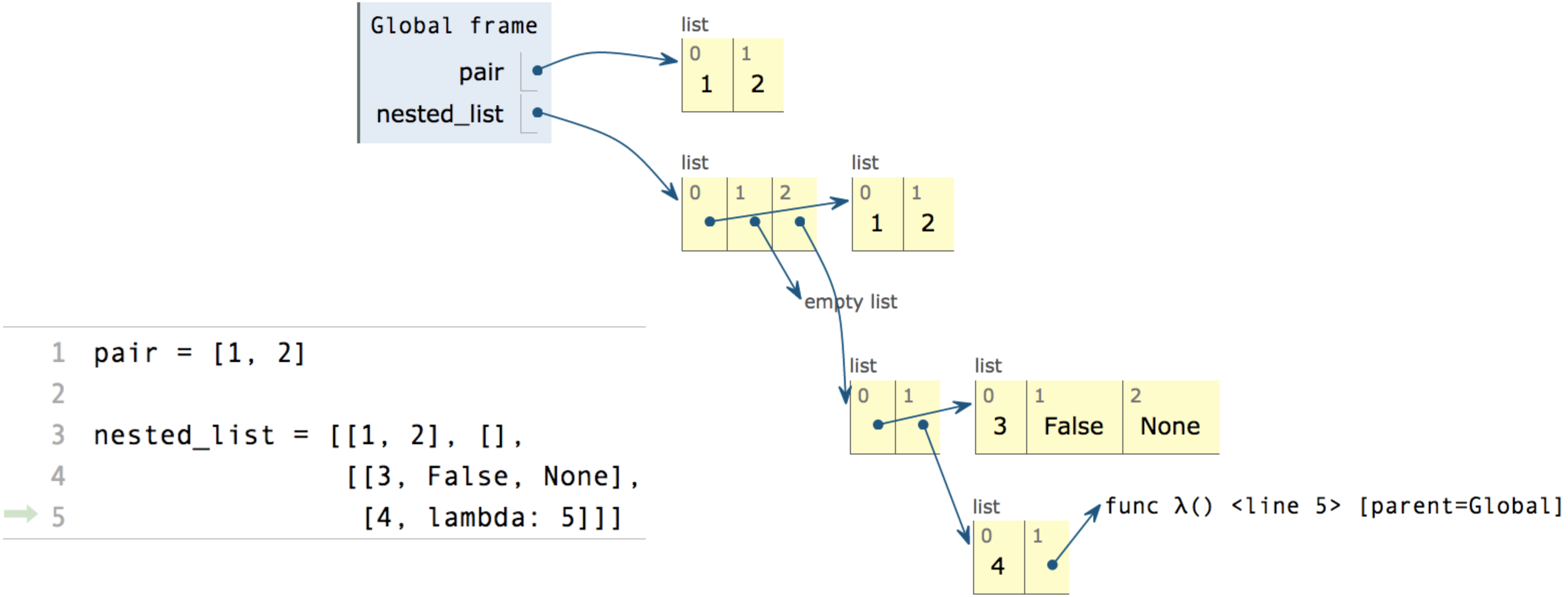


```
pair = [1, 2]
```

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



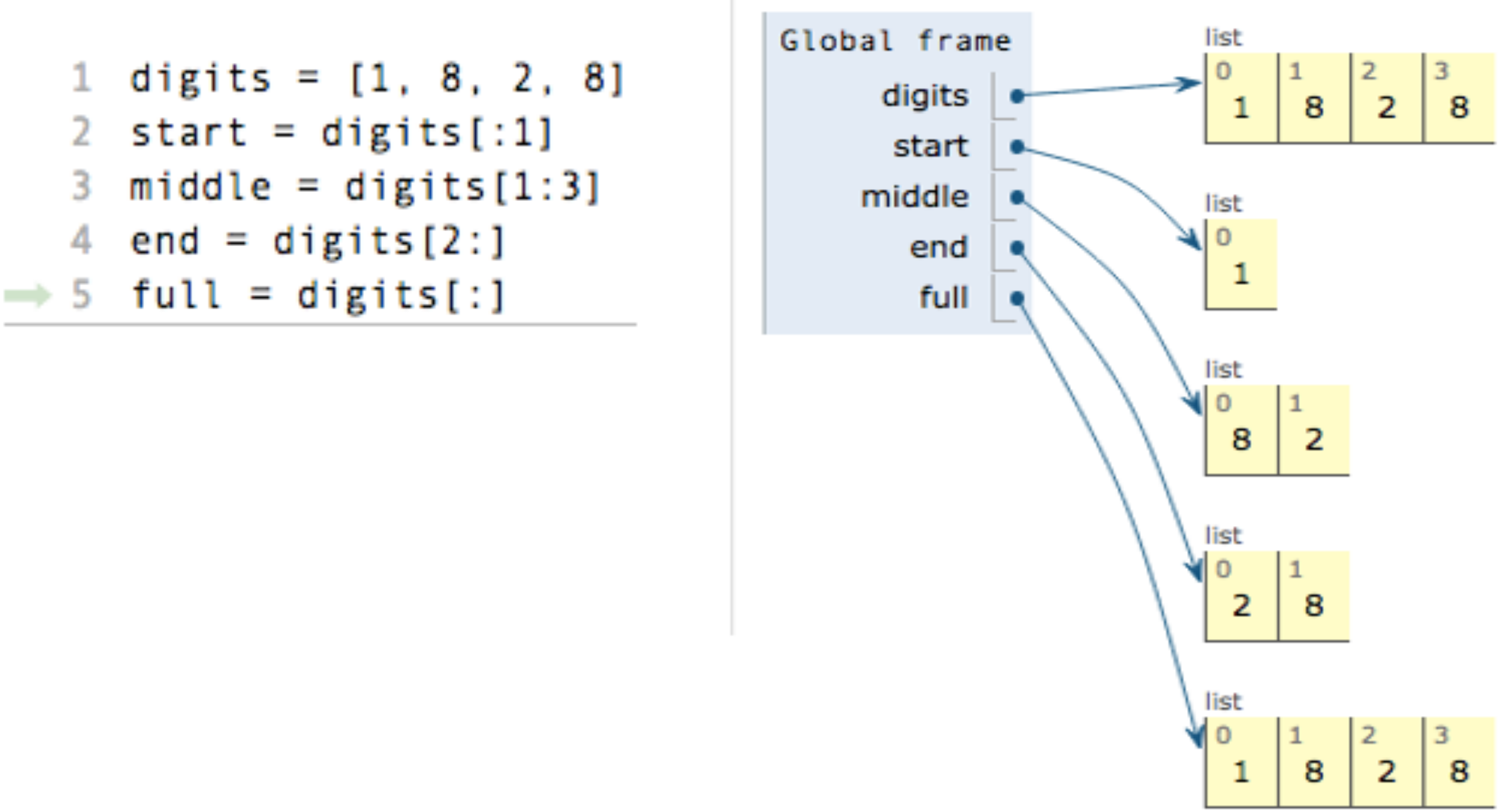
```
pair = [1, 2]

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

pythontutor.com/composingprograms.html#code=pair%20%3D%20[1,%202]%0A%0Anested_list%20%3D%20[[1,%202],%20[],%0A%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20[[3,%20False,%20None],%0A%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20%20[4,%20

Slicing

`(Demo1)`

Slicing Creates New Values



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

pythontutor.com/composingprograms.html#code=digits%20%3D%20[1,%208,%202,%208]%0Astart%20%3D%20digits[%3A1]%0Amiddle%20%3D%20digits[1%3A3]%0Aend%20%3D%20digits[2%3A]%0Afull%20%3D%20digits[%3A]&cumulative%3Dtrue&curInstr%3D5&mode=display&origin=composingprograms.js&py=3&rawInputLstJSON=[]

Processing Container Values

Sequence Aggregation (Demo2 after each one)

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

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

  ```
  Return the sum of an iterable (not of 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

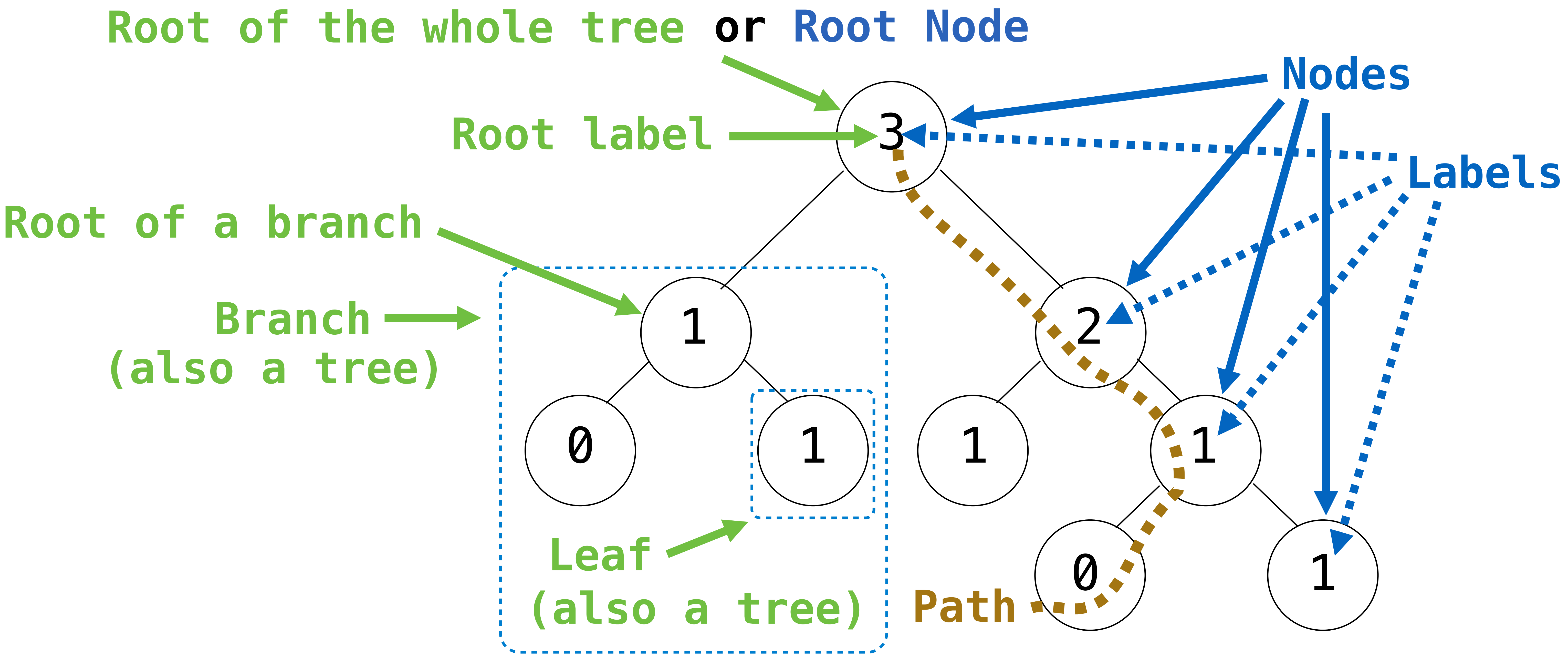
  ```
  With a single iterable argument, return its largest item.
  With two or more arguments, return the largest argument.
  ```

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

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

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

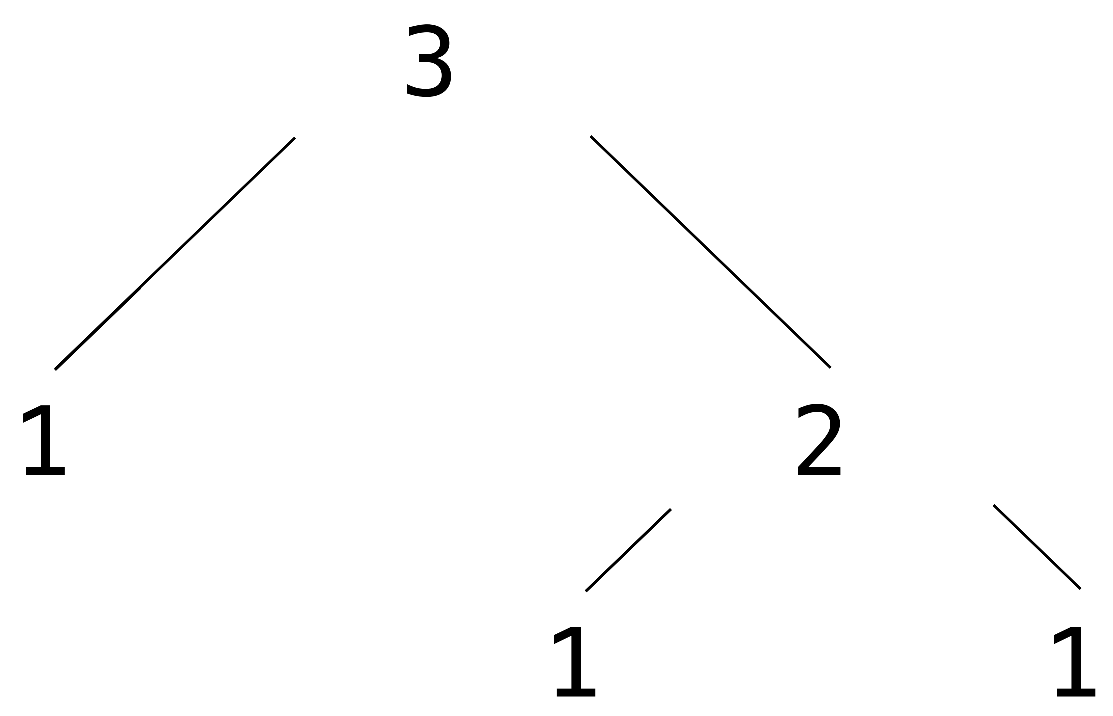
The top node is the **root node**

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

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)
```

Verifies the tree definition

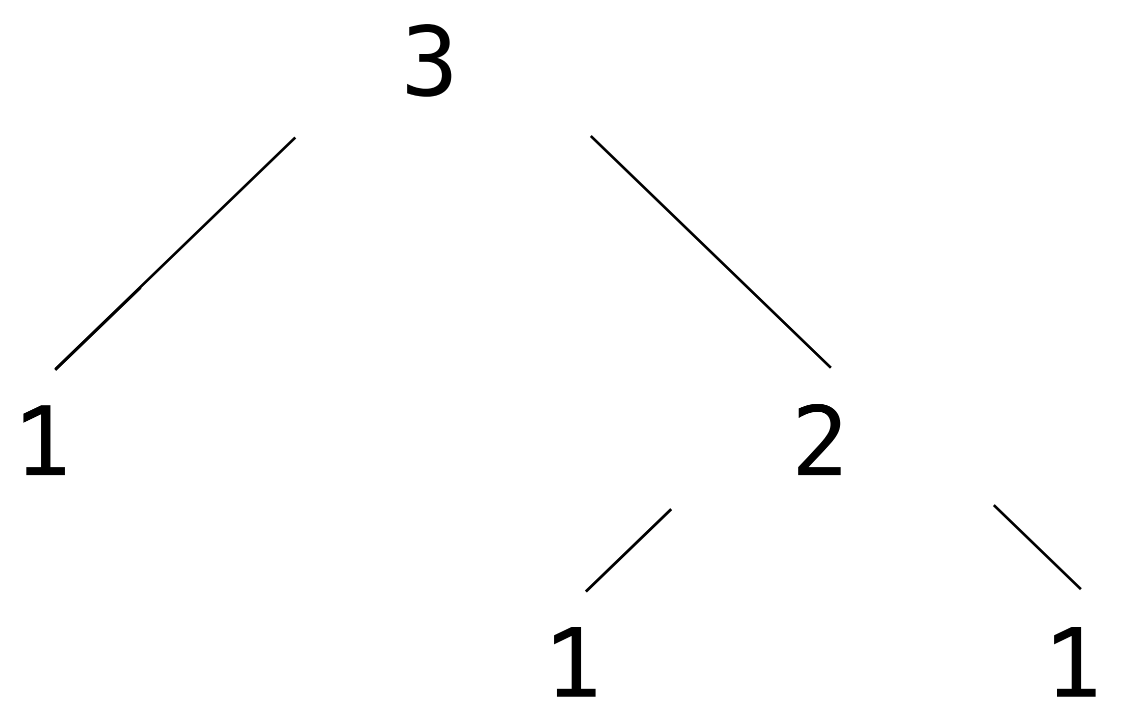
Creates a list from a sequence of branches

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

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

Verifies that tree is bound to a list

```
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
```

- 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)
```

(Demo3)

Example: Printing Trees

`(Demo4a)`

Tree Processing

`(Demo4)`

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)
```

(Demo5)

Discussion Question

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

Hint: If you `sum` a list of lists, you get a 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 leaves(tree):
    """Return a list containing the leaf labels of tree.

    >>> leaves(fib_tree(5))
    [1, 0, 1, 0, 1, 1, 0, 1]
    """
    if is_leaf(tree):
        return [label(tree)]
    else:
        return sum(List of leaf labels 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

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 is_leaf(t):
        return tree(label(t) + 1)
    else:
        bs = [increment_leaves(b) for b in branches(t)]
        return tree(label(t), bs)

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

def tree_map(t,f):
    """Return a tree like t but with all labels having f applied to them."""
    return tree(f(label(t)), [tree_map(b,f) for b in branches(t)])
```