Mutable Values

Today we'll cover...

- Tree creation algorithms
- Mutability vs. Immutability
- Mutable trees
- List mutations
- Identity and Equality

Trees

Tree: Layers of abstraction

Primitive Representation	`1 2 3 True False` `(..,..) [..,..] {...}`
Data abstraction	`tree() children() label()` `is_leaf()`
User program	`count_leaves(t)` `double(t)`

Each layer only uses the layer above it.

Abstractions involve choices

- What operations should be exposed?
- What should those operations be named?
- What are the parameters and return values?

Two possible `tree()` abstractions (of many):

This lecture	Your assignments
`tree(label, children=None)`	`tree(label, branches=[])`
`label(tree)`	`label(tree)`
`children(tree)`	`branches(tree)`

👀Can you spot the differences?

A tree() implementation

A number-list tuple for each tree/subtree:

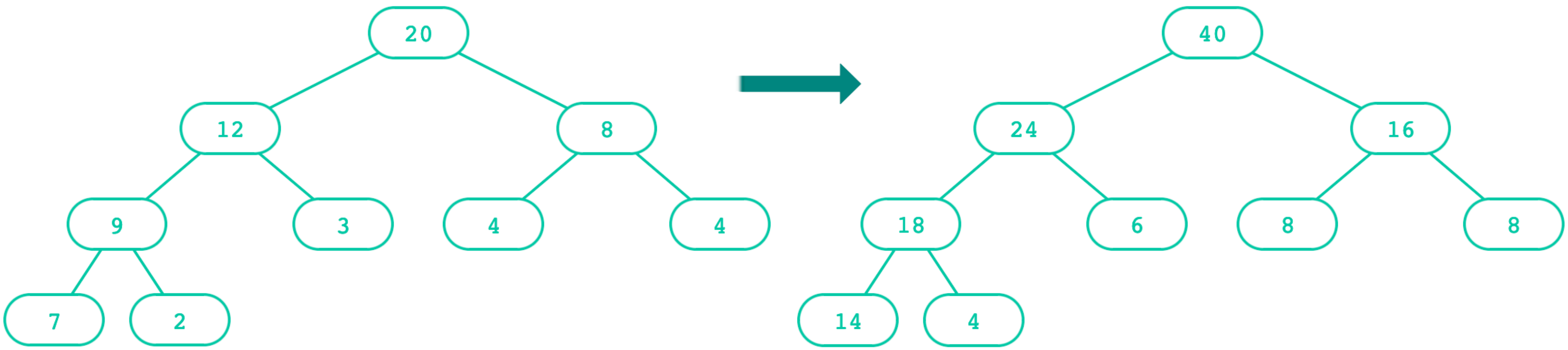
```
(20,[(12,[(9,[(7,[]),(2, [])]),(3, [])]),(8,[(4,[]),(4,[])])])
```

```
def tree(label, children=None):
    """ Creates a tree whose root node is labeled LABEL and
        optionally has CHILDREN, a list of trees."""
    return (label, list(children or []))

def label(tree):
    """ Returns the label of the root node of TREE. """
    return tree[0]

def children(tree):
    """ Returns a list of children of TREE. """
    return tree[1]
```

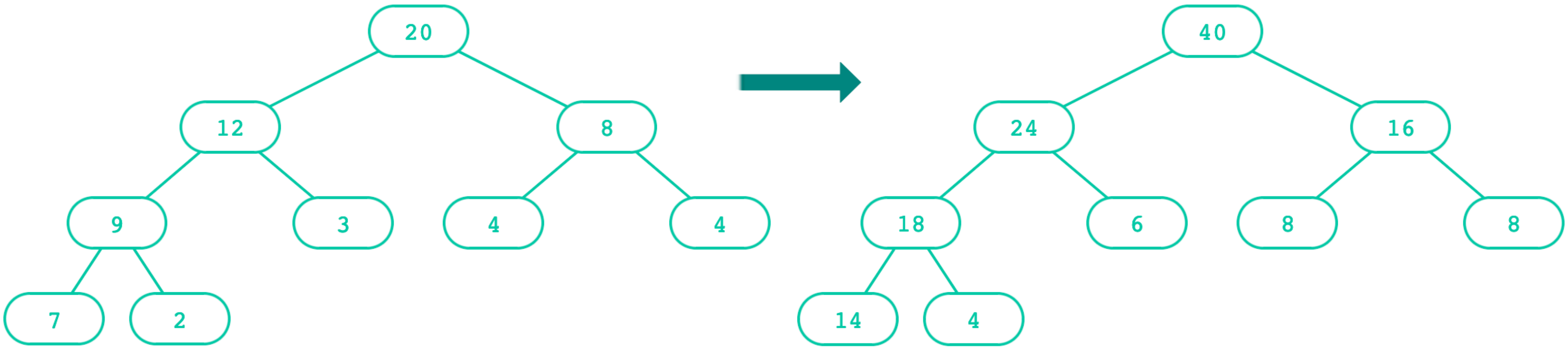
```
t = tree(20, [tree(12,
               [tree(9,
                 [tree(7), tree(2)]),
                tree(3)]),
              tree(8,
               [tree(4), tree(4)])])
```

Tree creation algorithms

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



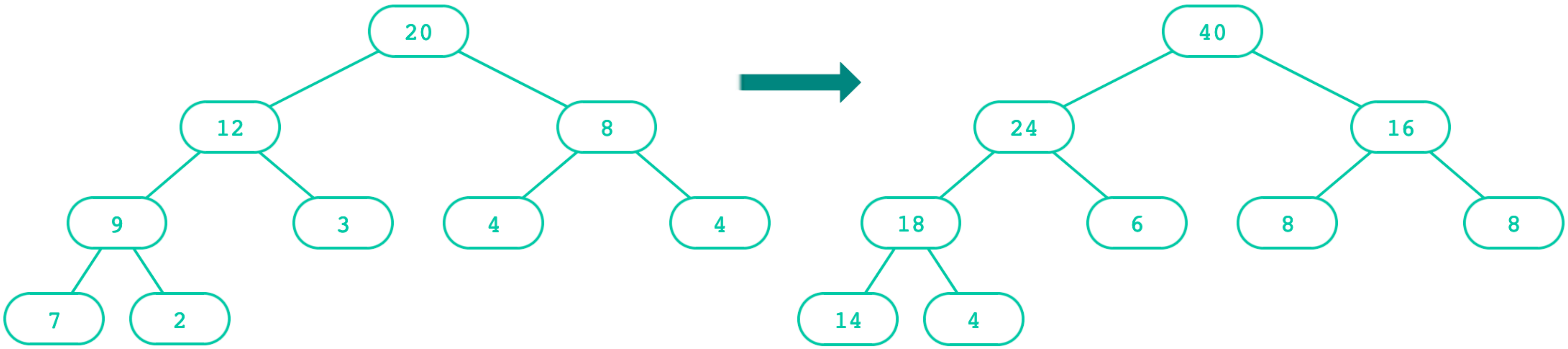
Tree creation: Doubling labels




```
def double(t):
    """Returns a tree identical to T, but with all labels doubled."""
    if

    else:
```

What's the base case? What's the recursive call?

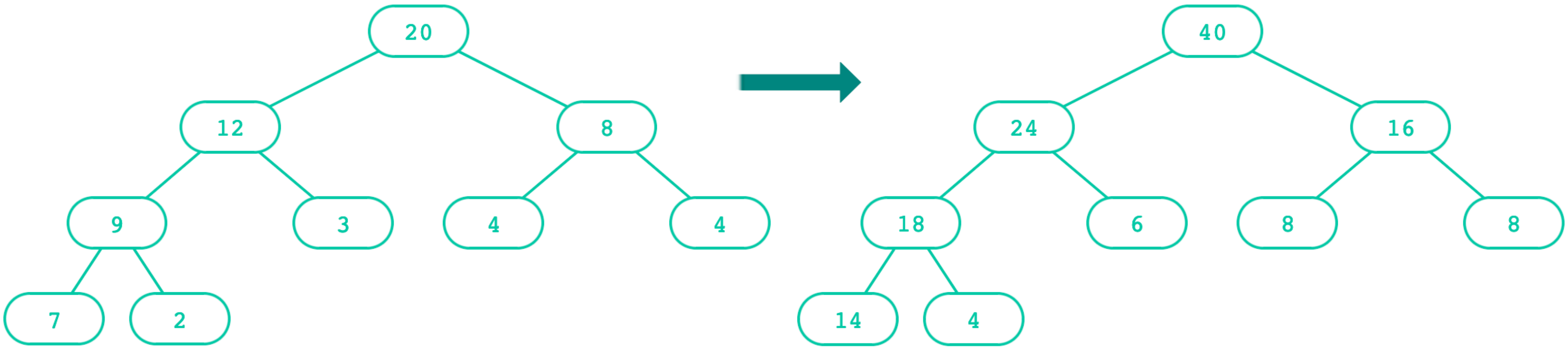
Tree creation: Doubling labels




```
def double(t):
    """Returns a tree identical to T, but with all labels doubled."""
    if is_leaf(t):

    else:
```

What's the base case? What's the recursive call?

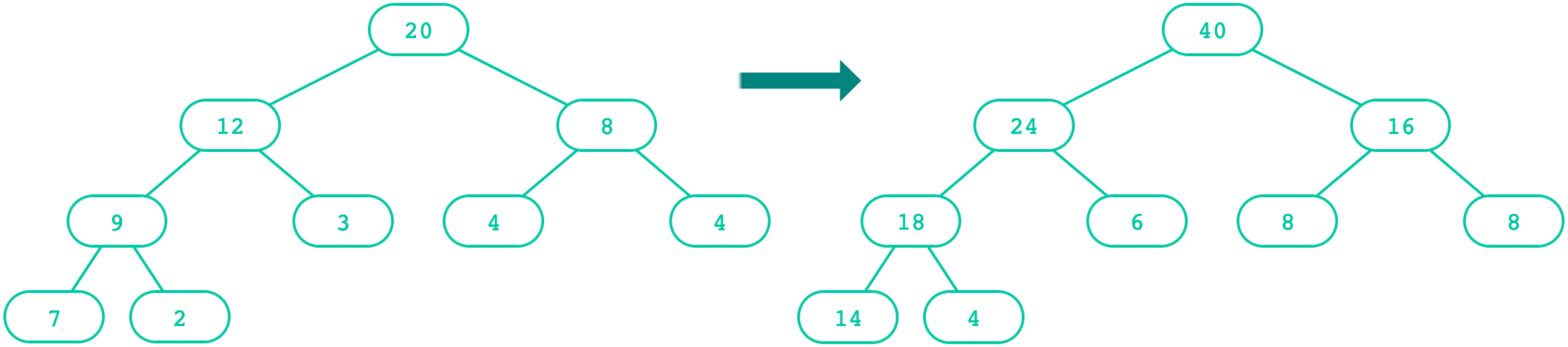
Tree creation: Doubling labels




```
def double(t):
    """Returns a tree identical to T, but with all labels doubled."""
    if is_leaf(t):
        return tree(label(t) * 2)
    else:
```

What's the base case? What's the recursive call?

Tree creation: Doubling labels




```
def double(t):
    """Returns a tree identical to T, but with all labels doubled."""
    if is_leaf(t):
        return tree(label(t) * 2)
    else:
        doubled_children = []
        for c in children(t):
            doubled_children += [double(c)]
        return tree(label(t) * 2, doubled_children)
```

What's the base case? What's the recursive call?

Creating trees: Doubling labels

How can we shorten this?

```
doubled_children = []
for c in children(t):
    doubled_children += [double(c)]
```

Creating trees: Doubling labels

How can we shorten this?

```
doubled_children = []
for c in children(t):
    doubled_children += [double(c)]
```

List comprehension!

```
def double(t):
    """Returns a tree identical to T, but with all labels doubled."""
    if is_leaf(t):
        return tree(label(t) * 2)
    else:
        return tree(label(t) * 2,
            [double(c) for c in children(t)])
```

Creating trees: Doubling labels

How can we shorten this?

```
doubled_children = []
for c in children(t):
    doubled_children += [double(c)]
```

List comprehension!

```
def double(t):
    """Returns a tree identical to T, but with all labels doubled."""
    if is_leaf(t):
        return tree(label(t) * 2)
    else:
        return tree(label(t) * 2,
            [double(c) for c in children(t)])
```

Even shorter!

```
def double(t):
    """Returns the number of leaf nodes in T."""
    return tree(label(t) * 2,
            [double(c) for c in children(t)])
```

Mutation

Non-destructive 🏛 vs. Destructive 💥

A non-destructive operation:

```
>>> aThing
<output A>
>>> <operation on aThing (that obey abstraction boundaries)>
>>> aThing
<output A>
```

A is never changed by the operation. 🏛

A destructive operation:

```
>>> aThing
<output A>
>>> <operation on aThing (that obey abstraction boundaries)>
>>> aThing
<output B>
```

A and B don't always differ, but if they ever differ, it's destructive! 💥

Non-destructive 🏛️ vs. Destructive 💥

```
def double(t):
    """Returns the number of leaf nodes in T."""
    return tree(label(t) * 2,
            [double(c) for c in children(t)])
```

Is `double(t)`...

- destructive?
- non-destructive?

Non-destructive 🏛️ vs. Destructive 💥

```
def double(t):
    """Returns the number of leaf nodes in T."""
    return tree(label(t) * 2,
            [double(c) for c in children(t)])
```

Is `double(t)`...

- destructive?
- non-destructive? ⬅

`double(t)` did not mutate the original input data, so it is considered a **non-destructive** operation.

Immutability vs. Mutability

An **immutable** value is unchanging once created.

Immutable types (that we've covered): int, float, string, tuple

```
a_tuple = (1, 2)
a_tuple[0] = 3
a_string = "Hi y'all"
a_string[1] = "I"
a_string += ", how you doing?"
an_int = 20
an_int += 2
```

A **mutable** value can change in value throughout the course of computation. All names that refer to the same object are affected by a mutation.

Mutable types (that we've covered): list, dict

```
grades = [90, 70, 85]
grades_copy = grades
grades[1] = 100
words = {"agua": "water"}
words["pavo"] = "turkey"
```

Immutability vs. Mutability

An **immutable** value is unchanging once created.

Immutable types (that we've covered): int, float, string, tuple

```
a_tuple = (1, 2)
a_tuple[0] = 3                  # 🚫 Error! Tuple items cannot be set.
a_string = "Hi y'all"
a_string[1] = "I"               # 🚫 Error! String elements cannot be set.
a_string += ", how you doing?"
an_int = 20
an_int += 2
```

A **mutable** value can change in value throughout the course of computation. All names that refer to the same object are affected by a mutation.

Mutable types (that we've covered): list, dict

```
grades = [90, 70, 85]
grades_copy = grades
grades[1] = 100
words = {"agua": "water"}
words["pavo"] = "turkey"
```

Immutability vs. Mutability

An **immutable** value is unchanging once created.

Immutable types (that we've covered): int, float, string, tuple

```
a_tuple = (1, 2)
a_tuple[0] = 3                 # 🚫 Error! Tuple items cannot be set.
a_string = "Hi y'all"
a_string[1] = "I"              # 🚫 Error! String elements cannot be set.
a_string += ", how you doing?"  # 🤔 How does this work?
an_int = 20
an_int += 2                    # 🤔 And this?
```

A **mutable** value can change in value throughout the course of computation. All names that refer to the same object are affected by a mutation.

Mutable types (that we've covered): list, dict

```
grades = [90, 70, 85]
grades_copy = grades
grades[1] = 100
words = {"agua": "water"}
words["pavo"] = "turkey"
```

Mutation in function calls

An function can change the value of any object in its scope.

```
four = [1, 2, 3, 4]
print(four[0])
do_stuff_to(four)
print(four[0])
```

Try in PythonTutor

Even without arguments:

```
four = [1, 2, 3, 4]
print(four[3])
do_other_stuff()
print(four[3])
```

Try in PythonTutor

Mutables inside immutables

An immutable sequence may still change if it contains a mutable value as an element.

```
t = (1, [2, 3])
t[1][0] = 99
t[1][1] = "Problems"
```

Try in PythonTutor

Immutability vs. Mutability

```
def tree(label, children=None):
    """ Creates a tree whose root node is labeled LABEL and
        optionally has CHILDREN, a list of trees."""
    return (label, list(children or []))

def label(tree):
    """ Returns the label of the root node of TREE. """
    return tree[0]

def children(tree):
    """ Returns a list of children of TREE. """
    return tree[1]
```

Is `tree()`...

- mutable?
- immutable?

Immutability vs. Mutability

```
def tree(label, children=None):
    """ Creates a tree whose root node is labeled LABEL and
        optionally has CHILDREN, a list of trees."""
    return (label, list(children or []))

def label(tree):
    """ Returns the label of the root node of TREE. """
    return tree[0]

def children(tree):
    """ Returns a list of children of TREE. """
    return tree[1]
```

Is `tree()`...

- mutable?
- immutable? ⬅

Our current `tree()` abstraction is immutable, as long as we don't break the abstraction barrier. We **cannot** mutate a tree once it's created.

A mutable tree()?

Suppose we add two mutators to our abstraction:

```
def set_label(tree, label):
    """Sets the label of TREE's root node to LABEL"""


def set_children(tree, children):
    """Sets the children of TREE to CHILDREN, a list of trees."""
```

A mutable tree()?

Suppose we add two mutators to our abstraction:

```
def set_label(tree, label):
    """Sets the label of TREE's root node to LABEL"""
    tree[0] = label

def set_children(tree, children):
    """Sets the children of TREE to CHILDREN, a list of trees."""
    tree[1] = children
```

Will that work? Let's find out...

A mutable tree()?

Suppose we add two mutators to our abstraction:

```
def set_label(tree, label):
    """Sets the label of TREE's root node to LABEL"""
    tree[0] = label

def set_children(tree, children):
    """Sets the children of TREE to CHILDREN, a list of trees."""
    tree[1] = children
```

Will that work? Let's find out...

Remember our current implementation of `tree()`:

```
def tree(label, children=None):
    return (label, list(children or []))
```

We can't mutate elements of tuples, since tuples are immutable.

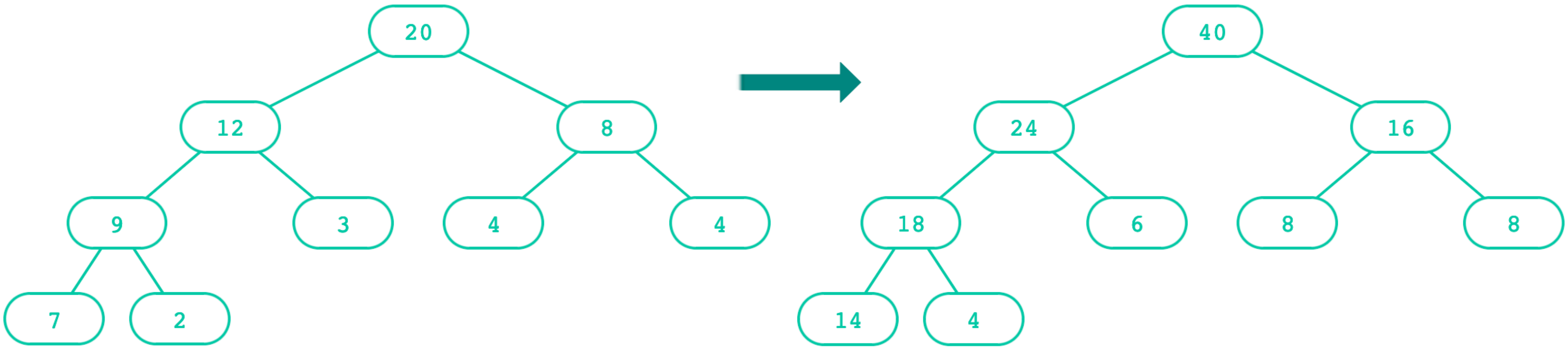
A mutable tree()

A list with label and a list for each child:

```
def tree(label, children=None):
    return [label] + list(children or [])

def label(tree):
    return tree[0]

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

def set_label(tree, label):
    tree[0] = label

def set_children(tree, children):
    tree[1] = children
```

```
t = tree(20, [tree(12,
               [tree(9,
                  [tree(7), tree(2)]),
                tree(3)]),
              tree(8,
                [tree(4), tree(4)])])
set_label(t, 40)
set_children(t, [tree(24)])
```

A destructive tree doubling

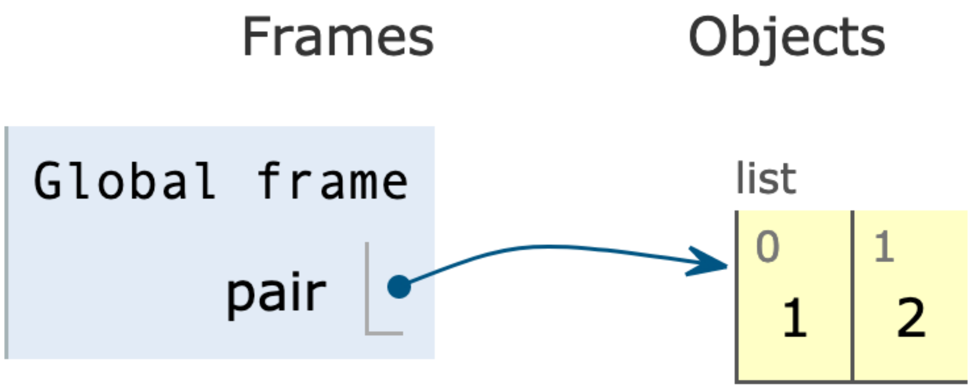



```
def double(t):
    """Doubles every label in T, mutating T."""
    set_label(t, label(t) * 2)
    if not is_leaf(t):
        for c in children(t):
            double(c)
```

Lists

Lists in environment diagrams

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



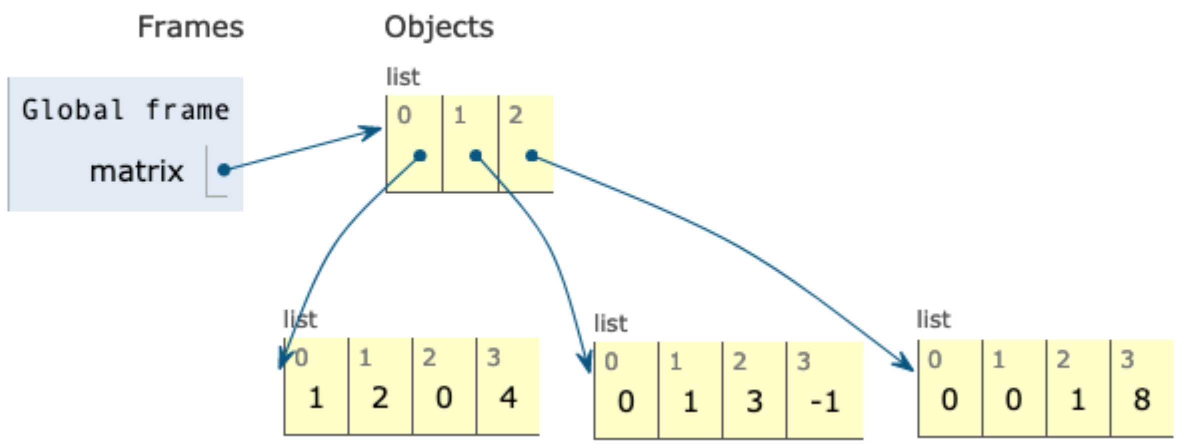
- Lists are represented as a row of index-labeled adjacent boxes, one per element
- Each box either contains a primitive value or points to a compound value

Try in PythonTutor.

Lists in environment diagrams

A nested list:

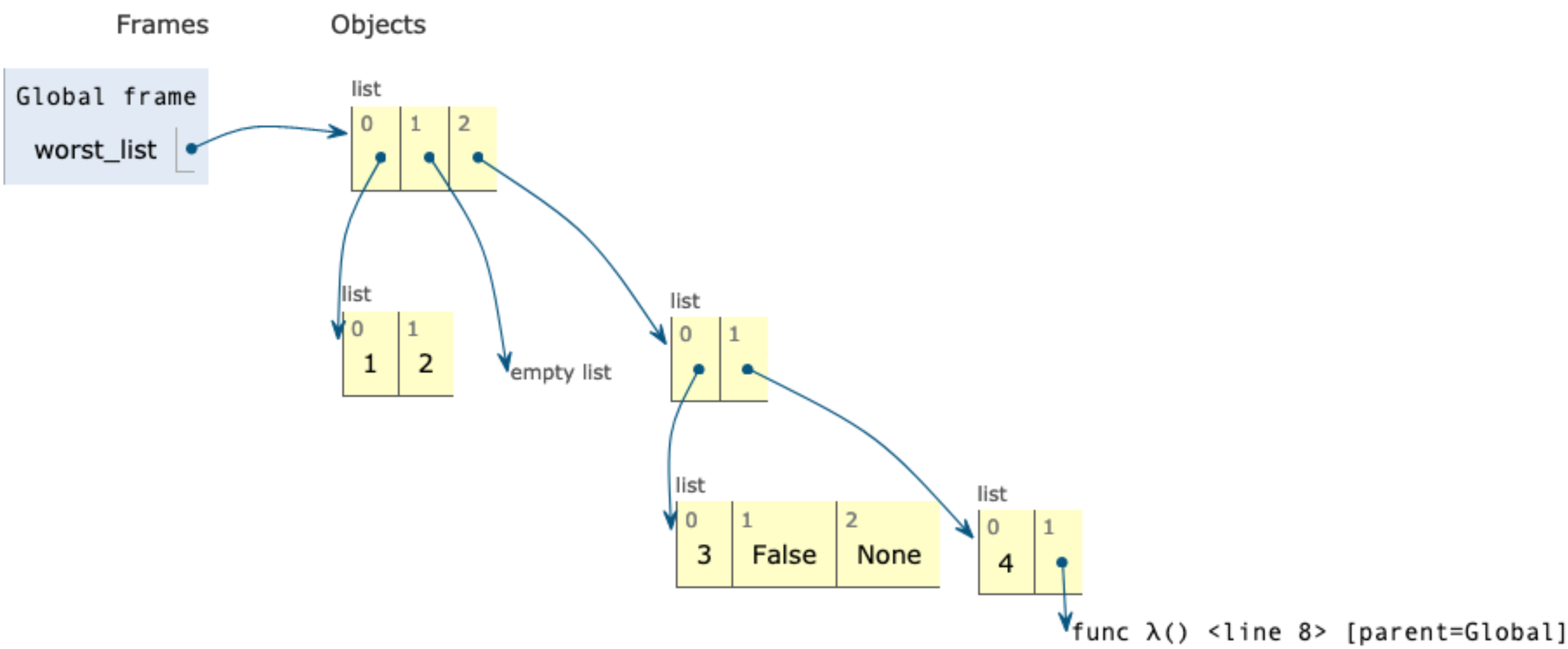
```
matrix = [ [1,2,0,4], [0,1,3,-1], [0,0,1,8] ]
```



Lists in environment diagrams

A *very* nested list:

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



Copying lists

Slicing a whole list copies a list:

```
listA = [2, 3]
listB = listA

listC = listA[:]
listA[0] = 4
listB[1] = 5
```

`list()` creates a new list containing existing elements from any iterable:

```
listA = [2, 3]
listB = listA

listC = list(listA)
listA[0] = 4
listB[1] = 5
```

Try both in PythonTutor.

Python3 provides more ways in the copy module.

Mutability

Is `list(l)`...

- destructive?
- non-destructive?

Are lists...

- mutable?
- immutable?

Mutability

Is `list(l)`...

- destructive?
- non-destructive? 🡸

`list(l)` did not mutate the original iterable, so it is considered a **non-destructive** operation.

Are lists...

- mutable?
- immutable?

Mutability

Is `list(l)`...

- destructive?
- non-destructive? ⬅

`list(l)` did not mutate the original iterable, so it is considered a **non-destructive** operation.

Are lists...

- mutable? ⬅
- immutable?

Python lists *are* mutable. Let's see ways to mutate them!

Mutating lists with slicing

We can do a lot with just brackets/slice notation:

```
L = [1, 2, 3, 4, 5]

L[2] = 6

L[1:3] = [9, 8]

L[2:4] = []             # Deleting elements

L[1:1] = [2, 3, 4, 5]  # Inserting elements

L[len(L):] = [10, 11]  # Appending

L = L + [20, 30]

L[0:0] = range(-3, 0)  # Prependi ng
```

Try in PythonTutor.

Mutating lists with methods

`append()` adds a single element to a list:

```
s = [2, 3]
t = [5, 6]
s.append(4)
s.append(t)
t = 0
```

Try in PythonTutor.

`extend()` adds all the elements in one list to a list:

```
s = [2, 3]
t = [5, 6]
s.extend(4)
s.extend(t)
t = 0
```

Try in PythonTutor.

Mutating lists with methods

`append()` adds a single element to a list:

```
s = [2, 3]
t = [5, 6]
s.append(4)
s.append(t)
t = 0
```

Try in PythonTutor.

`extend()` adds all the elements in one list to a list:

```
s = [2, 3]
t = [5, 6]
s.extend(4) # 🚫 Error: 4 is not an iterable!
s.extend(t)
t = 0
```

Try in PythonTutor. (After deleting the bad line)

Mutating lists with methods

`pop()` removes and returns the last element:

```
s = [2, 3]
t = [5, 6]
t = s.pop()
```

Try in PythonTutor.

`remove()` removes the first element equal to the argument:

```
s = [6, 2, 4, 8, 4]
s.remove(4)
```

Try in PythonTutor.

Identity of objects vs. Equality of contents

Identity: `exp0 is exp1`

evaluates to `True` if both `exp0` and `exp1` evaluate to the same object

Equality: `exp0 == exp1`

evaluates to `True` if both `exp0` and `exp1` evaluate to objects containing equal values

```
list1 = [1,2,3]
list2 = [1,2,3]
identical = list1 is list2
are_equal = list1 == list2
```

Try in PythonTutor.

Identical objects always have equal values.