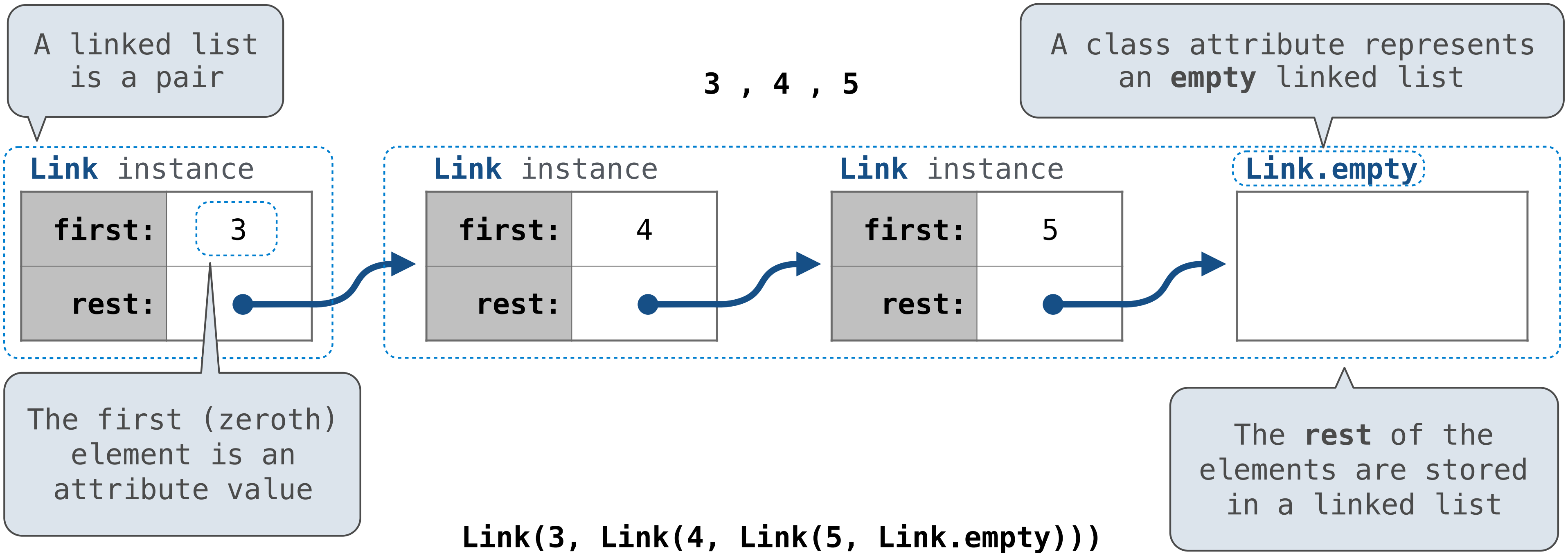
Composition

Announcements

Linked Lists

Linked List Structure

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



A linked list is a pair

3 , 4 , 5

A class attribute represents an **empty** linked list

The first (zeroth) element is an attribute value

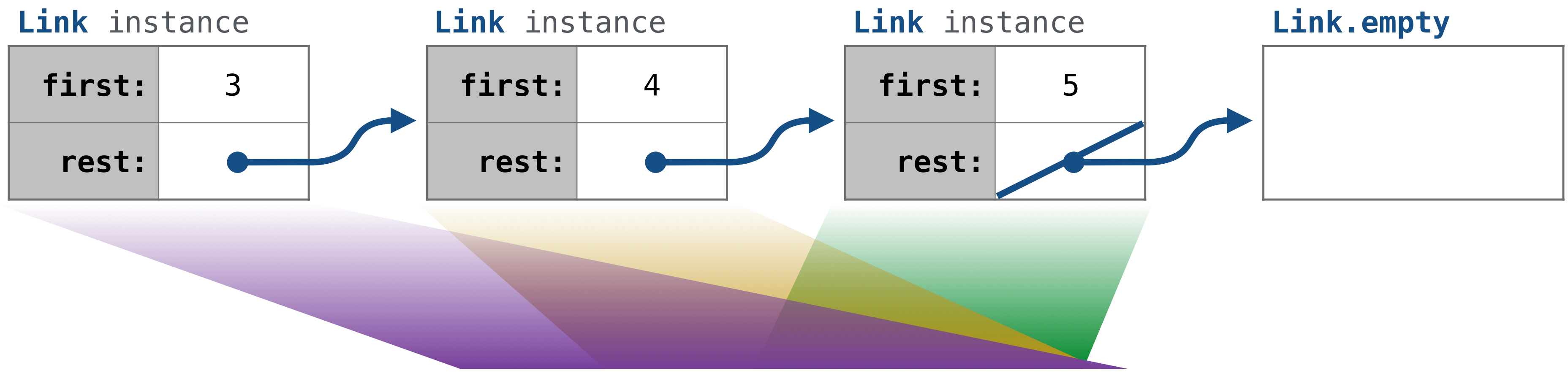
The **rest** of the elements are stored in a linked list

Link(3, Link(4, Link(5, Link.empty)))

Linked List Structure

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

3 , 4 , 5



Link instance

first: 3

rest:

Link instance

first: 4

rest:

Link instance

first: 5

rest:

Link.empty

```
Link(3, Link(4, Link(5, Link.empty)))
```

default = Link.empty

Linked List Class

Linked list class: attributes are passed to `__init__`

```
class Link:

    empty = ()

    def __init__(self, first, rest=empty):
        assert rest is Link.empty or isinstance(rest, Link)
        self.first = first
        self.rest = rest
```

Some zero-length sequence

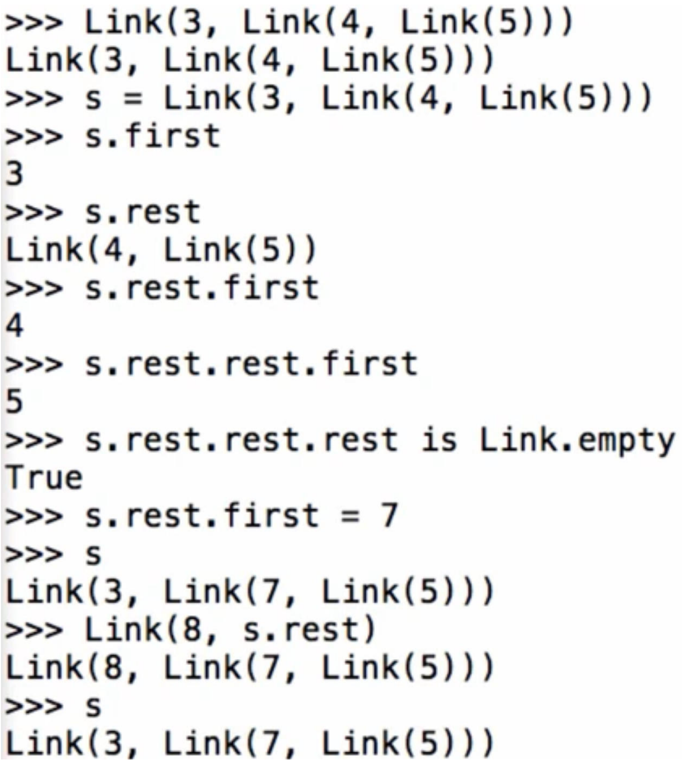
Returns whether rest is a Link

help(isinstance): Return whether an object is an instance of a class or of a subclass thereof.

an instance inheriting from class

Link(3, Link(4, Link(5)))

(Demo)



```
>>> Link(3, Link(4, Link(5)))
Link(3, Link(4, Link(5)))
>>> s = Link(3, Link(4, Link(5)))
>>> s.first
3
>>> s.rest
Link(4, Link(5))
>>> s.rest.first
4
>>> s.rest.rest.first
5
>>> s.rest.rest.rest is Link.empty
True
>>> s.rest.first = 7
>>> s
Link(3, Link(7, Link(5)))
>>> Link(8, s.rest)
Link(8, Link(7, Link(5)))
>>> s
Link(3, Link(7, Link(5)))
```

Property Methods

Property Methods

In some cases, we want the value of instance attributes to be computed on demand

For example, if we want to access the second element of a linked list

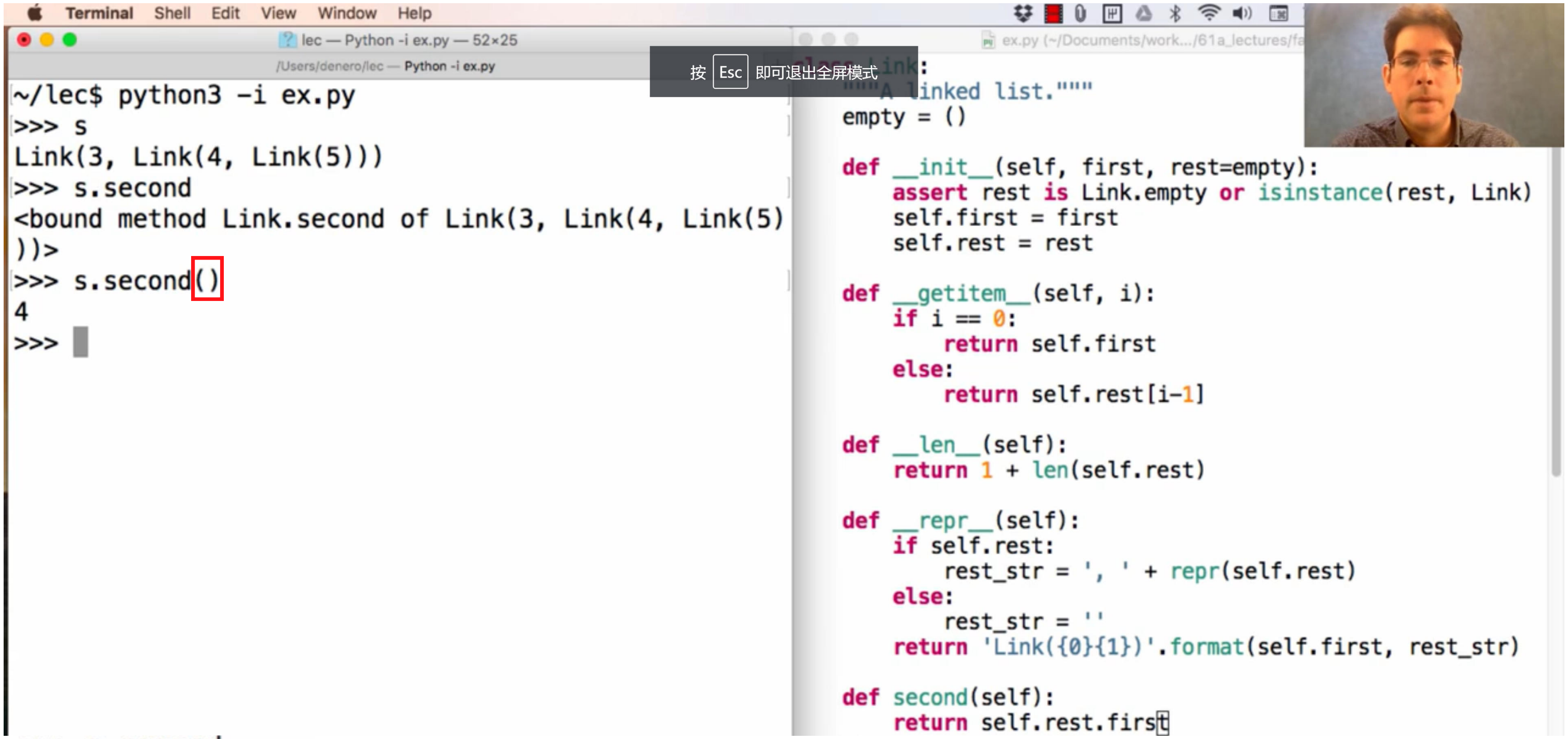
```
>>> s = Link(3, Link(4, Link(5)))
>>> s.second
4
>>> s.second = 6
>>> s.second
6
>>> s
Link(3, Link(6, Link(5)))
```

No method calls!

The @property decorator on a method designates that it will be called whenever it is looked up on an instance

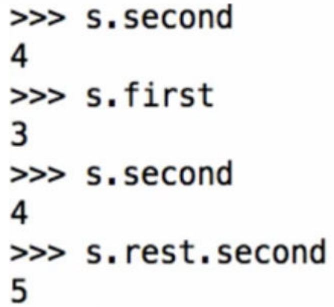
A @<attribute>.setter decorator on a method designates that it will be called whenever that attribute is assigned. <attribute> must be an existing property method.

(Demo)

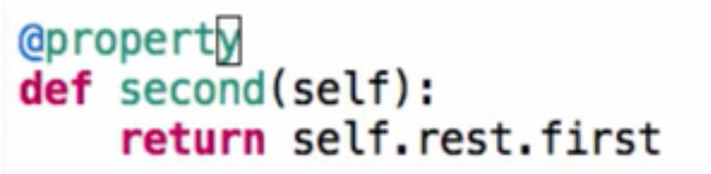

Terminal
Shell
Edit
View
Window
Help
lec — Python -i ex.py — 52×25
/Users/denero/lec — Python -i ex.py
ex.py (~/Documents/work.../61a_lectures/fa
按 Esc 即可退出全屏模式

```
~/lec$ python3 -i ex.py
>>> s
Link(3, Link(4, Link(5)))
>>> s.second
<bound method Link.second of Link(3, Link(4, Link(5)
))>
>>> s.second()
4
>>> 
```

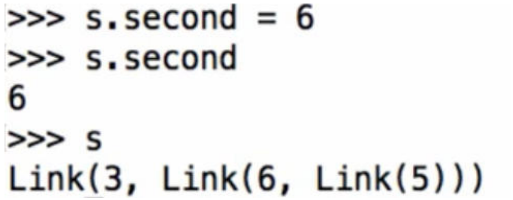
```
class Link:
    """A linked list."""
    empty = ()

    def __init__(self, first, rest=empty):
        assert rest is Link.empty or isinstance(rest, Link)
        self.first = first
        self.rest = rest

    def __getitem__(self, i):
        if i == 0:
            return self.first
        else:
            return self.rest[i-1]

    def __len__(self):
        return 1 + len(self.rest)

    def __repr__(self):
        if self.rest:
            rest_str = ', ' + repr(self.rest)
        else:
            rest_str = ''
        return 'Link({0}{1})'.format(self.first, rest_str)

    def second(self):
        return self.rest.first
```



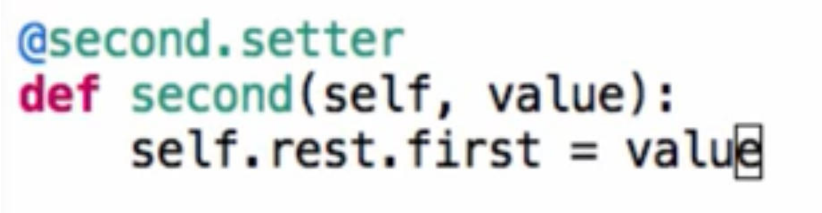
```
>>> s.second
4
>>> s.first
3
>>> s.second
4
>>> s.rest.second
5
```



```
@property
def second(self):
    return self.rest.first
```



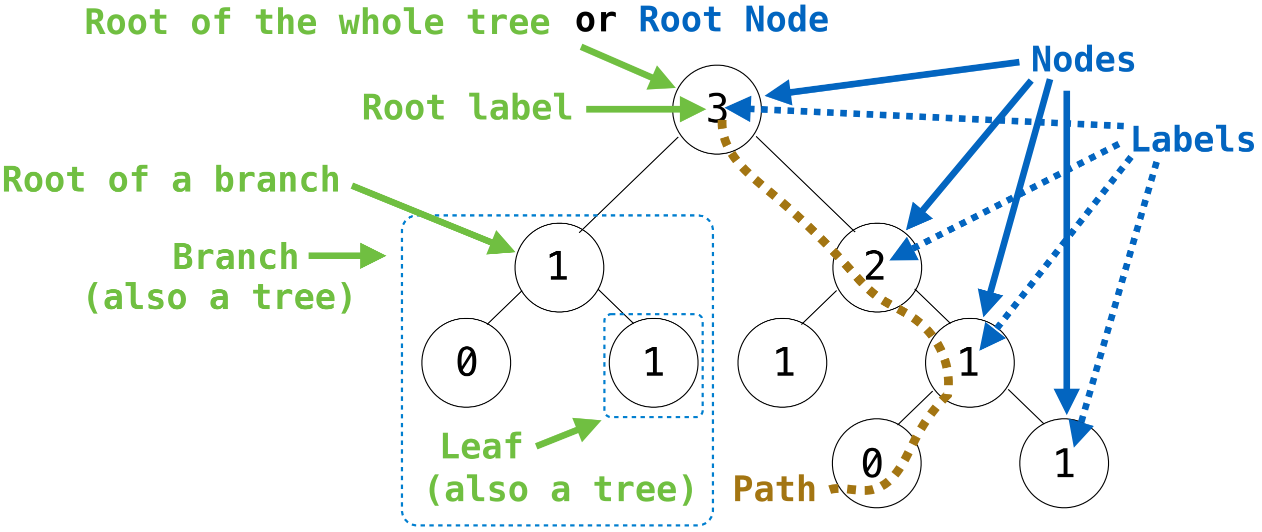
```
>>> s.second = 6
>>> s.second
6
>>> s
Link(3, Link(6, Link(5)))
```



```
@second.setter
def second(self, value):
    self.rest.first = value
```

Tree Class

Tree Abstraction (Review)



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"

Tree Class

A Tree has a label and a list of branches; each branch is a Tree

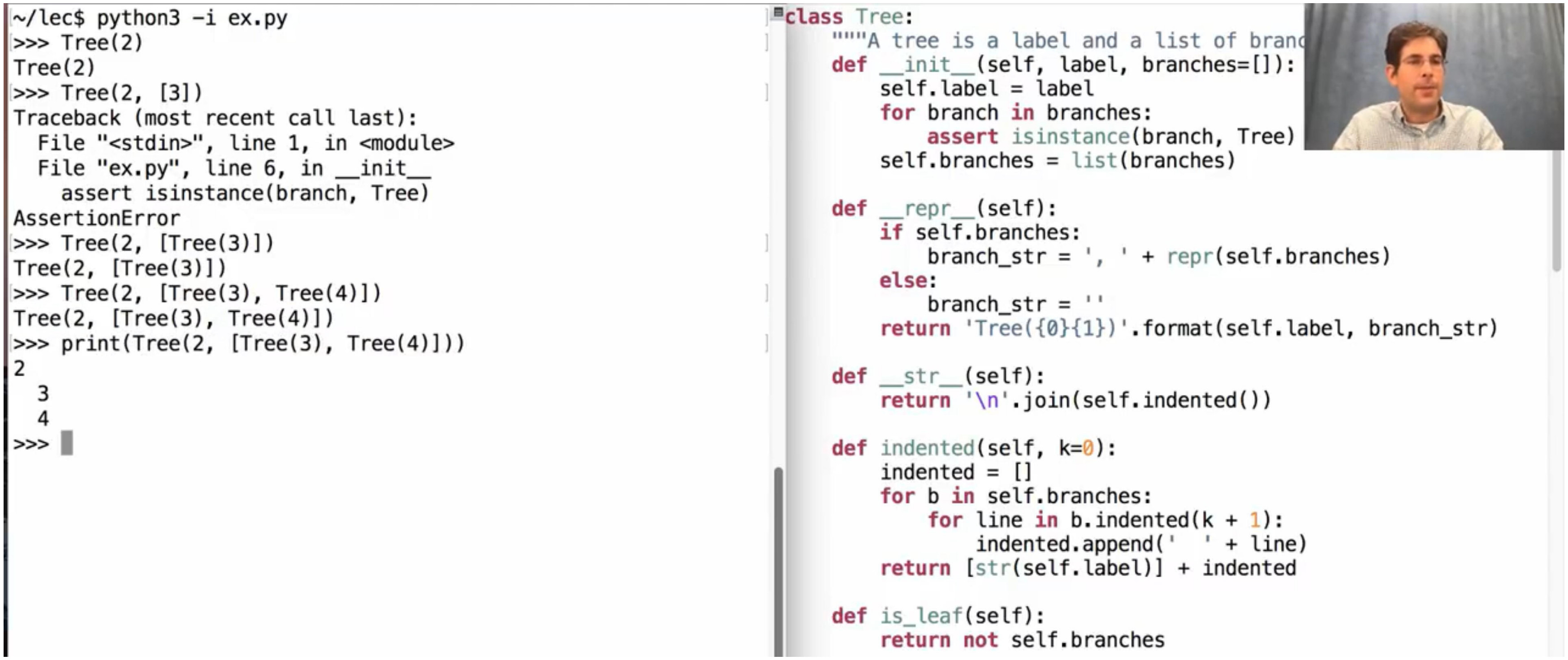
```
class Tree:
    def __init__(self, label, branches=[]):
        self.label = label
        for branch in branches:
            assert isinstance(branch, Tree)
        self.branches = list(branches)
```

```
def fib_tree(n):
    if n == 0 or n == 1:
        return Tree(n)
    else:
        left = fib_tree(n-2)
        right = fib_tree(n-1)
        fib_n = left.label + right.label
        return Tree(fib_n, [left, right])
```

```
def tree(label, branches=[]):
    for branch in branches:
        assert is_tree(branch)
    return [label] + list(branches)
def label(tree):
    return tree[0]
def branches(tree):
    return tree[1:]
```

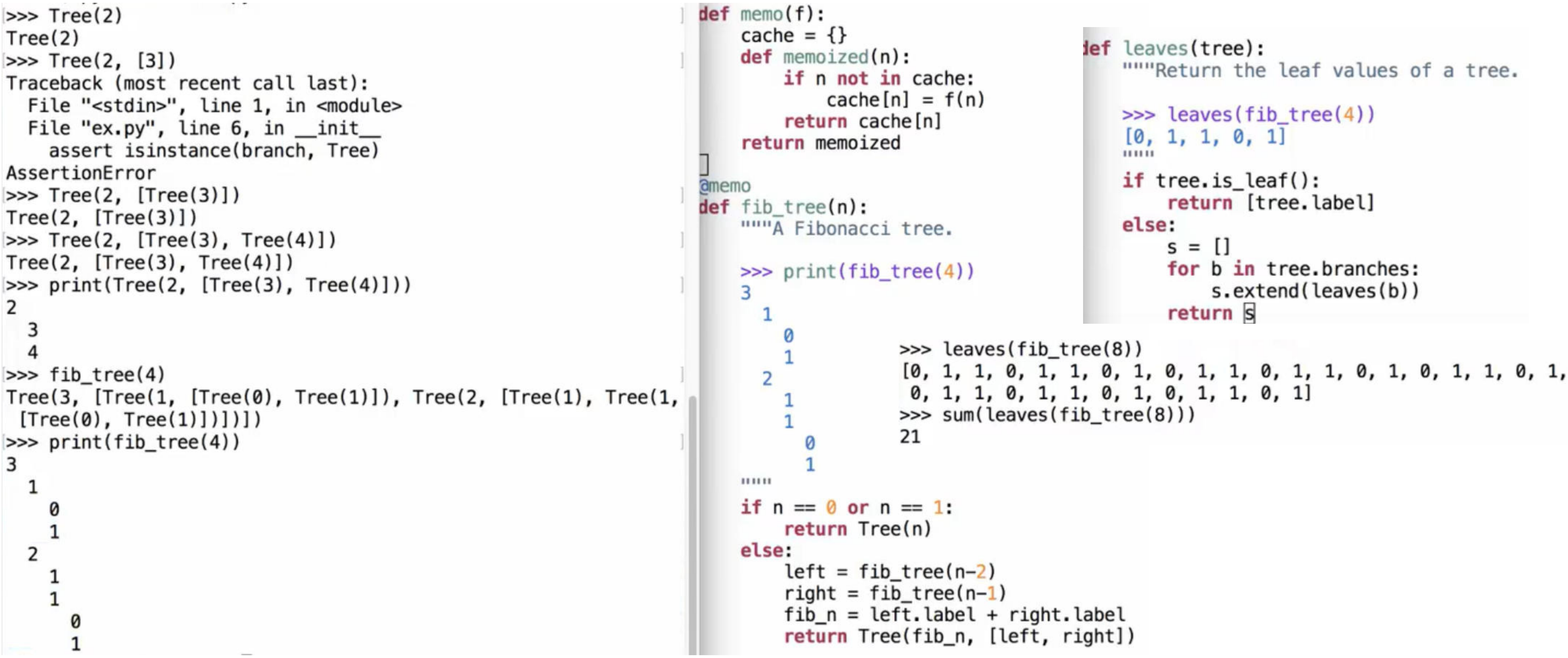
```
def fib_tree(n):
    if n == 0 or n == 1:
        return tree(n)
    else:
        left = fib_tree(n-2)
        right = fib_tree(n-1)
        fib_n = label(left) + label(right)
        return tree(fib_n, [left, right])
```

(Demo)



```
~/lec$ python3 -i ex.py
>>> Tree(2)
Tree(2)
>>> Tree(2, [3])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "ex.py", line 6, in __init__
    assert isinstance(branch, Tree)
AssertionError
>>> Tree(2, [Tree(3)])
Tree(2, [Tree(3)])
>>> Tree(2, [Tree(3), Tree(4)])
Tree(2, [Tree(3), Tree(4)])
>>> print(Tree(2, [Tree(3), Tree(4)]))
2
  3
  4
>>> 
```

```
class Tree:
    """A tree is a label and a list of branc
    def __init__(self, label, branches=[]):
        self.label = label
        for branch in branches:
            assert isinstance(branch, Tree)
        self.branches = list(branches)

    def __repr__(self):
        if self.branches:
            branch_str = ', ' + repr(self.branches)
        else:
            branch_str = ''
        return 'Tree({0}{1})'.format(self.label, branch_str)

    def __str__(self):
        return '\n'.join(self.indented())

    def indented(self, k=0):
        indented = []
        for b in self.branches:
            for line in b.indented(k + 1):
                indented.append('  ' + line)
        return [str(self.label)] + indented

    def is_leaf(self):
        return not self.branches
```



```
>>> Tree(2)
Tree(2)
>>> Tree(2, [3])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "ex.py", line 6, in __init__
    assert isinstance(branch, Tree)
AssertionError
>>> Tree(2, [Tree(3)])
Tree(2, [Tree(3)])
>>> Tree(2, [Tree(3), Tree(4)])
Tree(2, [Tree(3), Tree(4)])
>>> print(Tree(2, [Tree(3), Tree(4)]))
2
  3
  4
>>> fib_tree(4)
Tree(3, [Tree(1, [Tree(0), Tree(1)]), Tree(2, [Tree(1), Tree(1,
 [Tree(0), Tree(1)])])])
>>> print(fib_tree(4))
3
  1
    0
    1
  2
    1
    1
      0
      1
```

```
def memo(f):
    cache = {}
    def memoized(n):
        if n not in cache:
            cache[n] = f(n)
        return cache[n]
    return memoized

@memo
def fib_tree(n):
    """A Fibonacci tree.

    >>> print(fib_tree(4))
    3
      1
        0
        1
      2
        1
        1
          0
          1
    """
    if n == 0 or n == 1:
        return Tree(n)
    else:
        left = fib_tree(n-2)
        right = fib_tree(n-1)
        fib_n = left.label + right.label
        return Tree(fib_n, [left, right])
```

```
def leaves(tree):
    """Return the leaf values of a tree.

    >>> leaves(fib_tree(4))
    [0, 1, 1, 0, 1]
    """
    if tree.is_leaf():
        return [tree.label]
    else:
        s = []
        for b in tree.branches:
            s.extend(leaves(b))
        return s
```

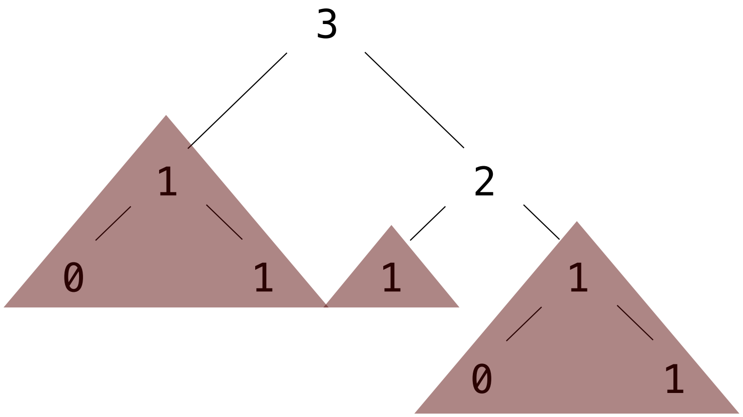
```
>>> leaves(fib_tree(8))
[0, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1,
 0, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1]
>>> sum(leaves(fib_tree(8)))
21
```

Tree Mutation

Example: Pruning Trees

Removing subtrees from a tree is called *pruning*

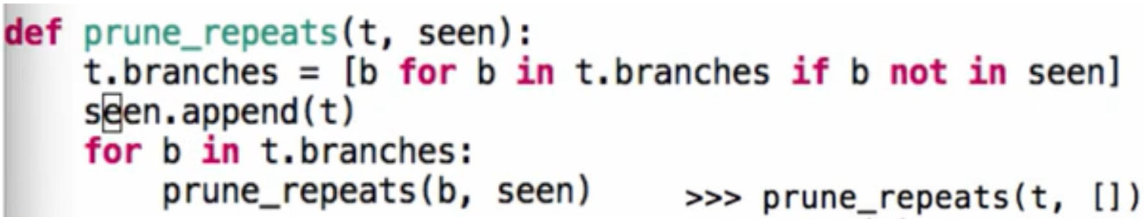
Prune branches before recursive processing



```
def prune(t, n):
    """Prune sub-trees whose label value is n."""
    t.branches = [_______b_______ for b in t.branches if _____b.label != n_____]
    for b in t.branches:
        prune(_______________b_______________, _______________n_______________)
```

(Demo)

1. Pruning the major branch
2. Update the seen list, preparing for the pruning of the minor branches, e.g. the fib(3) under fib(4)
3. Call recursively



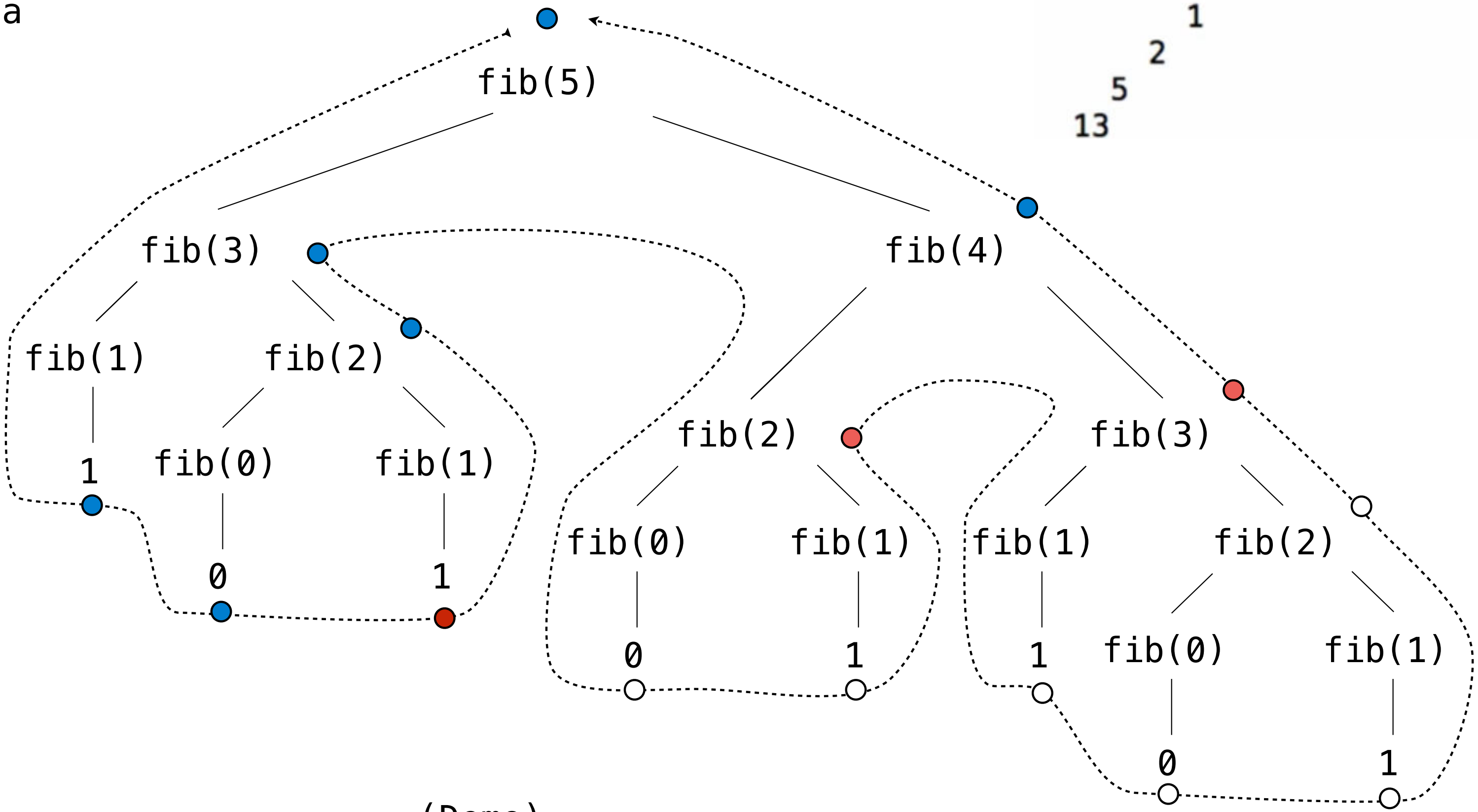
```
def prune_repeats(t, seen):
    t.branches = [b for b in t.branches if b not in seen]
    seen.append(t)
    for b in t.branches:
        prune_repeats(b, seen)
```

```
>>> prune_repeats(t, [])
>>> print(t)
21
  8
    3
      1
        0
        1
      2
    5
  13
```

Example: Pruning Trees

Removing subtrees from a tree is called *pruning*

Prune branches before recursive processing



Memoization:

● Returned by fib

● Found in cache

○ Skipped

(Demo)