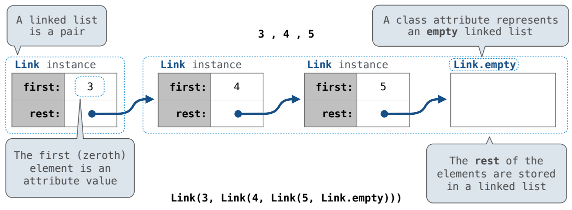
## Linked Lists

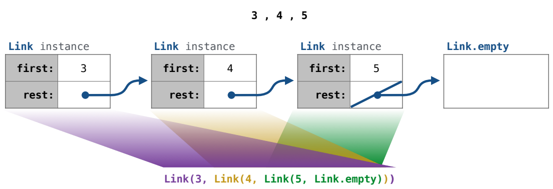
## Linked List Structure

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




## Linked List Structure

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




## 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.

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

(Demo)

## 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

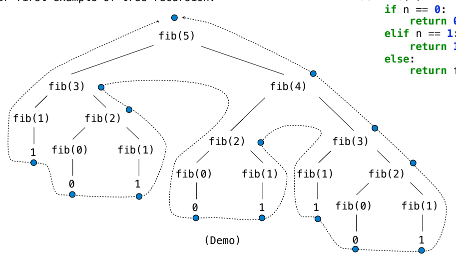
A @<attribute>.setter decorator on a method designates that it will be called whenever that attribute is assigned. <attribute> must be an existing property method.

(Demo)

## Tree Recursion Efficiency

## Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:




```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```

http://en.wikipedia.org/wiki/File:Fibonacci.jpg

## Memoization

## Memoization

**Idea:** Remember the results that have been computed before

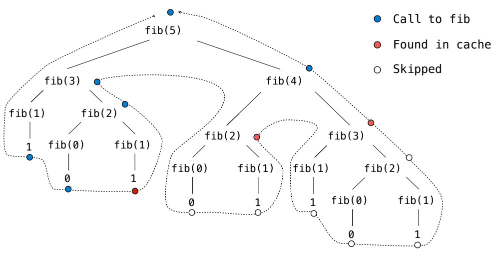
```
def memo(f):
    cache = {}
    def memoized(n):
        if n not in cache:
            cache[n] = f(n)
        return cache[n]
    return memoized
```

Keys are arguments that map to return values

Same behavior as f, if f is a pure function

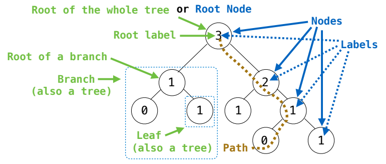
(Demo)

## Memoized Tree Recursion




## 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 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 = left.label + right.label
        return Tree(fib_n, [left, right])
```

```
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])
```

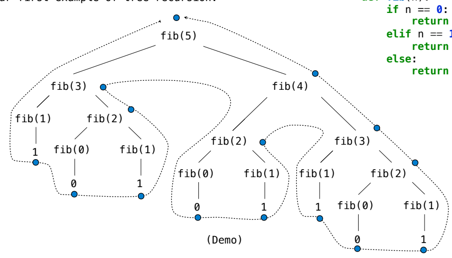
(Demo)

## Measuring Efficiency

## Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```




(Demo)

http://en.wikipedia.org/wiki/File:Fibonacci.jpg

## Memoization

## Memoization

**Idea:** Remember the results that have been computed before

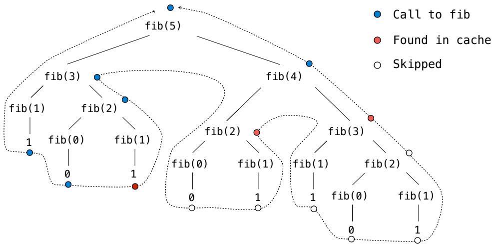
```
def memo(f):
    cache = {}
    def memoized(n):
        if n not in cache:
            cache[n] = f(n)
        return cache[n]
    return memoized
```

Keys are arguments that map to return values

Same behavior as f, if f is a pure function

(Demo)

## Memoized Tree Recursion




## Exponentiation

## Exponentiation

**Goal:** one more multiplication lets us double the problem size

```
def exp(b, n):
    if n == 0:
        return 1
    else:
        return b * exp(b, n-1)
```

$$b^n = \begin{cases} 1 & \text{if } n = 0 \\ b \cdot b^{n-1} & \text{otherwise} \end{cases}$$

```
def exp_fast(b, n):
    if n == 0:
        return 1
    elif n % 2 == 0:
        return square(exp_fast(b, n//2))
    else:
        return b * exp_fast(b, n-1)

def square(x):
    return x * x
```

$$b^n = \begin{cases} 1 & \text{if } n = 0 \\ (b^{\frac{1}{2}n})^2 & \text{if } n \text{ is even} \\ b \cdot b^{n-1} & \text{if } n \text{ is odd} \end{cases}$$

(Demo)

## Exponentiation

**Goal:** one more multiplication lets us double the problem size

```
def exp(b, n):
    if n == 0:
        return 1
    else:
        return b * exp(b, n-1)
```

Linear time:

- Doubling the input **doubles** the time
- 1024x the input takes 1024x as much time

```
def exp_fast(b, n):
    if n == 0:
        return 1
    elif n % 2 == 0:
        return square(exp_fast(b, n//2))
    else:
        return b * exp_fast(b, n-1)

def square(x):
    return x * x
```

Logarithmic time:

- Doubling the input **increases** the time by a constant C
- 1024x the input increases the time by only 10 times C

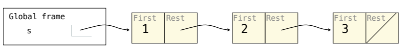
## Mutable Linked Lists

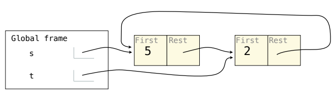
## Recursive Lists Can Change

Attribute assignment statements can change first and rest attributes of a Link

The rest of a linked list can contain the linked list as a sub-list

```
>>> s = Link(1, Link(2, Link(3)))
>>> s.first = 5
>>> t = s.rest
>>> t.rest = s
>>> s.first
5
>>> s.rest.rest.rest.rest.rest.first
2
```

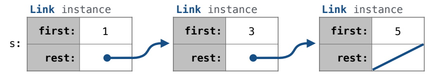



Note: The actual environment diagram is much more complicated.

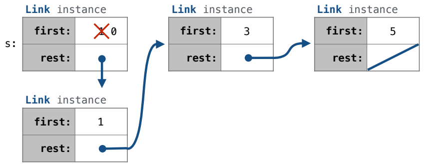
## Linked List Mutation Example

## Adding to an Ordered List



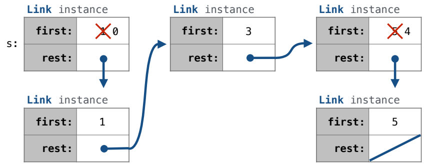

```
def add(s, v):
    """Add v to an ordered list s with no repeats, returning modified s."""
    (Note: If v is already in s, then don't modify s, but still return it.)

        add(s, 0)
```

## Adding to an Ordered List



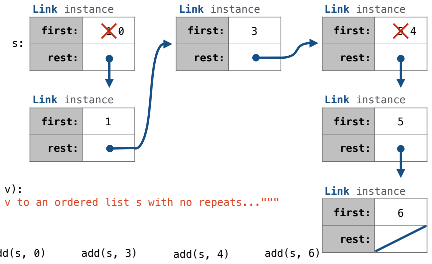

```
def add(s, v):
    """Add v to an ordered list s with no repeats, returning modified s."""
    (Note: If v is already in s, then don't modify s, but still return it.)

        add(s, 0)     add(s, 3)     add(s, 4)
```

## Adding to an Ordered List



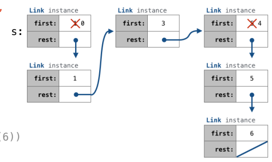

```
def add(s, v):
    """Add v to an ordered list s with no repeats..."""

        add(s, 0)     add(s, 3)     add(s, 4)     add(s, 6)
```

## Adding to an Ordered List




```
def add(s, v):
    """Add v to an ordered list s with no repeats..."""

        add(s, 0)     add(s, 3)     add(s, 4)     add(s, 6)
```

## Adding to a Set Represented as an Ordered List



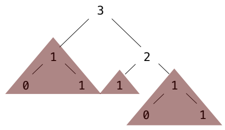

```
def add(s, v):
    """Add v to a set s, returning modified s."""

    >>> s = Link(1, Link(3, Link(5)))
    >>> add(s, 0)
    Link(0, Link(1, Link(3, Link(5))))
    >>> add(s, 3)
    Link(0, Link(1, Link(3, Link(5))))
    >>> add(s, 4)
    Link(0, Link(1, Link(3, Link(4, Link(5)))))
    >>> add(s, 6)
    Link(0, Link(1, Link(3, Link(4, Link(5, Link(6))
    """
    assert s is not Link.empty
    if s.first > v:
                                  v                        Link(s.first, s.rest)
        s.first, s.rest = ____________________ , ______________________
    elif s.first < v and empty(s.rest):
                                        Link(v)
        s.rest = ______________________________________________
    elif s.first < v:
                                         add(s.rest, v)
        ___________________________________________________
    return s
```

## Tree Mutation

## Example: Pruning Trees

Removing subtrees from a tree is called *pruning*

Prune branches before recursive processing




```
def prune(t, n):
    """Prune all sub-trees whose label is n."""
    t.branches = [___b___ for b in t.branches if ___b.label != n___]
    for b in t.branches:
        prune(___b___, ___n___)
```