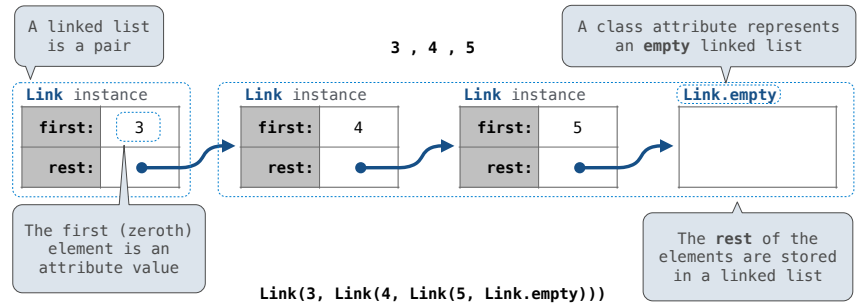# Composition

# Announcements

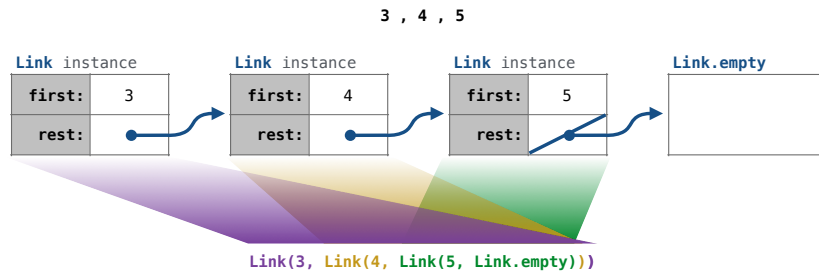# Linked Lists

## 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

| **Link** instance | | | **Link** instance | | **Link** instance | | **Link.empty** |
|---|---|---|---|---|---|---|---|
| **first:** | 3 | | **first:** | 4 | **first:** | 5 | |
| **rest:** | | | **rest:** | | **rest:** | | |

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    **Link** instance    **Link** instance    **Link.empty**

| first: | 3 |
| rest: | ● |

| first: | 4 |
| rest: | ● |

| first: | 5 |
| rest: | ● |

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

---

## Linked List Class

Linked list class: attributes are passed to __init__

```
class Link:
    empty = ()          Some zero-length sequence

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

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)

---

## Linked List Processing

---

## Example: Range, Map, and Filter for Linked Lists

```
square, odd = lambda x: x * x, lambda x: x % 2 == 1
list(map(square, filter(odd, range(1, 6))))          # [1, 9, 25]
map_link(square, filter_link(odd, range_link(1, 6)))  # Link(1, Link(9, Link(25)))

def range_link(start, end):
    """Return a Link containing consecutive integers from start to end.

    >>> range_link(3, 6)
    Link(3, Link(4, Link(5)))
    """

def map_link(f, s):
    """Return a Link that contains f(x) for each x in Link s.

    >>> map_link(square, range_link(3, 6))
    Link(9, Link(16, Link(25)))
    """

def filter_link(f, s):
    """Return a Link that contains only the elements x of Link s for which f(x)
    is a true value.

    >>> filter_link(odd, range_link(3, 6))
    Link(3, Link(5))
    """
```
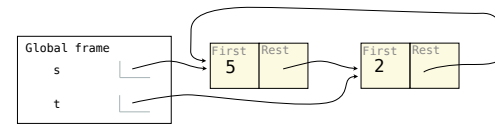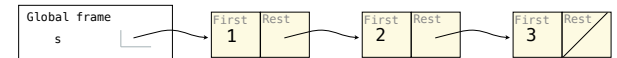
# Linked Lists Mutation

## Linked 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
```
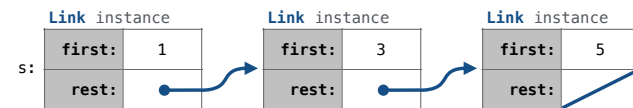


Note: The actual environment diagram is much more complicated.

---

# 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)
```

**Link** instance — first: ❌0, rest: →
**Link** instance — first: 3, rest: →
**Link** instance — first: 5, rest: (empty)
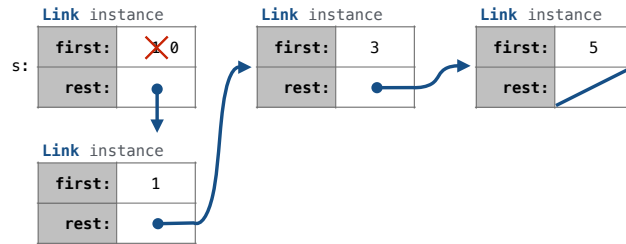**Link** instance — first: 1, rest: →
s:

```
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)
```
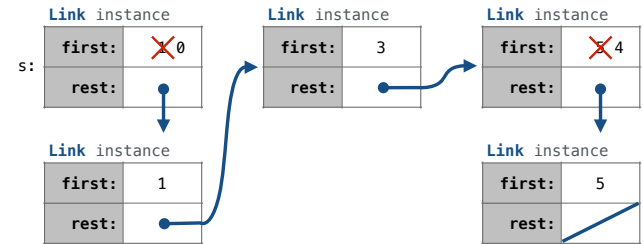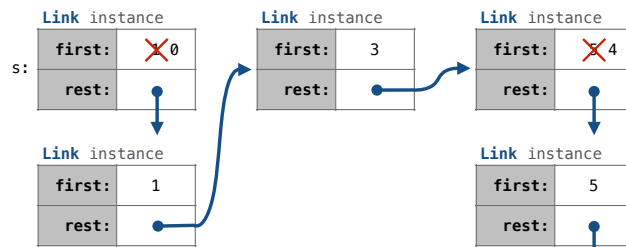
---

**Link** instance — first: ❌0, rest: →
**Link** instance — first: 3, rest: →
**Link** instance — first: ❌4, rest: →
**Link** instance — first: 1, rest: →
**Link** instance — first: 5, rest: (empty)
s:

```
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)
```

---

**Link** instance — first: ❌0, rest: →
**Link** instance — first: 3, rest: →
**Link** instance — first: ❌4, rest: →
**Link** instance — first: 1, rest: →
**Link** instance — first: 5, rest: →
**Link** instance — first: 6, rest: (empty)
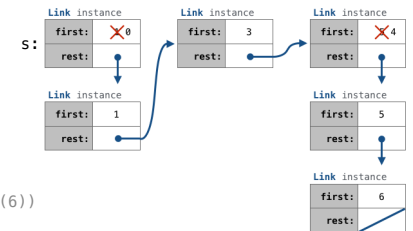s:

```
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 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 List.empty
    if s.first > v:
        s.first, s.rest = _____ v _____ , _____ Link(s.first, s.rest) _____
    elif s.first < v and empty(s.rest):
        s.rest = _____ Link(v) _____
    elif s.first < v:
        _____ add(s.rest, v) _____
    return s
```
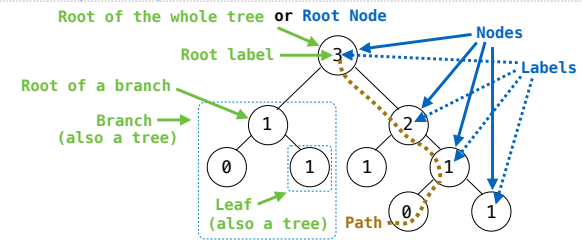
**Link** instance — first: ❌0, rest: →
**Link** instance — first: 3, rest: →
**Link** instance — first: ❌4, rest: →
**Link** instance — first: 1, rest: →
**Link** instance — first: 5, rest: →
**Link** instance — first: 6, rest: (empty)
s:

## Tree Class

---

## Tree Abstraction (Review)



Root of the whole tree **or** Root Node

Root label

Root of a branch

Branch → (also a tree)

Leaf (also a tree)

Nodes

Labels

Path

**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)

---

## Tree Mutation

Removing subtrees from a
tree is called *pruning*

Prune branches before
recursive processing
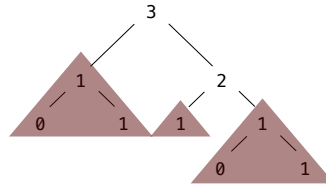


```python
def prune(t, n):

    """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_____)
```

21