Sets

One more built-in Python container type
Set literals are enclosed in braces
Duplicate elements are removed on construction
Sets have arbitrary order, just like dictionary entries

```python
>>> s = {3, 2, 1, 4, 4}
>>> s
{1, 2, 3, 4}
>>> 3 in s
True
>>> len(s)
4
>>> s.
union({1, 5})
{1, 2, 3, 4, 5}
>>> s.intersection({6, 5, 4, 3})
{3, 4}
>>> s
demonstration set
```

Implementing Sets

What we should be able to do with a set:
- Membership testing: Is a value an element of a set?
- Union: Return a set with all elements in set1 or set2
- Intersection: Return a set with any elements in set1 and set2
- Adjoin: Return a set with all elements in s and a value v

**Sets as Unordered Sequences**

Proposal 1: A set is represented by a linked list that contains no duplicate items.

```
def empty(s):
    return s is Link.empty

def contains(s, v):
    """Return whether set s contains value v.
    >>> s = Link(1, Link(3, Link(2)))
    >>> contains(s, 3)
    True
    >>> contains(s, 5)
    False"
    return s is null or contains(s.rest, v)

def adjoin(s, v):
    if contains(s, v):
        return s
    else:
        return Link(v, s)

def intersect(set1, set2):
    in_set2 = lambda v: contains(set2, v)
    return filter_link(in_set2, set1)

def union(set1, set2):
    not_in_set2 = lambda v: not contains(set2, v)
    set1_not_set2 = filter_link(not_in_set2, set1)
    return extend_link(set1_not_set2, set2)

def filter_link(in_set, link):
    return extend_link(filter(in_set, link), Link.empty)
```

Time order of worst-case growth
- empty(s): \(\Theta(1)\)
- contains(s, v): \(\Theta(n)\) where \(n\) is the size of the set
- adjoin(s, v): \(\Theta(n)\) in the worst case; \(\Theta(1)\) in the average case
- intersect(set1, set2): \(\Theta(n^2)\)
- union(set1, set2): \(\Theta(n^2)\)

Sets as Linked Lists

```
def intersection(set1, set2):
    in_set2 = lambda v: contains(set2, v)
    return filter_link(in_set2, set1)

def union(set1, set2):
    not_in_set2 = lambda v: not contains(set2, v)
    set1_not_set2 = filter_link(not_in_set2, set1)
    return extend_link(set1_not_set2, set2)
```

Time order of worst-case growth
- empty(s): \(\Theta(1)\)
- contains(s, v): \(\Theta(1)\)
- adjoin(s, v): \(\Theta(n)\)
- intersection(set1, set2): \(\Theta(n^2)\)
- union(set1, set2): \(\Theta(n^2)\)
Sets as Ordered Linked Lists

### Searching an Ordered List

```python
>>> s = Link(1, Link(3, Link(5)))
>>> contains(s, 1)
True
>>> contains(s, 2)
False
>>> t = adjoint(s, 2)
```

### Intersecting Ordered Linked Lists

```python
def intersect(set1, set2):
    if empty(set1) or empty(set2):
        return Link.empty
    else:
        e1, e2 = set1.first, set2.first
        if e1 == e2:
            return Link(e1, intersect(set1.rest, set2.rest))
        elif e1 < e2:
            return intersect(set1.rest, set2)
        else:
            return intersect(set1, set2.rest)
```

### Adding to an Ordered List

```python
add(s, 0)
```

`add(s, 0)` Try to return the same object as input.

---

**Sets as Ordered Sequences**

**Proposal 2:** A set is represented by a linked list with unique elements that is ordered from least to greatest

<table>
<thead>
<tr>
<th>Parts of the program that...</th>
<th>Assume that sets are...</th>
<th>Using...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use sets to contain values</td>
<td>unordered collections</td>
<td>empty, contains, adjoint, intersect, union</td>
</tr>
</tbody>
</table>

Different parts of a program may make different assumptions about data.
Adding to an Ordered List

```python
add(s, 5)
```

Adding to a Set Represented as an Ordered List

```python
def add(s, v):
    """Add v to a set s, returning modified s."""
    if empty(s):
        return Link(v)
    elif s.first < v:
        s.first, s.rest = add(s.rest, v)
        return s
    elif s.first > v and empty(s.rest):
        s.first, s.rest = add(s.first, v)
        return s
    else:
        add(s.first, v)
```

```python
>>> s = Link(2, Link(3, Link(5)))
>>> add(s, 0)
Link(0, Link(2, Link(3, Link(5))))
>>> add(s, 3)
Link(0, Link(2, Link(3, Link(5))))
>>> add(s, 4)
Link(0, Link(2, Link(3, Link(4, Link(5)))))
```

```python
>>> add(s, 6)
Link(0, Link(2, Link(3, Link(4, Link(5, Link(6)))))
```