Set All Versions

Examples

Operations on sets

with other sets

```
# Intersection
{1, 2, 3, 4, 5}.intersection({3, 4, 5, 6})  # {3, 4, 5}
{1, 2, 3, 4, 5} & {3, 4, 5, 6}              # {3, 4, 5}

# Union
{1, 2, 3, 4, 5}.union({3, 4, 5, 6})  # {1, 2, 3, 4, 5, 6}
{1, 2, 3, 4, 5} | {3, 4, 5, 6}       # {1, 2, 3, 4, 5, 6}

# Difference
{1, 2, 3, 4}.difference({2, 3, 5})  # {1, 4}
{1, 2, 3, 4} - {2, 3, 5}            # {1, 4}

# Symmetric difference with
{1, 2, 3, 4}.symmetric_difference({2, 3, 5})  # {1, 4, 5}
{1, 2, 3, 4} ^ {2, 3, 5}                      # {1, 4, 5}

# Superset check
{1, 2}.issuperset({1, 2, 3})  # False
{1, 2} >= {1, 2, 3}           # False

# Subset check
{1, 2}.issubset({1, 2, 3})  # True
{1, 2} <= {1, 2, 3}         # True

# Disjoint check
{1, 2}.isdisjoint({3, 4})  # True
{1, 2}.isdisjoint({1, 4})  # False
```

with single elements

```
# Existence check
2 in {1,2,3}      # True
4 in {1,2,3}      # False
4 not in {1,2,3}  # True

# Add and Remove
s = {1,2,3}
s.add(4)        # s == {1,2,3,4}

s.discard(3)    # s == {1,2,4}
s.discard(5)    # s == {1,2,4}

s.remove(2)     # s == {1,4}
s.remove(2)     # KeyError!
```

Set operations return new sets, but have the corresponding in-place versions:

method	in-place operation	in-place method
union	s |= t	update
intersection	s &= t	intersection_update
difference	s -= t	difference_update
symmetric_difference	s ^= t	symmetric_difference_update

For example:

```
s = {1, 2}
s.update({3, 4})   # s == {1, 2, 3, 4}
```

Get the unique elements of a list

Let's say you've got a list of restaurants -- maybe you read it from a file. You care about the *unique* restaurants in the list. The best way to get the unique elements from a list is to turn it into a set:

```
restaurants = ["McDonald's", "Burger King", "McDonald's", "Chicken Chicken"]
```

```
unique_restaurants = set(restaurants)
print(unique_restaurants)
# prints {'Chicken Chicken', "McDonald's", 'Burger King'}
```

Note that the set is not in the same order as the original list; that is because sets are *unordered* , just like `dict` s.

This can easily be transformed back into a `List` with Python's built in `list` function, giving another list that is the same list as the original but without duplicates:

```
list(unique_restaurants)
# ['Chicken Chicken', "McDonald's", 'Burger King']
```

It's also common to see this as one line:

```
# Removes all duplicates and returns another list
list(set(restaurants))
```

Now any operations that could be performed on the original list can be done again.

Set of Sets

```
{{1,2}, {3,4}}
```

leads to:

```
TypeError: unhashable type: 'set'
```

Instead, use `frozenset` :

```
{frozenset({1, 2}), frozenset({3, 4})}
```

Set Operations using Methods and Builtins

We define two sets `a` and `b`

```
>>> a = {1, 2, 2, 3, 4}
>>> b = {3, 3, 4, 4, 5}
```

> NOTE: `{1}` creates a set of one element, but `{}` creates an empty `dict` . The correct way to create an empty set is `set()` .

Intersection

`a.intersection(b)` returns a new set with elements present in both `a` and `b`

```
>>> a.intersection(b)
{3, 4}
```

Union

`a.union(b)` returns a new set with elements present in either `a` and `b`

```
>>> a.union(b)
{1, 2, 3, 4, 5}
```

Difference

`a.difference(b)` returns a new set with elements present in `a` but not in `b`

```
>>> a.difference(b)
{1, 2}
>>> b.difference(a)
{5}
```

Symmetric Difference

`a.symmetric_difference(b)` returns a new set with elements present in either `a` or `b` but not in both

```
>>> a.symmetric_difference(b)
{1, 2, 5}
```

```
>>> b.symmetric_difference(a)
{1, 2, 5}
```

NOTE : a.symmetric_difference(b) == b.symmetric_difference(a)

Subset and superset

c.issubset(a) tests whether each element of c is in a .

a.issuperset(c) tests whether each element of c is in a .

```
>>> c = {1, 2}
>>> c.issubset(a)
True
>>> a.issuperset(c)
True
```

Disjoint sets

Sets a and d are disjoint if no element in a is also in d and vice versa.

```
>>> d = {5, 6}
>>> a.isdisjoint(b) # {2, 3, 4} are in both sets
False
>>> a.isdisjoint(d)
True

# This is an equivalent check, but less efficient
>>> len(a & d) == 0
True

# This is even less efficient
>>> a & d == set()
True
```

Testing membership

The builtin in keyword searches for occurances

```
>>> 1 in a
True
>>> 6 in a
False
```

Length

The builtin len() function returns the number of elements in the set

```
>>> len(a)
4
>>> len(b)
3
```

Sets versus multisets

Sets are unordered collections of distinct elements. But sometimes we want to work with unordered collections of elements that are not necessarily distinct and keep track of the elements' multiplicities.

Consider this example:

```
>>> setA = {'a','b','b','c'}
>>> setA
set(['a', 'c', 'b'])
```

By saving the strings 'a' , 'b' , 'b' , 'c' into a set data structure we've lost the information on the fact that 'b' occurs twice. Of course saving the elements to a list would retain this information

```
>>> listA = ['a','b','b','c']
>>> listA
['a', 'b', 'b', 'c']
```

but a list data structure introduces an extra unneeded ordering that will slow down our computations.

For implementing multisets Python provides the Counter class from the collections module (starting from version 2.7):

Python 2.x ≥2.7

```
>>> from collections import Counter
>>> counterA = Counter(['a','b','b','c'])
>>> counterA
```

```
Counter({'b': 2, 'a': 1, 'c': 1})
```

`Counter` is a dictionary where where elements are stored as dictionary keys and their counts are stored as dictionary values. And as all dictionaries, it is an unordered collection.

Syntax

```
empty_set = set() # initialize an empty set
```

```
literal_set = {'foo', 'bar', 'baz'} # construct a set with 3 strings inside it
```

```
set_from_list = set(['foo', 'bar', 'baz']) # call the set function for a new set
```

```
set_from_iter = set(x for x in range(30)) # use arbitrary iterables to create a set
```

```
set_from_iter = {x for x in [random.randint(0,10) for i in range(10)]} # alternative notation
```

Parameters

Remarks

Sets are *unordered* and have *very fast lookup time* (amortized O(1) if you want to get technical). It is great to use when you have a collection of things, the order doesn't matter, and you'll be looking up items by name a lot. If it makes more sense to look up items by an index number, consider using a list instead. If order matters, consider a list as well.

Sets are *mutable* and thus cannot be hashed, so you cannot use them as dictionary keys or put them in other sets, or anywhere else that requires hashable types. In such cases, you can use an immutable frozenset .

The elements of a set must be *hashable* . This means that they have a correct `__hash__` method, that is consistent with `__eq__` . In general, mutable types such as `list` or `set` are not hashable and cannot be put in a set. If you encounter this problem, consider using `dict` and immutable keys.