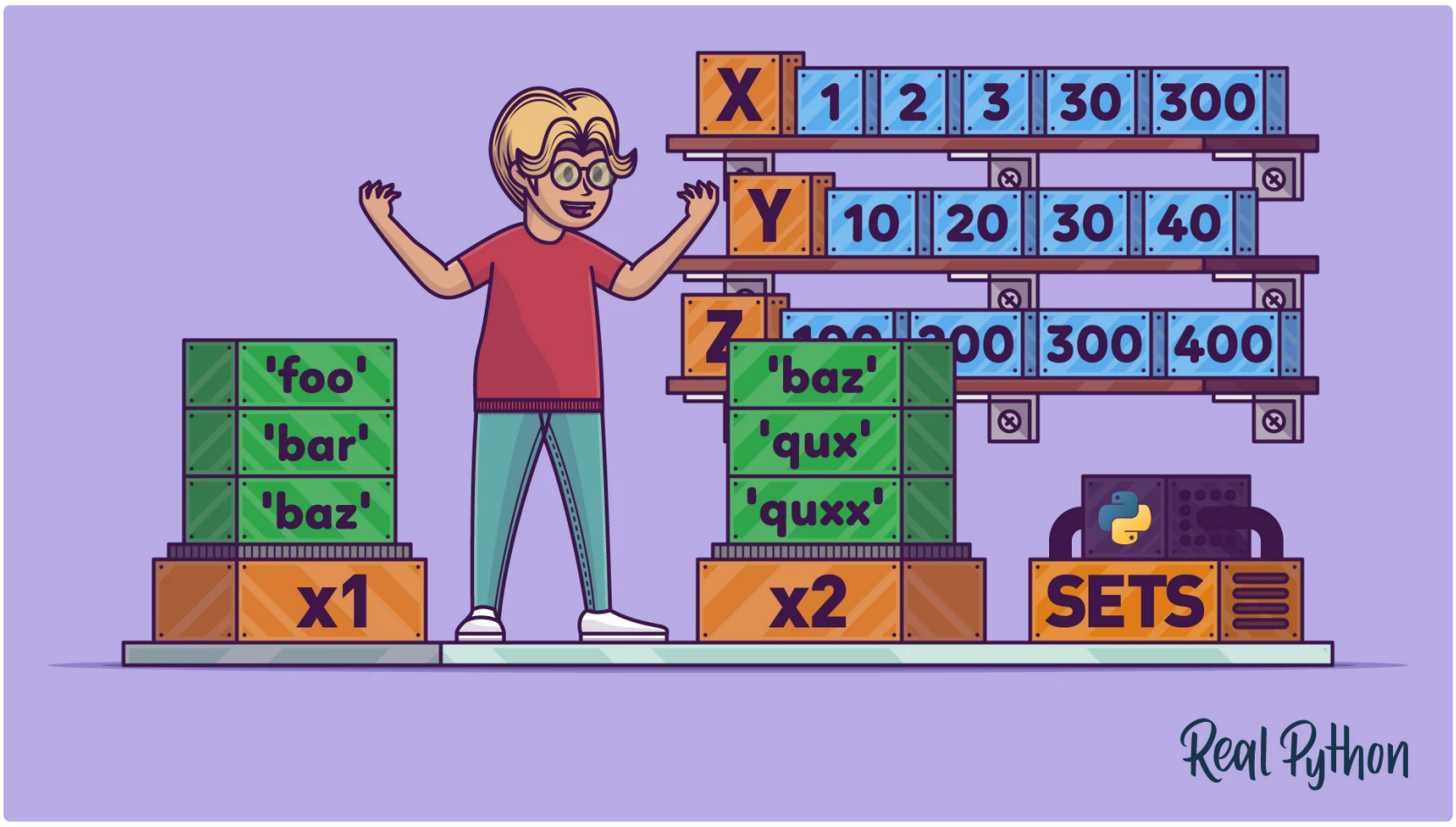


Sets in Python

by John Sturtz 16 Comments basics python

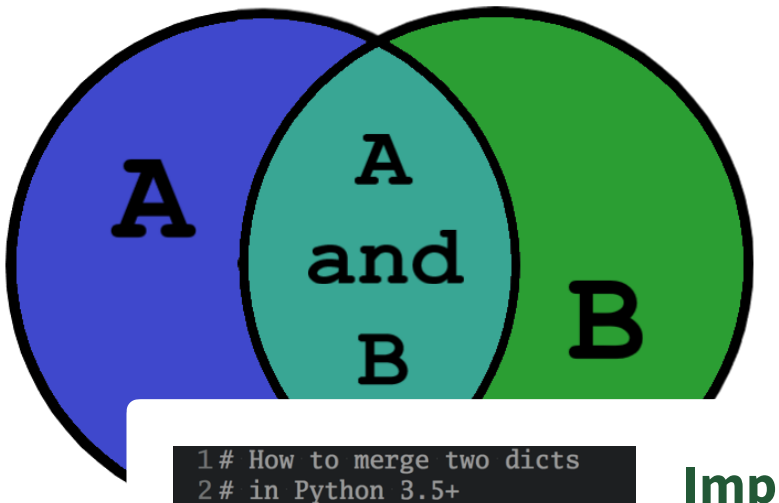
Mark as Completed

Tweet Share Email

Table of Contents

Watch Now This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: **Sets in Python**

Perhaps you recall learning about **sets** and **set theory** at some point in your mathematical education. Maybe you even remember Venn diagrams:



If this doesn't ring a bell, don't worry! Thi

In mathematics, a rigorous definition of a thought of simply as a well-defined colle

Grouping objects into a set can be useful Sets are distinguished from other object

Here's what you'll learn in this tutorial that they support. As with the earlier tutc should have a good feel for when a set is similar to sets except for one important detail.

```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```


> 🎓 **Take the Quiz:** Test your knowledge with our interactive "Python Sets" quiz. Upon completion you will receive a score so you can track your learning progress over time:
>
> Take the Quiz »

Defining a Set

Python's built-in `set` type has the following characteristics:

- Sets are unordered.
- Set elements are unique. Duplicate elements are not allowed.
- A set itself may be modified, but the elements contained in the set must be of an immutable type.

Let's see what all that means, and how you can work with sets in Python.

A set can be created in two ways. First, you can define a set with the built-in `set()` function:

Python

```
x = set(<iter>)
```

In this case, the argument `<iter>` is an iterable—again, for the moment, think list or tuple—that generates the list of objects to be included in the set. This is analogous to the `<iter>` argument given to the `.extend()` list method:

Python >>>

```
>>> x = set(['foo', 'bar', 'baz', 'foo', 'qux'])
>>> x
{'qux', 'foo', 'bar', 'baz'}
```

```
{'qux', 'foo', 'bar', 'baz'}

>>> x = set(('foo', 'bar', 'baz', 'foo', 'qux'))
>>> x
{'qux', 'foo', 'bar', 'baz'}
```

Strings are also iterable, so a string can be passed to `set()` as well. You have already seen that `list(s)` generates a list of the characters in the string `s`. Similarly, `set(s)` generates a set of the characters in `s`:

Python >>>

```
>>> s = 'quux'

>>> list(s)
['q', 'u', 'u', 'x']
>>> set(s)
{'x', 'u', 'q'}
```

You can see that the resulting sets are un
preserved. Additionally, duplicate values
examples and the letter `'u'` in the third.

Alternately, a set can be defined with cur

Python

```
x = {<obj>, <obj>, ..., <obj>}
```


When a set is defined this way, each `<obj>` becomes a distinct element of the set, even if it is an iterable. This behavior is similar to that of the `.append()` list method.

Thus, the sets shown above can also be defined like this:

Python >>>

```
>>> x = {'foo', 'bar', 'baz', 'foo', 'qux'}
>>> x
{'qux', 'foo', 'bar', 'baz'}

>>> x = {'q', 'u', 'u', 'x'}
>>> x
{'x', 'q', 'u'}
```

To recap:

- The argument to `set()` is an iterable. It generates a list of elements to be placed into the set.
- The objects in curly braces are placed into the set intact, even if they are iterable.

Observe the difference between these two set definitions:

Python >>>

```
>>> {'foo'}
{'foo'}

>>> set('foo')
{'o', 'f'}
```

A set can be empty. However, recall that Python interprets empty curly braces (`{}`) as an empty dictionary, so the only way to define an empty set is with the `set()` function:

Python >>>

```
>>> x = set()
>>> type(x)
<class 'set'>
>>> x
set()

>>> x = {}
```

```
>>> type(x)
<class 'dict'>
```

An empty set is falsy in a Boolean context:

Python >>>

```
>>> x = set()
>>> bool(x)
False
>>> x or 1
1
>>> x and 1
set()
```

You might think the most intuitive sets w

Python

```
>>> s1 = {2, 4, 6, 8, 10}
>>> s2 = {'Smith', 'McArthur', 'Wi
```

Python does not require this, though. Th

Python

```
>>> x = {42, 'foo', 3.14159, None}
>>> x
{None, 'foo', 42, 3.14159}
```

Don't forget that set elements must be immutable. For example, a tuple may be included in a set:

Python >>>

```
>>> x = {42, 'foo', (1, 2, 3), 3.14159}
>>> x
{42, 'foo', 3.14159, (1, 2, 3)}
```

But lists and dictionaries are mutable, so they can't be set elements:

Python >>>

```
>>> a = [1, 2, 3]
>>> {a}
Traceback (most recent call last):
  File "<pyshell#70>", line 1, in <module>
    {a}
TypeError: unhashable type: 'list'

>>> d = {'a': 1, 'b': 2}
>>> {d}
Traceback (most recent call last):
  File "<pyshell#72>", line 1, in <module>
    {d}
TypeError: unhashable type: 'dict'
```


Set Size and Membership

The `len()` function returns the number of elements in a set, and the `in` and `not in` operators can be used to test for membership:

Python >>>

```
>>> x = {'foo', 'bar', 'baz'}

>>> len(x)
3

>>> 'bar' in x
True
>>> 'qux' in x
False
```

Operating on a Set

Many of the operations that can be used f
example, sets can't be indexed or sliced. I
generally mimic the operations that are c

```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```


Operators vs. Methods

Most, though not quite all, set operations
method. Let's take a look at how these op

Given two sets, `x1` and `x2`, the union of `x1`

Consider these two sets:

Python

```
x1 = {'foo', 'bar', 'baz'}
x2 = {'baz', 'qux', 'quux'}
```

The union of `x1` and `x2` is `{'foo', 'bar', 'baz', 'qux', 'quux'}`.

> **Note:** Notice that the element `'baz'`, which appears in both `x1` and `x2`, appears only once in the union. Sets never contain duplicate values.

In Python, set union can be performed with the `|` operator:

Python >>>

```
>>> x1 = {'foo', 'bar', 'baz'}
>>> x2 = {'baz', 'qux', 'quux'}
>>> x1 | x2
{'baz', 'quux', 'qux', 'bar', 'foo'}
```

Set union can also be obtained with the `.union()` method. The method is invoked on one of the sets, and the other is passed as an argument:

Python >>>

```
>>> x1.union(x2)
{'baz', 'quux', 'qux', 'bar', 'foo'}
```

The way they are used in the examples above, the operator and method behave identically. But there is a subtle difference between them. When you use the `|` operator, both operands must be sets. The `.union()` method, on the other hand, will take any iterable as an argument, convert it to a set, and then perform the union.

Observe the difference between these two statements:

Python >>>

```
>>> x1 | ('baz', 'qux', 'quux')
Traceback (most recent call last):
```

```
  File "<pyshell#43>", line 1, in <module>
    x1 | ('baz', 'qux', 'quux')
TypeError: unsupported operand type(s) for |: 'set' and 'tuple'

>>> x1.union(('baz', 'qux', 'quux'))
{'baz', 'quux', 'qux', 'bar', 'foo'}
```

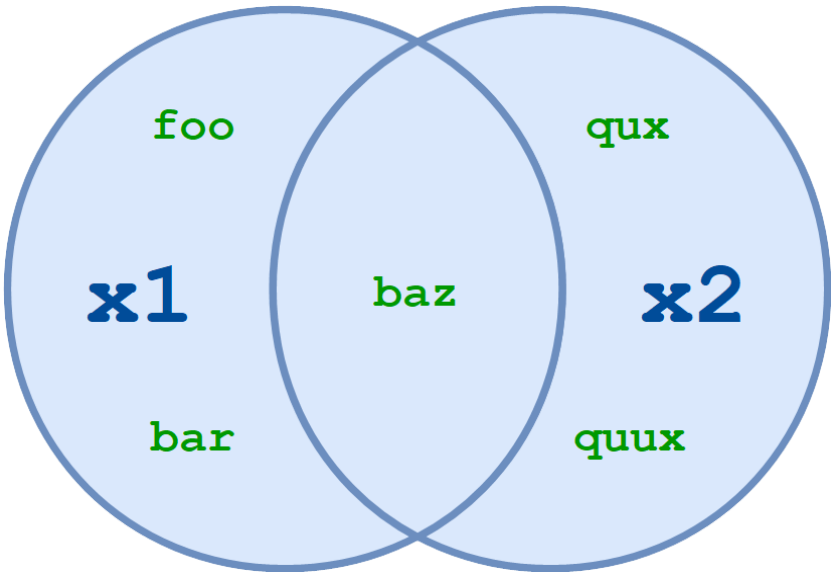
Both attempt to compute the union of `x1` and the tuple `('baz', 'qux', 'quux')`. This fails with the `|` operator but succeeds with the `.union()` method.

Available Operators and Me

Below is a list of the set operations availa
by both. The principle outlined above ge
iterable as an argument, but operators re

```
x1.union(x2[, x3 ...])
x1 | x2 [| x3 ...]
```

Compute the union of two or more sets.



Set Union

`x1.union(x2)` and `x1 | x2` both return the set of all elements in either `x1` or `x2`:

Python >>>

```
>>> x1 = {'foo', 'bar', 'baz'}
>>> x2 = {'baz', 'qux', 'quux'}

>>> x1.union(x2)
{'foo', 'qux', 'quux', 'baz', 'bar'}

>>> x1 | x2
{'foo', 'qux', 'quux', 'baz', 'bar'}
```

More than two sets may be specified with either the operator or the method:

Python >>>

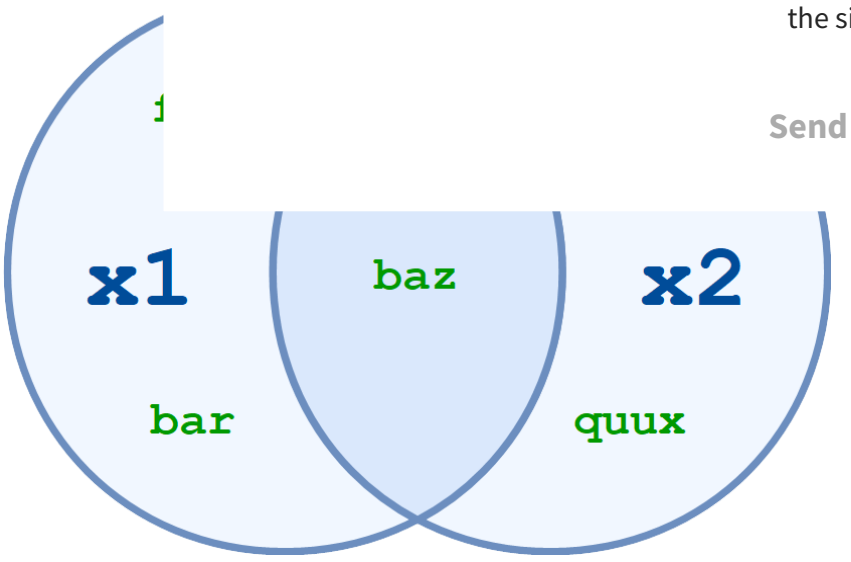
```
>>> a = {1, 2, 3, 4}
>>> b = {2, 3, 4, 5}
>>> c = {3, 4, 5, 6}
>>> d = {4, 5, 6, 7}

>>> a.union(b, c, d)
{1, 2, 3, 4, 5, 6, 7}

>>> a | b | c | d
{1, 2, 3, 4, 5, 6, 7}
```

The resulting set contains all elements th

```
x1.intersection(x2[, x3 ..
```

```
x1 & x2 [& x3 ...]
```

> Compute the intersection of two or m



Set Intersection

`x1.intersection(x2)` and `x1 & x2` return the set of elements common to both `x1` and `x2`:

Python >>>

```
>>> x1 = {'foo', 'bar', 'baz'}
>>> x2 = {'baz', 'qux', 'quux'}

>>> x1.intersection(x2)
{'baz'}

>>> x1 & x2
{'baz'}
```

You can specify multiple sets with the intersection method and operator, just like you can with set union:

Python >>>

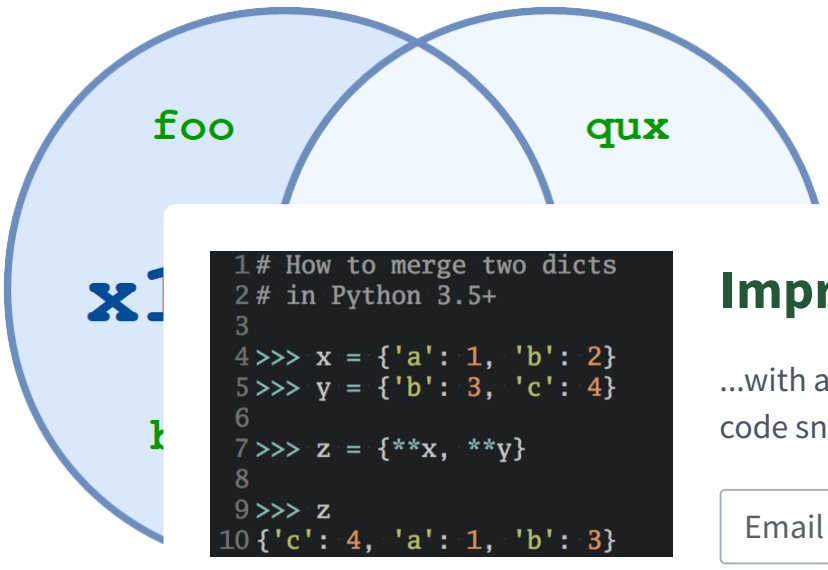
```
>>> a = {1, 2, 3, 4}
>>> b = {2, 3, 4, 5}
>>> c = {3, 4, 5, 6}
>>> d = {4, 5, 6, 7}

>>> a.intersection(b, c, d)
{4}

>>> a & b & c & d
{4}
```

The resulting set contains only elements that are present in all of the specified sets.

`x1.difference(x2[, x3 ...])`

`x1 - x2 [- x3 ...]`

> Compute the difference between two or more sets.



`x1.difference(x2)` and `x1 - x2` return the set that results when any elements in `x2` are removed or subtracted from `x1`.

Python

```
>>> x1 = {'foo', 'bar', 'baz'}
>>> x2 = {'baz', 'qux', 'quux'}

>>> x1.difference(x2)
{'foo', 'bar'}

>>> x1 - x2
{'foo', 'bar'}
```

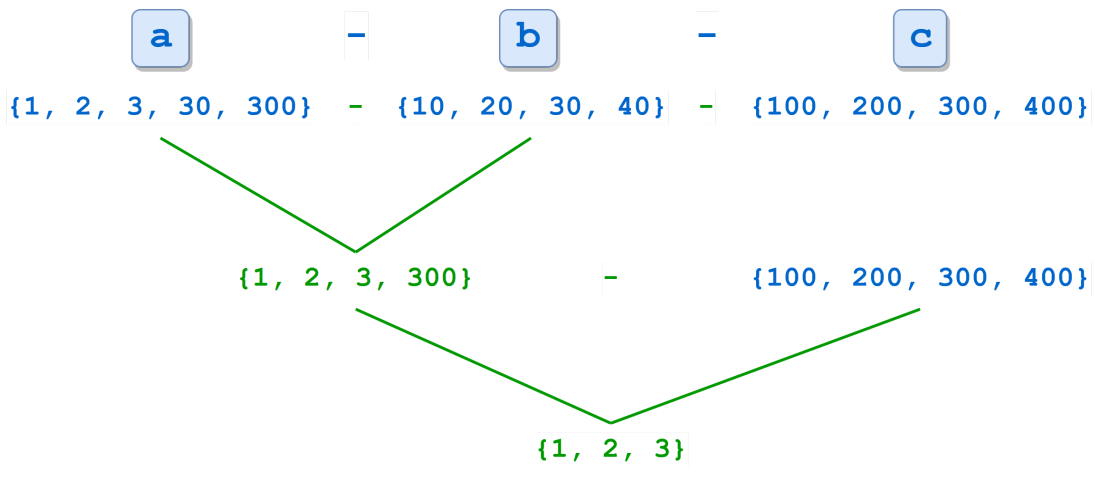
Another way to think of this is that `x1.difference(x2)` and `x1 - x2` return the set that results when any elements in `x2` are removed or subtracted from `x1`.

Once again, you can specify more than two sets:

Python >>>

```
>>> a = {1, 2, 3, 30, 300}
>>> b = {10, 20, 30, 40}
>>> c = {100, 200, 300, 400}

>>> a.difference(b, c)
{1, 2, 3}

>>> a - b - c
{1, 2, 3}
```

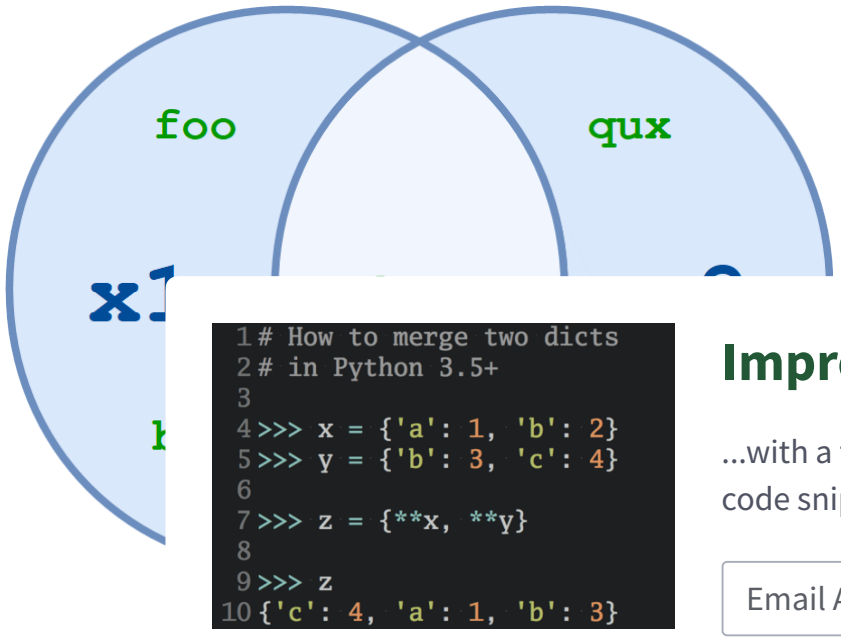
When multiple sets are specified, the operation is performed from left to right. In the example above, `a - b` is computed first, resulting in `{1, 2, 3, 300}`. Then `c` is subtracted from that set, leaving `{1, 2, 3}`:



`x1.symmetric_difference(x2)`

`x1 ^ x2 [^ x3 ...]`

> Compute the symmetric difference between sets.



`x1.symmetric_difference(x2)` and `x1 ^ x2` return a new set containing only the elements that are in exactly one of the sets:

Python

```
>>> x1 = {'foo', 'bar', 'baz'}
>>> x2 = {'baz', 'qux', 'quux'}

>>> x1.symmetric_difference(x2)
{'foo', 'qux', 'quux', 'bar'}

>>> x1 ^ x2
{'foo', 'qux', 'quux', 'bar'}
```

The `^` operator also allows more than two sets:

Python >>>

```
>>> a = {1, 2, 3, 4, 5}
>>> b = {10, 2, 3, 4, 50}
>>> c = {1, 50, 100}

>>> a ^ b ^ c
{100, 5, 10}
```

As with the difference operator, when multiple sets are specified, the operation is performed from left to right.

Curiously, although the `^` operator allows multiple sets, the `.symmetric_difference()` method doesn't:

Python >>>

```
>>> a = {1, 2, 3, 4, 5}
>>> b = {10, 2, 3, 4, 50}
>>> c = {1, 50, 100}

>>> a.symmetric_difference(b, c)
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    a.symmetric_difference(b, c)
TypeError: symmetric_difference() takes exactly one argument (2 given)
```

`x1.isdisjoint(x2)`

> Determines whether or not two sets have any elements in common.

`x1.isdisjoint(x2)` returns `True` if `x1` and `x2` have no elements in common:

Python >>>

```
>>> x1 = {'foo', 'bar', 'baz'}
>>> x2 = {'baz', 'qux', 'quux'}

>>> x1.isdisjoint(x2)
False

>>> x2 - {'baz'}
{'quux', 'qux'}
>>> x1.isdisjoint(x2 - {'baz'})
True
```

If `x1.isdisjoint(x2)` is `True`, then `x1 &`

Python >>>

```
>>> x1 = {1, 3, 5}
>>> x2 = {2, 4, 6}

>>> x1.isdisjoint(x2)
True
>>> x1 & x2
set()
```

Note: There is no operator that corres

`x1.issubset(x2)`

`x1 <= x2`

> Determine whether one set is a subset of the other.

In set theory, a set `x1` is considered a subset of another set `x2` if every element of `x1` is in `x2`.

`x1.issubset(x2)` and `x1 <= x2` return `True` if `x1` is a subset of `x2`:

Python >>>

```
>>> x1 = {'foo', 'bar', 'baz'}
>>> x1.issubset({'foo', 'bar', 'baz', 'qux', 'quux'})
True

>>> x2 = {'baz', 'qux', 'quux'}
>>> x1 <= x2
False
```

A set is considered to be a subset of itself:

Python >>>

```
>>> x = {1, 2, 3, 4, 5}
>>> x.issubset(x)
True
>>> x <= x
True
```

It seems strange, perhaps. But it fits the definition—every element of `x` is in `x`.

`x1 < x2`

> Determines whether one set is a proper subset of the other.

A proper subset is the same as a subset, except that the sets can't be identical. A set `x1` is considered a proper subset of another set `x2` if every element of `x1` is in `x2`, and `x1` and `x`

`x1 < x2` returns `True` if `x1` is a proper subset of `x2`:

Python >>>

```
>>> x1 = {'foo', 'bar'}
>>> x2 = {'foo', 'bar', 'baz'}
>>> x1 < x2
True

>>> x1 = {'foo', 'bar', 'baz'}
>>> x2 = {'foo', 'bar', 'baz'}
>>> x1 < x2
False
```

While a set is considered a subset of itself

Python >>>

```
>>> x = {1, 2, 3, 4, 5}
>>> x <= x
True
>>> x < x
False
```

Note: The < operator is the only way to

`x1.issuperset(x2)`

`x1 >= x2`

> Determine whether one set is a superset of the other.

A superset is the reverse of a subset. A set `x1` is considered a superset of another set `x2` if `x1` contains every element of `x2`.

`x1.issuperset(x2)` and `x1 >= x2` return `True` if `x1` is a superset of `x2`:

Python >>>

```
>>> x1 = {'foo', 'bar', 'baz'}

>>> x1.issuperset({'foo', 'bar'})
True

>>> x2 = {'baz', 'qux', 'quux'}
>>> x1 >= x2
False
```

You have already seen that a set is considered a subset of itself. A set is also considered a superset of itself:

Python >>>

```
>>> x = {1, 2, 3, 4, 5}
>>> x.issuperset(x)
True
>>> x >= x
True
```

`x1 > x2`

> Determines whether one set is a proper superset of the other.

A proper superset is the same as a superset, except that the sets can't be identical. A set `x1` is considered a proper superset of another set `x2` if `x1` contains every element of `x2`

`x1 > x2` returns `True` if `x1` is a proper superset of `x2`:

Python >>>

```
>>> x1 = {'foo', 'bar', 'baz'}
>>> x2 = {'foo', 'bar'}
>>> x1 > x2
True

>>> x1 = {'foo', 'bar', 'baz'}
>>> x2 = {'foo', 'bar', 'baz'}
>>> x1 > x2
False
```

A set is not a proper superset of itself:

Python >>>

```
>>> x = {1, 2, 3, 4, 5}
>>> x > x
False
```

Note: The > operator is the only way to
method.

Modifying a Set

Although the elements contained in a set must be of immutable type, sets themselves can be modified. Like the operations above, there are a mix of operators and methods that can be used to change the contents of a set.

Augmented Assignment Operators and Methods

Each of the union, intersection, difference, and symmetric difference operators listed above has an augmented assignment form that can be used to modify a set. For each, there is a corresponding method as well.

`x1.update(x2[, x3 ...])`
`x1 |= x2 [| x3 ...]`

Modify a set by union.

`x1.update(x2)` and `x1 |= x2` add to `x1` any elements in `x2` that `x1` does not already have:

Python >>>

```
>>> x1 = {'foo', 'bar', 'baz'}
>>> x2 = {'foo', 'baz', 'qux'}

>>> x1 |= x2
>>> x1
{'qux', 'foo', 'bar', 'baz'}

>>> x1.update(['corge', 'garply'])
>>> x1
{'qux', 'corge', 'garply', 'foo', 'bar', 'baz'}
```


`x1.intersection_update(x2[, x3 ...])`

x1.intersection_update(x2[, x3 ...])

x1 &= x2 [& x3 ...]

> Modify a set by intersection.

`x1.intersection_update(x2)` and `x1 &= x2` update `x1`, retaining only elements found in both `x1` and `x2`:

Python

```
>>> x1 = {'foo', 'bar', 'baz'}
>>> x2 = {'foo', 'baz', 'qux'}

>>> x1 &= x2
>>> x1
{'foo', 'baz'}

>>> x1.intersection_update(['baz',
>>> x1
{'baz'}
```


x1.difference_update(x2[, x3 ...])

x1 -= x2 [| x3 ...]

> Modify a set by difference.

`x1.difference_update(x2)` and `x1 -= x2` update `x1`, removing elements found in `x2`:

Python >>>

```
>>> x1 = {'foo', 'bar', 'baz'}
>>> x2 = {'foo', 'baz', 'qux'}

>>> x1 -= x2
>>> x1
{'bar'}

>>> x1.difference_update(['foo', 'bar', 'qux'])
>>> x1
set()
```

x1.symmetric_difference_update(x2)

x1 ^= x2

> Modify a set by symmetric difference.

`x1.symmetric_difference_update(x2)` and `x1 ^= x2` update `x1`, retaining elements found in either `x1` or `x2`, but not both:

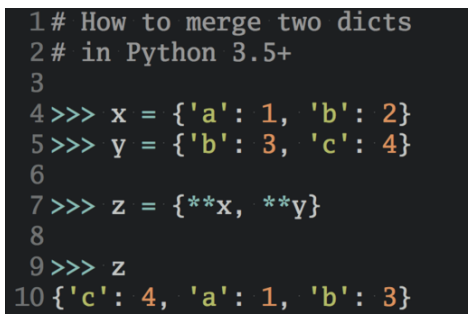
Python >>>

```
>>> x1 = {'foo', 'bar', 'baz'}
>>> x2 = {'foo', 'baz', 'qux'}
>>>
```

```
>>> x1 ^= x2
>>> x1
{'bar', 'qux'}
>>>
>>> x1.symmetric_difference_update(['qux', 'corge'])
>>> x1
{'bar', 'corge'}
```


Other Methods For Modifyi

Aside from the augmented operators abc



```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```


`x.add(<elem>)`

> Adds an element to a set.

`x.add(<elem>)` adds `<elem>`, which must

Python >>>

```
>>> x = {'foo', 'bar', 'baz'}

>>> x.add('qux')
>>> x
{'bar', 'baz', 'foo', 'qux'}
```

`x.remove(<elem>)`

> Removes an element from a set.

`x.remove(<elem>)` removes `<elem>` from `x`. Python raises an exception if `<elem>` is not in `x`:

Python >>>

```
>>> x = {'foo', 'bar', 'baz'}

>>> x.remove('baz')
>>> x
{'bar', 'foo'}

>>> x.remove('qux')
Traceback (most recent call last):
  File "<pyshell#58>", line 1, in <module>
    x.remove('qux')
KeyError: 'qux'
```

`x.discard(<elem>)`

> Removes an element from a set.

`x.discard(<elem>)` also removes `<elem>` from `x`. However, if `<elem>` is not in `x`, this method quietly does nothing instead of raising an exception:

Python >>>

```
>>> x = {'foo', 'bar', 'baz'}

>>> x.discard('baz')
>>> x
{'bar', 'foo'}

>>> x.discard('qux')
>>> x
{'bar', 'foo'}
```

```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```


`x.pop()`

> Removes a random element from a se

`x.pop()` removes and returns an arbitrarily chosen element from `x`. If `x` is empty, `x.pop()` raises an exception:

Python >>>

```
>>> x = {'foo', 'bar', 'baz'}

>>> x.pop()
'bar'
>>> x
{'baz', 'foo'}

>>> x.pop()
'baz'
>>> x
{'foo'}

>>> x.pop()
'foo'
>>> x
set()

>>> x.pop()
Traceback (most recent call last):
  File "<pyshell#82>", line 1, in <module>
    x.pop()
KeyError: 'pop from an empty set'
```

`x.clear()`

> Clears a set.

`x.clear()` removes all elements from `x`:

Python >>>

```
>>> x = {'foo', 'bar', 'baz'}
>>> x
{'foo', 'bar', 'baz'}
>>>
>>> x.clear()
>>> x
```

```
set()
```


Frozen Sets

Python provides another built-in type cal
frozenset is immutable. You can perform

Python

```
>>> x = frozenset(['foo', 'bar', '
>>> x
frozenset({'foo', 'baz', 'bar'})

>>> len(x)
3

>>> x & {'baz', 'qux', 'quux'}
frozenset({'baz'})
```

```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```


But methods that attempt to modify a frozenset fail:

Python >>>

```
>>> x = frozenset(['foo', 'bar', 'baz'])

>>> x.add('qux')
Traceback (most recent call last):
  File "<pyshell#127>", line 1, in <module>
    x.add('qux')
AttributeError: 'frozenset' object has no attribute 'add'

>>> x.pop()
Traceback (most recent call last):
  File "<pyshell#129>", line 1, in <module>
    x.pop()
AttributeError: 'frozenset' object has no attribute 'pop'

>>> x.clear()
Traceback (most recent call last):
  File "<pyshell#131>", line 1, in <module>
    x.clear()
AttributeError: 'frozenset' object has no attribute 'clear'

>>> x
frozenset({'foo', 'bar', 'baz'})
```

Deep Dive: Frozensets and Augmented Assignment

Since a frozenset is immutable, you might think it can't be the target of an augmented assignment operator. But observe:

Python >>>

```
>>> f = frozenset(['foo', 'bar', 'baz'])
>>> s = {'baz', 'qux', 'quux'}

>>> f &= s
>>> f
frozenset({'baz'})
```

What gives?

Python does not perform augmented
equivalent to `x = x & s`. It isn't modi
originally referenced is gone.

You can verify this with the `id()` funct

Python >>>

```
>>> f = frozenset(['foo', 'bar'
>>> id(f)
56992872
>>> s = {'baz', 'qux', 'quux'}

>>> f &= s
>>> f
frozenset({'baz'})
>>> id(f)
56992152
```

`f` has a different integer identifier following the augmented assignment. It has been reassigned, not modified in place.

Some objects in Python are modified in place when they are the target of an augmented assignment operator. But frozensets aren't.

Frozensets are useful in situations where you want to use a set, but you need an immutable object. For example, you can't define a set whose elements are also sets, because set elements must be immutable:

Python >>>

```
>>> x1 = set(['foo'])
>>> x2 = set(['bar'])
>>> x3 = set(['baz'])
>>> x = {x1, x2, x3}
Traceback (most recent call last):
  File "<pyshell#38>", line 1, in <module>
    x = {x1, x2, x3}
TypeError: unhashable type: 'set'
```

If you really feel compelled to define a set of sets (hey, it could happen), you can do it if the elements are frozensets, because they are immutable:

Python >>>

```
>>> x1 = frozenset(['foo'])
>>> x2 = frozenset(['bar'])
>>> x3 = frozenset(['baz'])
>>> x = {x1, x2, x3}
>>> x
{frozenset({'bar'}), frozenset({'baz'}), frozenset({'foo'})}
```

Likewise, recall from the previous tutorial on dictionaries that a dictionary key must be immutable. You can't use the built-in set type as a dictionary key:

Python >>>

```
>>> x = {1, 2, 3}
>>> y = {'a', 'b', 'c'}
>>>
>>> d = {x: 'foo', y: 'bar'}
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    d = {x: 'foo', y: 'bar'}
TypeError: unhashable type: 'set'
```

If you find yourself needing to use sets as

Python

```
>>> x = frozenset({1, 2, 3})
>>> y = frozenset({'a', 'b', 'c'})
>>>
>>> d = {x: 'foo', y: 'bar'}
>>> d
{frozenset({1, 2, 3}): 'foo', froz
```

```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```


Conclusion

In this tutorial, you learned how to define **set** objects in Python, and you became familiar with the functions, operators, and methods that can be used to work with sets.

You should now be comfortable with the basic built-in data types that Python provides.

Next, you will begin to explore how the code that operates on those objects is organized and structured in a Python program.

🎓 **Take the Quiz:** Test your knowledge with our interactive "Python Sets" quiz. Upon completion you will receive a score so you can track your learning progress over time:

Take the Quiz »

« Dictionaries in Python

Sets in Python

Python Program Lexical Structure »

```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```

About **John Sturtz**

Joh

» M


```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```


Each tutorial at Real Python is created *who worked on this tutorial are:*

David

Joanna

What Do You Think?

Rate this article:

What's your #1 takeaway or favorite thing you learned? How are you going to put your newfound skills to use? Leave a comment below and let us know.

Commenting Tips: The most useful comments are those written with the goal of learning from or helping out other students. Get tips for asking good questions and get answers to common questions in our support portal.

Looking for a real-time conversation? Visit the Real Python Community Chat or join the next "Office Hours" Live Q&A Session. Happy Pythoning!

Keep Learning

Related Tutorial Categories: basics py

Recommended Video Course: Sets in P

All Tutorial Topics

advanced api basics best-practices community databases data-science devops django docker flask front-end gamedev gui intermediate machine-learning projects python testing tools web-dev web-scraping

Table of Con


```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```