

# Sets in Python

by Leodanis Pozo Ramos  May 05, 2025  47m  21 Comments

 basics python

Mark as Completed



↑ Share

## Table of Contents

- Getting Started With Python's set Data Type
- Building Sets in Python
  - Creating Sets Through Literals
  - Using the set() Constructor

— FREE Email Series —



```
1 # How to merge two dicts
2 # in Python 3.5+
3
4 >>> x = {'a': 1, 'b': 2}
5 >>> y = {'b': 3, 'c': 4}
6
7 >>> z = {**x, **y}
8
9 >>> z
10 {'c': 4, 'a': 1, 'b': 3}
```

Email...

Get Python Tricks »

 No spam. Unsubscribe any time.

 Browse Topics  Guided Learning Paths

 Basics  Intermediate  Advanced

ai algorithms api best-practices career

community databases data-science

data-structures data-viz devops django

docker editors flask front-end gamedev

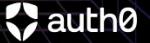
gui machine-learning news numpy

- Using Set Comprehensions
- Performing Common Set Operations
  - Union
  - Intersection
  - Difference
  - Symmetric Difference
- Using Augmented Set Operations
  - Augmented Union
  - Augmented Intersection
  - Augmented Difference
  - Augmented Symmetric Difference
- Comparing Sets
  - Subsets
  - Proper Subsets
  - Supersets
  - Proper Supersets
  - Disjoint Sets
- Using Other Set Methods
  - Adding an Element With `.add()`
  - Removing an Existing Element With `.remove()`
  - Deleting an Existing or Missing Element With `.discard()`
  - Removing and Returning an Element With `.pop()`
  - Removing All Elements With `.clear()`
  - Creating Shallow Copies of Sets With `.copy()`
- Traversing Sets
  - Accessing and Modifying Elements in a Loop
  - Processing and Removing Elements in a Loop
  - Iterating Through a Sorted Set
- Exploring Other Set Capabilities
  - Finding the Number of Elements With `len()`
  - Running Membership Tests on Sets

projects python stdlib testing tools  
web-dev web-scraping

```
auth0-b2b-saas-starter@1.0.0 auth0:bootstrap
> node ./scripts/bootstrap.mjs
✓ Checking that the Auth0 CLI has been installed
✓ Initialize tenant settings
✓ Configuring prompt settings
✓ Creating SaaStart Management client
✓ Creating Management API Client Grant
✓ Creating SaaStart Dashboard client
✓ Creating SaaStart-Shared-Database connection
✓ Creating admin role
✓ Creating member role
✓ Creating Security Policies Action
✓ Creating Add Default Role Action
✓ Creating Add Role to Tokens Action
✓ Updating trigger bindings for Actions
✓ Saving environment variables to .env.local
✓ Creating theme for Universal Login
```

TIL you can manage and automate your auth directly from the command line with the Auth0 Deploy CLI.

Try free today → 

## Table of Contents

- Getting Started With Python's set Data Type
- Building Sets in Python
- Performing Common Set Operations
- Using Augmented Set Operations
- Comparing Sets
- Using Other Set Methods
- Traversing Sets
- Exploring Other Set Capabilities
- Conclusion
- Frequently Asked Questions

Mark as Completed



- Conclusion
- Frequently Asked Questions



Share



Secure access for developers.  
But not attackers.

Try free today

Remove ads



This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: [Using Sets in Python](#)

Recommended Video Course

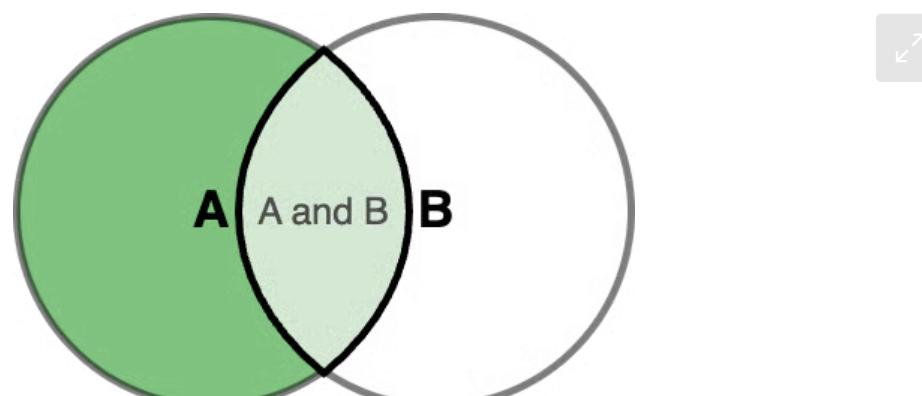
Using Sets in Python



High Quality  
Python Video Courses

[Watch Now »](#)

Python provides a built-in set data type. It differs from other built-in data types in that it's an unordered collection of unique elements. It also supports operations that differ from those of other data types. You might recall learning about sets and set theory in math class. Maybe you even remember Venn diagrams:



Venn Diagram

In mathematics, the definition of a set can be abstract and difficult to grasp. In practice, you can think of a set as a well-defined collection of unique objects, typically called **elements** or

**members.** Grouping objects in a set can be pretty helpful in programming. That's why Python has sets built into the language.

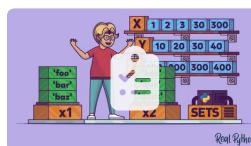
### By the end of this tutorial, you'll understand that:

- A **set** is an unordered collection of unique, hashable elements.
- The **set() constructor** works by converting any iterable into a set, removing duplicate elements in the process.
- You can **initialize a set** using literals, the set() constructor, or comprehensions.
- Sets are **unordered** because they don't maintain a specific order of elements.
- Sets are useful when you need to run **set operations**, **remove duplicates**, run **efficient membership tests**, and more.

In this tutorial, you'll dive deep into the features of Python sets and explore topics like set creation and initialization, common set operations, set manipulation, and more.

**Get Your Code:** Click here to download the free sample code that shows you how to work with sets in Python.

 **Take the Quiz:** Test your knowledge with our interactive “Python Sets” quiz. You’ll receive a score upon completion to help you track your learning progress:



#### Interactive Quiz Python Sets

In this quiz, you'll assess your understanding of Python's built-in set data type. You'll revisit the definition of unordered, unique, hashable collections, how to create and initialize sets, and key set operations.

## Getting Started With Python's set Data Type

Python's built-in set data type is a [mutable](#) and unordered collection of unique and [hashable](#) elements. In this definition, the qualifiers mean the following:

- **Mutable:** You can add or remove elements from an existing set.
- **Unordered:** A set doesn't maintain any particular order of its elements.
- **Unique elements:** Duplicate elements aren't allowed.
- **Hashable elements:** Each element must have a [hash value](#) that stays the same for its entire lifetime.

As with other mutable data types, you can modify sets by increasing or decreasing their [size](#) or number of elements. To this end, sets provide a series of handy methods that allow you to add and remove elements to and from an existing set.

The elements of a set must be unique. This feature makes sets especially useful in scenarios where you need to remove duplicate elements from an existing [iterable](#), such as a [list](#) or [tuple](#):

```
Python
>>> numbers = [1, 2, 2, 2, 3, 4, 5, 5]
>>> set(numbers)
{1, 2, 3, 4, 5}
```

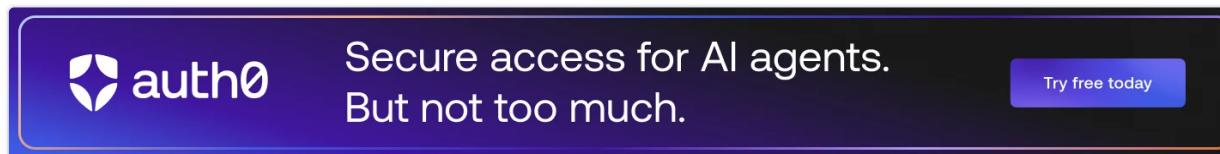
In practice, removing duplicate items from an iterable might be one of the most useful and commonly used features of sets.

Python implements sets as [hash tables](#). A great feature of hash tables is that they make lookup operations almost instantaneous. Because of this, sets are exceptionally efficient in membership operations with the [in](#) and [not in](#) operators.

Finally, Python sets support common [set operations](#), such as [union](#), [intersection](#), [difference](#), [symmetric difference](#), and others. This feature makes them useful when you need to do some of the following tasks:

- **Find common elements** in two or more sets
- **Find differences** between two or more sets
- **Combine multiple sets** together while avoiding duplicates

As you can see, set is a powerful data type with characteristics that make it useful in many contexts and situations. Throughout the rest of this tutorial, you'll learn more about the features that make sets a worthwhile addition to your programming toolkit.



The banner features the Auth0 logo (a white diamond shape with four smaller diamonds inside) and the text "Secure access for AI agents. But not too much." Below the text is a purple button labeled "Try free today".

 Remove ads

## Building Sets in Python

To use a set, you first need to create it. You'll have different ways to build sets in Python. For example, you can create them using one of the following techniques:

- Set [literals](#)
- The `set()` constructor
- A [set comprehension](#)

In the following sections, you'll learn how to use the three approaches listed above to create new sets in Python. You'll start with set literals.

### Creating Sets Through Literals

You can define a new set by providing a comma-separated series of hashable objects within curly braces {} as shown below:

Python Syntax

```
{obj_0[, obj_1, ..., obj_n]}
```

The portion of this syntax enclosed in square brackets is optional. This means you can build a set with a single item if you remove the code inside the square brackets:

Python

```
>>> # Single-element set
>>> hex_colors = {"#33FF57"}
>>> hex_colors
{'#33FF57'}
```

```
>>> # Multiple-element set
>>> hex_colors = {
...     "#33FF57",  # Green
...     "#3357FF",  # Blue
...     "#F1C40F",  # Yellow
...     "#E74C3C",  # Red
... }
>>> hex_colors
{'#F1C40F', '#33FF57', '#E74C3C', '#3357FF'}
```

In the first example, you create a set with a single element representing a color in hexadecimal notation. Later, you define the set with four elements.

**Note:** You can't use a literal to create an empty set because an empty pair of curly braces creates a dictionary. More about this topic in a moment.

When you define a set following the literal syntax, each object becomes a distinct element. This syntax won't [unpack](#) iterable objects. For example, if you use a tuple, then it'll be added to the set as a tuple rather than a series of elements:

Python

```
>>> rgb_colors = {
...     (51, 255, 87), # Green
...     (51, 87, 255), # Blue
...     (241, 196, 15), # Yellow
...     (231, 76, 60), # Red
... }

>>> rgb_colors
{
    (51, 255, 87),
    (241, 196, 15),
    (51, 87, 255),
    (231, 76, 60)
}
```

In this example, you create a set of colors expressed as three-value tuples that follow the [RGB](#) color model. Note that the resulting set contains tuples. It doesn't unpack the tuples' content.

In practice, most sets will contain similar objects—for example, even [numbers](#) or surnames:

```
Python >_

>>> {2, 4, 6, 8, 10, 8, 2}
{2, 4, 6, 8, 10}

>>> {"Smith", "McArthur", "Wilson", "Johansson", "Smith"}
{'Johansson', 'McArthur', 'Wilson', 'Smith'}
```

Holding objects of homogeneous data type or similar semantics isn't a requirement for Python sets. Additionally, note how in both examples, the resulting set removed the duplicate. This behavior guarantees that the resulting set contains only unique elements.

The elements of a set can be objects of different data types:

```
Python >_
```

```
>>> {42, "Hi!", 3.14159, None, "Python"}  
{None, 42, 3.14159, 'Hi!', 'Python'}
```

In this example, your set contains numbers, strings, and even the `None` object. Even though sets can store objects of different data types, it's common to have sets of semantically equivalent objects, such as colors, letters, surnames, and so on. This kind of set, in most cases, results in objects of the same data type.

It's also important to remember that set elements must be hashable. For example, you can include a tuple in a set, as you already learned, but you can't include a list because lists aren't hashable:

```
Python  
  
>>> rgb_colors = {  
...     [51, 255, 87], # Green  
...     [51, 87, 255], # Blue  
...     [241, 196, 15], # Yellow  
...     [231, 76, 60], # Red  
... }  
Traceback (most recent call last):  
...  
TypeError: unhashable type: 'list'
```

Apart from lists, you can't use `dictionaries`, `bytearray` objects, or other sets as the elements of a set. These data types are mutable, and therefore, they can't be hashable.

A special case occurs with tuples. If you have tuples containing hashable values, then you can add those tuples to a set. However, if your tuples contain unhashable values, then you can't add them to a set:

```
Python  
  
>>> students = {  
...     ("Jane", 18, ["Math", "Physics", "History"]),
```

```
...     ("John", 19, ["English", "History", "Philosophy"]),
...
Traceback (most recent call last):
...
TypeError: unhashable type: 'list'
```

In this example, you have tuples holding students' data. Inside each tuple, you have a list of subjects. Lists are unhashable, so they can't be added to sets even if they're contained in a tuple.



[i Remove ads](#)

## Using the set( ) Constructor

You can also build sets using the `set()` [constructor](#). When would you use this approach? For example, there's no literal for creating empty sets because an empty pair of curly braces creates an empty dictionary. If you want to create an empty set, then you have to use the constructor:

```
Python
>>> empty = set()
>>> empty
set()
```

Calling the `set` constructor is the most straightforward and readable way to create an empty set. Note that instead of getting an empty pair of curly braces in the output, you get `set()` to avoid confusion with empty dictionaries.

The syntax to build a set using the constructor is shown below:

## Python Syntax

```
set([iterable])
```

The argument `iterable` is optional and must be an iterable like a list or tuple. For example, the following code builds a set from a list:

### Python

```
>>> set([1, 2, 2, 3, 4, 4, 5])
{1, 2, 3, 4, 5}
```

In this example, you pass a list of numbers to `set()`. Note how the originally duplicate values are only represented once in the resulting set.

Python [strings](#) are also iterable, so you can pass a string to `set()`. This will generate a set of characters in the input string:

### Python

```
>>> language = "Python"
>>> set(language)
{'y', 'P', 'o', 't', 'n', 'h'}
```

The resulting set of characters is unordered. The original order, as specified in the string, isn't preserved.

There's a subtle difference between using the `set()` constructor and the literal syntax. Observe the difference with a quick example:

### Python

```
>>> set("Hello!")
{'H', 'l', 'e', '!', 'o'}
```

```
>>> {"Hello!"}  
{'Hello!'}
```

When you create sets using the `set()` constructor, the argument must be iterable. The constructor unpacks the iterable and adds its items to the resulting set as individual elements. In contrast, the set literal syntax adds the iterable as a single element to the set. It doesn't unpack its items.

## Using Set Comprehensions

A **set comprehension** is a concise way to create a set by evaluating an expression over an iterable. It works similarly to [list comprehensions](#) but automatically removes duplicates.

Here's the syntax for a set comprehension:

### Python Syntax

```
{expression for member in iterable [if condition]}
```

The set comprehension syntax consists of the following key components:

- The **enclosing brackets** (`{}`) define the set comprehension.
- The comprehension **expression** provides an element in each iteration.
- The current **member** represents the current item or value in the iterable.
- The **iterable** can be any iterable object, including a [list](#), [tuple](#), [set](#), [generator](#), or similar type.
- The **[if condition]** part is an optional conditional that you can use to [filter](#) existing collections or generate elements conditionally.

A set comprehension returns a new set. So, you use them to create, transform, and filter sets, which are essential skills for any Python programmer.

**Note:** To dive deeper into set comprehensions, check out the [Python Set Comprehensions: How and When to Use Them](#) tutorial.

To illustrate how to use a set comprehension, say that you have a list of usernames initially entered without validation. You need to clean the list by converting all the usernames to lowercase, removing leading and trailing spaces, and removing duplicates. You can do this with a set comprehension:

Python

```
>>> usernames = [
...     "Alice",
...     " bob",
...     "ALICE  ",
...     "Bob",
...     "charlie",
...     "Charlie",
...     "JOHN"
... ]

>>> {name.lower().strip() for name in usernames}
{'bob', 'alice', 'john', 'charlie'}
```

In this example, the set comprehension processes a list of strings. It converts each value into lowercase and removes any leading and trailing spaces. It also eliminates duplicates. This way, you have a new set of clean usernames.

Your Weekly Dose of All Things Python!

[pycoders.com](http://pycoders.com)



[Remove ads](#)

## Performing Common Set Operations

With PDFmyURL you can [easily save entire websites as PDF](#) and use our API service to [create PDFs automatically from webpages or HTML](#).

Python's set data type provides a host of operations that are based on the [operations](#) defined for mathematical sets. You can perform most set operations in two different ways:

1. Using an operator
2. Calling a method

In the following sections, you'll explore both approaches and their differences. To kick things off, you'll start with set union, which is one of the most common operations that you'll perform on sets.

## Union

The **union** of two sets returns a new set that contains all unique elements from both sets. You can perform a union using the `|` operator or the `.union()` method. Any duplicate elements are automatically removed, so each item appears only once in the result. Union is a great way to keep all the elements without unwanted duplication.

The following diagram provides a visual representation of the union operation between sets  $A$  and  $B$ :

Set Union

The darkest area in the center of the diagram represents the elements that were common to  $A$  and  $B$  and now are unique in the resulting set.

Here's the syntax for the union operation, using both the operator and the method:

#### Python Syntax

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

In this syntax, the portion between square brackets is optional. The union operator (`|`) is a binary operator, which means it operates on two operands. The `.union()` method can take any number of arguments.

To illustrate how the union operation works, say that you have the following sets:

#### Python

```
>>> pet_animals = {"dog", "cat", "hamster", "parrot"}  
>>> farm_animals = {"cow", "chicken", "goat", "dog", "cat"}
```

The union of `pet_animals` and `farm_animals` is a new set containing the elements from both initial sets. In the case of having one or more repeated elements, the resulting set will contain only one instance of those elements. In this example, "dog" and "cat" are repeated.

Here's how the union operator works:

#### Python

```
>>> pet_animals | farm_animals  
{'cow', 'hamster', 'cat', 'dog', 'goat', 'chicken', 'parrot'}  
  
>>> farm_animals | pet_animals  
{'cow', 'hamster', 'cat', 'dog', 'goat', 'chicken', 'parrot'}
```

Note that the resulting sets contain only one instance of "dog" and "cat". The rest of the elements are also in the result. It's important to mention that the union operation is

commutative, meaning the order of the operands doesn't affect the outcome.

The `.union()` method works similarly. You'll invoke the method on one of the sets and pass the other as an argument:

Python

```
>>> pet_animals.union(farm_animals)
{'hamster', 'chicken', 'cow', 'dog', 'parrot', 'cat', 'goat'}
```

```
>>> farm_animals.union(pet_animals)
{'hamster', 'chicken', 'cow', 'dog', 'parrot', 'cat', 'goat'}
```

The resulting sets contain the same elements as in the previous example. Of course, they're not in the same order because sets are unordered data types.

In both examples, the operator and method behave identically. However, there's a subtle difference between them. When you use the `|` operator, both operands must be sets. The `.union()` method, on the other hand, can take any iterables as arguments and then perform the union.

Observe the difference between these two statements:

Python

```
>>> pet_animals | ["cow", "chicken", "goat", "dog", "cat"]
Traceback (most recent call last):
...
TypeError: unsupported operand type(s) for |: 'set' and 'list'
```

```
>>> pet_animals.union(["cow", "chicken", "goat", "dog", "cat"])
{'hamster', 'chicken', 'cow', 'dog', 'parrot', 'cat', 'goat'}
```

The first attempt to compute the union of `pet_animals` and the list of farm animals fails because the `|` operator doesn't support data types different from sets. The `.union()`

method succeeds because it can handle different iterable types.

**Note:** The behavior of the union operator and method shown in the example above is consistent with the other operators and their corresponding methods.

Here's how you can use the operator and method with multiple sets:

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}
```

As usual, the resulting set contains all elements that are present in the specified sets without any duplication.



[Learn Python »](#)

Remove ads

## Intersection

The **intersection** of two sets is a new set containing only the elements common to both sets.

You can use the `&` operator or the `.intersection()` method to perform an intersection.

Intersection is a great way to find shared values between datasets, such as mutual friends, overlapping tags, or common items.

The following diagram provides a visual representation of the intersection between sets  $A$  and  $B$ :



### Set Intersection

The green area represents the intersection and holds the elements that are present in both  $A$  and  $B$ .

Here's the syntax for the intersection operation, using both the operator and the method:

#### Python Syntax

```
x1 & x2 [& x3 & ... & xN]  
  
x1.intersection(x2[, x3, ..., xN])
```

Again, the portion in square brackets is optional. The `&` operator is also binary, so you can use it with two operands. The `.intersection()` method can take any number of arguments. The operator and the method return a set of elements common to both.

Consider the following examples where you use an intersection to find mutual friends:

## Python

```
>>> john_friends = {"Linda", "Mathew", "Carlos", "Laura"}  
>>> jane_friends = {"Alice", "Bob", "Laura", "Mathew"}  
  
>>> john_friends & jane_friends  
{'Laura', 'Mathew'}  
  
>>> john_friends.intersection(jane_friends)  
{'Laura', 'Mathew'}
```

In this example, the resulting set contains the people who are both John and Jane's friends.

Isn't that cool?

You can specify multiple sets with the intersection operator and method, 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 & b & c & d  
{4}  
  
>>> a.intersection(b, c, d)  
{4}
```

The resulting set contains only elements that are present in all of the specified sets. In this example, only the number 4 is present in all sets.

Finally, it's important to mention that if an intersection finds no common elements, you get an empty set:

```
>>> x = {1, 2, 3, 4}  
>>> y = {5, 6, 7, 8}  
  
>>> x & y  
set()
```

In this example, there are no common elements in `x` and `y`, so you get an empty set as a result of the intersection operation.

## Difference

The **difference** between two sets is a new set containing elements that are in the first set but not in the second. It subtracts one set from another. You can perform a difference using the `-` operator or the `.difference()` method. Difference is applicable when you want to identify what's unique to one dataset compared to another.

The following diagram provides a visual representation of the difference between sets *A* and *B*:



Set Difference

The green area represents the elements that only exist in  $A$ , which is the result of the difference operation.

Here's the syntax for the difference operation, using both the operator and the method:

#### Python Syntax

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

Again, the operator works with two sets, and the method can take any number of arguments.

For a practical example, say that you have a set of people who have registered for an event and another set of people who actually attended. The difference tells you who didn't show up:

#### Python

```
>>> registered_users = {"Alice", "Bob", "Charlie", "Diana", "Linda"}  
>>> checked_in_users = {"Alice", "Charlie", "Linda"}  
  
>>> registered_users - checked_in_users  
{'Bob', 'Diana'}
```

In this example, the resulting set contains the users who registered but didn't attend the event. The set difference operation is useful when you have questions like who is missing or what hasn't been done yet.

**Note:** In the case of set difference, the order of the operands affects the result:

#### Python

```
>>> checked_in_users - registered_users
```

```
set()
```

Now, you get an empty set, which demonstrates that the difference operation isn't commutative.

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

When you specify multiple sets, the operation is performed from left to right. To better understand this behavior, consider the following diagram:



Here,  $a - b$  is computed first, resulting in  $\{1, 2, 3, 300\}$ . Then,  $c$  is subtracted from the resulting set, leaving  $\{1, 2, 3\}$ .



[Become a Python Expert »](#)

[i Remove ads](#)

## Symmetric Difference

The **symmetric difference** between two sets is a new set containing all the elements that appear in either set but not both. You can perform a symmetric difference using the `^` operator or the `.symmetric_difference()` method. The symmetric difference comes in handy when you need to identify elements that are in exactly one of the two sets.

The following diagram gives you a visual representation of the symmetric difference between sets  $A$  and  $B$ :



Set Symmetric Difference

The green area represents elements in  $A$  but not in  $B$ , and the other way around. The middle area represents the elements present in both  $A$  and  $B$ . So, the resulting set contains uncommon or unshared elements.

Here's the syntax for both the operator and method for the symmetric difference operation:

#### Python Syntax

```
x1 ^ x2 [^ x3 ^ ... ^ xN]  
x1.symmetric_difference(x2)
```

The operator works similarly to the other set operators. However, the method behaves differently because it only accepts one argument.

**Note:** The symmetric difference is also known as the [exclusive OR](#) because it represents elements that appear in either of the two sets, but not in both at the same time.

Going back to your event example, say that the event has morning and afternoon sessions, and you want to know the attendees who participated in only one of the sessions. You can do this with the symmetric difference operation:

#### Python

```
>>> morning_attendees = {"Alice", "Charlie", "Linda", "John", "Jane"}  
>>> afternoon_attendees = {"Charlie", "Linda", "Bob", "Jane"}  
  
>>> morning_attendees ^ afternoon_attendees  
{'John', 'Bob', 'Alice'}  
  
>>> morning_attendees.symmetric_difference(afternoon_attendees)  
{'John', 'Bob', 'Alice'}
```

The resulting set contains the attendees who were present in only one session. A closer look at the involved sets will allow you to conclude that Alice and John attended the morning session, while Bob only attended the afternoon session.

**Note:** Unlike the set difference operation, symmetric difference is commutative:

Python

```
>>> afternoon_attendees ^ morning_attendees
{'John', 'Bob', 'Alice'}
```

Despite the order of operands, you get the same result.

The symmetric difference operator (^) also allows you to process 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 you specify multiple sets, the operation is performed from left to right.

Curiously, the `.symmetric_difference()` method doesn't support multiple arguments:

Python

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

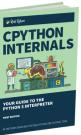
>>> a.symmetric_difference(b, c)
Traceback (most recent call last):
...
TypeError: symmetric_difference() takes exactly one argument (2 given)
```

If you try to call `.symmetric_difference()` with more than one argument, then you get a `TypeError` exception. You can work around this behavior by chaining calls:

Python

```
>>> a.symmetric_difference(b).symmetric_difference(c)
{10, 100, 5}
```

In this example, you call the method on `a`. Then, you call the method on the result of the previous call. The final result is the same as with the operator.



[Your Guided Tour Through the Python 3.9 Interpreter »](#)

Remove ads

## Using Augmented Set Operations

The union, intersection, difference, and symmetric difference operators covered in the previous section have [augmented variations](#) that you can use to modify a set [in place](#). Remember, sets are mutable data types, so you can add and remove elements from a set in place.

**Note:** Python has a variation of sets called `frozenset` that's immutable. This built-in data type works like normal sets but doesn't allow you to add or remove elements. Additionally, they don't support augmented operations.

For each augmented operator, you'll have an equivalent method. In the following sections, you'll learn about these augmented operators and how they work with sets.

### Augmented Union

The **augmented union** updates a set in place. You can use either the augmented union operator (`|=`) or the `.update()` method to do this. Here's the syntax for both:

#### Python Syntax

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

The part of the syntax that's enclosed in square brackets is optional. Both tools update the target set (`x1`) with elements from the other sets.

Returning to your event example, say that you'd like to have an up-to-date record of the attendees who have checked in:

#### Python

```
>>> checked_in_attendees = set()  
>>> id(checked_in_attendees)  
4364315648  
  
>>> checked_in_attendees |= {"Alice", "Charlie"}  
>>> checked_in_attendees  
{'Charlie', 'Alice'}  
>>> id(checked_in_attendees)  
4364315648  
  
>>> checked_in_attendees.update({"Linda", "Bob"})  
>>> checked_in_attendees  
{'Charlie', 'Bob', 'Linda', 'Alice'}  
>>> id(checked_in_attendees)  
4364315648
```

In this example, you first create an empty set using the `set()` constructor. Then, you use the `|=` operator to add two people to the checked-in set. Next, you add more people using the

.update() method. The built-in [id\(\)](#) function lets you check that your set remains the same and that the updates are in-place operations.

## Augmented Intersection

The **intersection update** allows you to modify a set in place by intersection. You can perform this operation using the augmented intersection operator (&=) or the [.intersection\\_update\(\)](#) method.

Here's the required syntax:

### Python Syntax

```
x1 &= x2[ & x3 & ... & xN]  
  
x1.intersection_update(x2[, x3, ..., xN])
```

The operator and the method update the target set (x1) with elements from the other sets. As a result, the final version of x1 will hold only the elements found in all the involved sets.

For example, say that you have a list of target customers for a marketing campaign, and you only want to keep those who have agreed to receive emails:

### Python

```
>>> accepted_emails = {"Bob", "Diana", "Charlie"}  
  
>>> target_customers = {"Alice", "Bob", "Charlie", "Diana", "Jane"}  
>>> target_customers &= accepted_emails  
>>> target_customers  
{'Charlie', 'Bob', 'Diana'}  
  
>>> # Or  
>>> target_customers = {"Alice", "Bob", "Charlie", "Diana", "Jane"}  
>>> target_customers.intersection_update(accepted_emails)
```

```
>>> target_customers
{'Charlie', 'Bob', 'Diana'}
```

With the intersection update, you've reduced the target customers to those who accepted marketing emails.

## Augmented Difference

The **difference update** lets you modify a set in place by difference. To run this type of update, you can use the augmented difference operator (`-=`) or the `.difference_update()` method.

Here's the syntax you should use to run a difference update:

### Python Syntax

```
x1 -= x2[ | x3 | ... | xN]  
x1.difference_update(x2[, x3, ..., xN])
```

Both the operator and the method update `x1` by removing elements found in the rest of the sets.

**Note:** The syntax for the augmented difference operator differs from the syntax of augmented union and augmented intersection. Note that the minus sign (`-`) isn't used in the optional part of the syntax. Instead, you have to use the union operator (`|`).

To learn how the augmented difference works, suppose you have a list of tasks to complete for a Python project and want to remove the ones you've just finished:

### Python

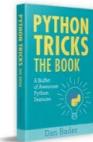
```
>>> todo_list = {
...     "Implement user login",
```

```
...     "Fix bug #123",
...     "Improve performance",
...     "Write unit tests"
... }

>>> completed_tasks = {
...     "Fix bug #123",
...     "Improve performance"
... }

>>> todo_list.difference_update(completed_tasks)
>>> todo_list
{'Implement user login', 'Write unit tests'}
```

In this example, you use the `.difference_update()` method to remove the completed tasks. Go ahead and try the `-=` operator!



**"I don't even feel like I've scratched the surface of what I can do with Python"**

[Write More Pythonic Code »](#)

[Remove ads](#)

## Augmented Symmetric Difference

The **symmetric difference update** allows you to modify a set by symmetric difference. You can perform this operation with the augmented symmetric difference operator (`^=`) or the `.symmetric_difference_update()` method.

Here's the required syntax:

### Python Syntax

```
x1 ^= x2

x1.symmetric_difference_update(x2)
```

These tools update `x1` in place, retaining the elements found in either `x1` or `x2`, but not in both.

Suppose you want to get the list of people who participated in only one session of your event. You can do it with a symmetric difference update:

Python

```
>>> morning_attendees = {"Alice", "Charlie", "Linda", "John", "Jane"}  
>>> afternoon_attendees = {"Charlie", "Linda", "Bob", "Jane"}  
  
>>> whole_day_attendees = set()  
>>> whole_day_attendees ^= morning_attendees  
>>> whole_day_attendees.symmetric_difference_update(afternoon_attendees)  
>>> whole_day_attendees  
{'Bob', 'Alice', 'John'}
```

In this example, you have two initial sets with the list of people who came in the morning and afternoon sessions. Next, you create a new empty set called `whole_day_attendees`. Then, you use the `^=` operator and the `.symmetric_difference_update()` to update the set with the elements from the initial sets. As a final result, you get the list of people that came to only one session.

## Comparing Sets

You can also use operators and methods to compare two sets. In the context of sets, relational operators like `>`, `<`, `>=`, and `<=` have a slightly different interpretation than in numerical comparisons. In the following sections, you'll learn how these operators work with sets and how to interpret their results. You'll also learn about the equivalent methods.

It's important to note that all the checks in the following sections return a `Boolean` value rather than a new set.

# Subsets

The **subset** operation lets you determine whether one set is contained in another. You can use the `<=` operator or the `.issubset()` method to perform this check.

Here's the syntax for both the operator and method for the subset check:

## Python Syntax

```
x1 <= x2  
x1.issubset(x2)
```

A set  $x_1$  is a subset of another set  $x_2$  if every element of  $x_1$  is in  $x_2$ . This type of check is commonly used to check whether the first set is contained in the second.

For example, suppose you want to prepare a delicious meal. You have the required ingredients for a recipe and the ingredients you currently have in your kitchen. You can quickly check if you have everything you need for your meal with a subset operation:

## Python

```
>>> required_ingredients = {"cheese", "eggs", "milk"}  
>>> available_ingredients = {"cheese", "eggs", "milk", "sugar", "salt"}  
  
>>> required_ingredients <= available_ingredients  
True  
  
>>> required_ingredients.issubset(available_ingredients)  
True
```

When you run this subset check, you get `True` as a result. This means that you have all the ingredients for your meal.

It's important to note that a set is considered a subset of itself:

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

>>> a <= a
True

>>> a.issubset(a)
True
```

This result may seem a bit strange at first, but it completely fits the definition: every element of `a` is in `a`.



**"I wished I had access to a book like this when I started learning Python many years ago"**  
— Mariatta Wijaya, CPython Core Developer

[Learn More »](#)

Remove ads

## Proper Subsets

With this check, you can determine whether one set is a proper subset of another. To perform this check, you'll use the `<` operator. You won't have a dedicated method.

The syntax for a proper subset check is shown below:

### Python Syntax

```
x1 < x2
```

A **proper subset** is a subset with the additional requirement that the compared sets can't be identical. For example, set `x1` is considered a proper subset of set `x2` if every element of `x1` is in `x2`, and `x1` isn't equal to `x2`.

**Note:** Two sets are equal when they contain the same elements. You can run this comparison with the equality operator (==), which doesn't consider the order of elements because sets are unordered data types.

To illustrate how you could use the proper subset check, say that you're offering a service with two plans: regular and premium. Using a proper subset check, you can confirm that the regular plan has fewer features than the premium plan:

Python

```
>>> regular_plan = {"Tutorials", "Quizzes"}  
>>> premium_plan = {"Tutorials", "Video Courses", "Quizzes", "Books"}  
  
>>> regular_plan < premium_plan  
True
```

In this example, when you check whether your regular plan is a proper subset of your premium plan, you get True. This check will warn you against any mistakes while updating the content of your plans.

Finally, while a set is considered a subset of itself, it isn't a proper subset of itself:

Python

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

This behavior is consistent with the proper subset definition, which states that both sets can't be equal.

## Supersets

The **superset** check lets you determine whether one set contains another. To perform this check, you can use the `>=` operator or the `.issuperset()` method.

Here's the syntax for both the operator and method for the superset check:

#### Python Syntax

```
x1 >= x2  
x1.issuperset(x2)
```

A superset is the opposite of a subset. In other words, set `x1` is a superset of set `x2` if `x1` contains every element of `x2`.

Returning to the meal example, you can also use the superset check to determine whether you have all the required ingredients:

#### Python

```
>>> required_ingredients = {"cheese", "eggs", "milk"}  
>>> available_ingredients = {"cheese", "eggs", "milk", "sugar", "salt"}  
  
>>> available_ingredients >= required_ingredients  
True  
  
>>> available_ingredients.issuperset(required_ingredients)  
True
```

This time, you run the check in the opposite direction. Instead of checking that all the ingredients are available in your kitchen, you're checking whether your available ingredients contain the required ones.

Finally, a set is also considered a superset of itself:

#### Python



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

Again, this behavior completely fits the definition: a contains every element of a.



## Your Practical Introduction to Python 3 »

 Remove ads

## Proper Supersets

With this check, you can determine whether one set is a proper superset of another. To perform this check, you'll use the `>` operator. As with a proper subset, you won't have a dedicated method to check for a proper superset.

Here's the syntax for a proper superset check:

### Python Syntax

```
x1 > x2
```

A **proper superset** is also a superset, except that the involved sets can't be identical. For example, set `x1` is a proper superset of set `x2` if `x1` contains every element of `x2`, and `x1` is different from `x2`.

The proper superset check is the opposite of the proper subset check. So, here's a different approach to the example about service plans:

### Python



```
>>> regular_plan = {"Tutorials", "Quizzes"}  
>>> premium_plan = {"Tutorials", "Video Courses", "Quizzes", "Books"}  
  
>>> premium_plan > regular_plan  
True
```

This time, you check in the opposite direction, checking whether the regular plan is contained in the premium plan but guaranteeing that both plans are different.

Finally, a set isn't a proper superset of itself:

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

Once again, this behavior is consistent with the definition of a proper superset, where the compared sets can't be equal.

## Disjoint Sets

Finally, you have the disjoint check, which allows you to determine if two sets don't have any elements in common. In this case, you only have a dedicated method called `.isdisjoint()`. You won't have an operator.

Here's the syntax for the disjoint check:

```
Python Syntax  
  
x1.isdisjoint(x2)
```

The `x1.isdisjoint(x2)` call returns `True` if `x1` and `x2` have no elements in common. This type of check is useful when you need to ensure that two sets have no overlap.

For example, say that you're coding the order system for a store. You'd like to prevent the purchase of certain products if the user is underage:

Python

```
>>> def verify_purchase(age, selection, restricted_products):
...     if age < 21 and not selection.isdisjoint(restricted_products):
...         print("Purchase denied: selection includes age-restricted products")
...     else:
...         print("Purchase approved")
...
```

Inside `verify_purchase()`, you first check whether the current user is under 21 years old. If that's the case, the user is underage, so you need to verify the selected products. To do that, you use a set disjoint check.

Here's how the function works:

Python

```
>>> verify_purchase(
...     age=18,
...     selection={"milk", "bread", "beer"},
...     restricted_products={"alcohol", "beer", "cigarettes"})
Purchase denied: selection includes age-restricted products

>>> verify_purchase(
...     age=18,
...     selection={"milk", "bread"},
...     restricted_products={"alcohol", "beer", "cigarettes"})
Purchase approved
```

```
>>> verify_purchase(  
...     age=35,  
...     selection={"milk", "bread", "beer"},  
...     restricted_products={"alcohol", "beer", "cigarettes"}  
... )  
Purchase approved
```

In the first call, the user is 18 years old and is trying to buy beer, which isn't allowed for underage users. In the second call, the beer was removed from the order, so the purchase was approved. Finally, if the user is an adult, then they don't have purchase restrictions.

**Find Your Dream Python Job**

[pythonjobshq.com](http://pythonjobshq.com)



[i Remove ads](#)

## Using Other Set Methods

So far, you've learned a lot about set-specific operations. However, those aren't the only operations that you can perform on sets. There are a mix of methods that you can call on sets to change their content.

You'll find methods for adding elements to a set and also for removing elements from a set. These are the two generic mutations that you can perform on sets. In the following sections, you'll learn about these methods and how to use them in your Python code.

### Adding an Element With `.add()`

The `.add()` method allows you to add a single element to an existing set. Below is a quick example that shows how this method works:

Python



```
>>> employees = {"Alice", "Charlie"}  
  
>>> employees.add("John")  
>>> employees  
{'Charlie', 'John', 'Alice'}  
  
>>> employees.add("Laura")  
>>> employees  
{'Charlie', 'John', 'Laura', 'Alice'}  
  
>>> employees.add("John")  
>>> employees  
{'Laura', 'Alice', 'Charlie', 'John'}  
  
>>> employees.add("Jane", "Bob")  
Traceback (most recent call last):  
...  
TypeError: set.add() takes exactly one argument (2 given)
```

In these examples, you add elements to an existing set. Note that when you call `.add()` with an element that's already in the set, nothing happens. Finally, you can only pass one argument to `.add()`. Otherwise, you get a `TypeError` exception.

## Removing an Existing Element With `.remove()`

If you need to remove elements from a set, then use the `.remove()` method. You can call this method with the element you want to remove. However, keep in mind that you'll get an exception if the target element isn't found in the set:

Python

```
>>> employees = {"Alice", "Charlie", "John", "Laura"}  
  
>>> employees.remove("Charlie")  
>>> employees  
{'John', 'Laura', 'Alice'}
```

```
>>> employees.remove("Linda")
Traceback (most recent call last):
...
KeyError: 'Linda'
```

With `.remove()`, you can delete one element at a time. When the target element doesn't exist in the set, you get a `KeyError` exception. Why a `KeyError`? Well, sets are implemented like dictionaries with keys but without associated values. So, set elements are like dictionary keys.

## Deleting an Existing or Missing Element With `.discard()`

Like `.remove()`, the `.discard()` method allows you to remove a single element from an existing set. The difference between both methods is that `.discard()` doesn't raise an [exception](#) if the element doesn't exist:

```
Python >
>>> employees = {"Alice", "Charlie", "John", "Laura"}

>>> employees.discard("Alice")
>>> employees
{'Charlie', 'John', 'Laura'}

>>> employees.discard("Linda")
>>> employees
{'Charlie', 'John', 'Laura'}
```

If the element exists in the set, then `.discard()` removes it. If the element isn't present in the set, then `.discard()` does nothing without raising an error.

## Removing and Returning an Element With `.pop()`

Sometimes, you need to retrieve an element from a set, process it, and then remove it. In that case, you can use the `.pop()` method, which removes and returns an element from a set. If the target set is empty, then `.pop()` raises a `KeyError` exception:

Python

```
>>> employees = {"Alice", "Charlie", "John", "Laura"}  
  
>>> employee = employees.pop()  
>>> employee  
'Charlie'  
>>> employee = employees.pop()  
>>> employee  
'John'  
>>> employees  
{'Laura', 'Alice'}  
  
>>> employees.pop()  
>>> employees.pop()  
>>> employees.pop()  
Traceback (most recent call last):  
...  
KeyError: 'pop from an empty set'
```

The `.pop()` method removes and returns an arbitrary element from a set in one go. Because sets are unordered data types, you can't predict which element will be removed. You can assign the removed element to a `variable` and get a reference as you did with `employee`.

## Removing All Elements With `.clear()`

If you ever need to remove all the elements from a set in a single operation, then you can use the `.clear()` method:

Python

```
>>> employees = {"Alice", "Charlie", "John", "Laura"}  
  
>>> employees.clear()  
>>> employees  
set()
```

The `.clear()` method comes in handy when you want to reset a set so you can start fresh.

Calling `.clear()` completely empties the set in one step, getting it ready for new data without the hassle of removing items individually.

## Creating Shallow Copies of Sets With `.copy()`

You can use the `.copy()` method to create a [shallow copy](#) of an existing set. This method is useful when you need to keep a copy of a set while performing some changes in the original data:

Python

```
>>> employees = {"Alice", "Charlie", "John", "Laura"}  
>>> employees_copy = employees.copy()  
  
>>> employees == employees_copy  
True  
>>> employees is employees_copy  
False  
  
>>> # Change the original set  
>>> employees.remove("Alice")  
>>> employees  
{'Charlie', 'John', 'Laura'}  
  
>>> employees_copy # Keeps the data unchanged  
{'Charlie', 'Alice', 'John', 'Laura'}
```

In this example, you create a copy of employees by calling the `.copy()` method. The copy will have the same data as the original set. In fact, it'll hold references to the data in the original set. That's why it's a shallow copy.

Once you've saved the data in the copy, you can perform any modifications to the original set. The copy will keep the data unchanged.

## Traversing Sets

When working with Python sets, you'll often need to traverse their elements in order to perform actions on them. A Python `for loop` lets you do that. If you want to iterate over a set while processing and removing its elements, then you can use a `while loop`.

However, you need to keep in mind that in both situations, the iteration order will be unknown beforehand because sets are unordered data types. If you need to examine a set's content in order, then you can loop through it using the built-in `sorted()` function.

In the following sections, you'll explore some short examples that will help you understand how to use these techniques for traversing sets in Python.

## Accessing and Modifying Elements in a Loop

Python sets are mutable in the sense that you can add or remove elements from them. However, you can't modify a set element in place as you can with list items, for example. So, when you use a loop to traverse a set, you can perform actions *with* each element of the set, but you can't modify them.

Consider the following example:

```
Python >
>>> employees = {"Alice", "Charlie", "John", "Laura"}
>>> for employee in employees:
```

```
...     print(f"Hello, {employee}!")
...
Hello, Charlie!
Hello, John!
Hello, Laura!
Hello, Alice!
```

In this example, you use a for loop to iterate over the set of employees. Inside the loop, you access the current element and use it to generate a message that you print to the screen. This loop doesn't modify the set's elements but does something with them.

If you want to modify the elements of a set, then you should create a new set. For example, say that you want the employees' names to be displayed in uppercase letters. You can do this with the following code:

```
Python >_>

>>> employees = {"Alice", "Charlie", "John", "Laura"}
>>> uppercase = set()

>>> for employee in employees:
...     uppercase.add(employee.upper())
...
>>> uppercase
{'ALICE', 'CHARLIE', 'JOHN', 'LAURA'}
```

In this example, you have the original set containing employees' names. Next, you create an empty set that you'll use to store the names in uppercase. In the loop, you use the .add() method to populate the uppercase set by applying the .upper() method to each employee name. As a result, you get a set of names in uppercase.

You can achieve the same result using a set comprehension like the following:

```
Python >_>
```

```
>>> employees = {"Alice", "Charlie", "John", "Laura"}
```

```
>>> {employee.upper() for employee in employees}
```

```
{'ALICE', 'CHARLIE', 'JOHN', 'LAURA'}
```

Set comprehensions provide a concise and efficient way to create sets from iterables, including other sets. In this example, you create a new set with the employees' names in uppercase letters.

## Processing and Removing Elements in a Loop

The `.pop()` method comes in handy when you need to process throwaway elements before removing them. For example, say that you have a set with the available seats on a flight. Once you assign a seat, you can remove it from the set:

Python

```
>>> available_seats = {"A1", "A2", "B1", "B2", "C1"}  
  
>>> while available_seats:  
...     assigned_seat = available_seats.pop()  
...     print(f"Seat {assigned_seat} was assigned")  
...  
Seat B1 was assigned  
Seat A1 was assigned  
Seat B2 was assigned  
Seat C1 was assigned  
Seat A2 was assigned  
  
>>> available_seats  
set()
```

In this example, you use a `while` loop to process the set of available seats. The `.pop()` method lets you get a reference to the current seat in the loop. Then, you assign the seat to a

variable immediately after removing it from the set of available seats.

## Iterating Through a Sorted Set

As you already know, sets don't keep their elements in order. Therefore, you can't predict which element will be processed in each iteration. The built-in `sorted()` function can help you put some order to your iteration:

Python

```
>>> cities = {"Vancouver", "Berlin", "London", "Warsaw", "Vienna"}  
  
>>> for city in sorted(cities):  
...     print(city)  
  
Berlin  
London  
Vancouver  
Vienna  
Warsaw
```

In this example, you use the `sorted()` function to iterate over a set of cities in alphabetical order.

**Note:** You can't sort a set in place because they're unordered data types. With the `sorted()` function, you get a new list containing the set's elements in sorted order. This list takes additional memory.

To add some spice to this example, suppose the set contains tuples of the form `(city, population)`, and you want to iterate it sorted by population. In this situation, you can do something like the following:

Python

```
>>> cities = {
...     ("Vancouver", 675000),
...     ("Berlin", 3800000),
...     ("London", 8980000),
...     ("Warsaw", 1790000),
...     ("Vienna", 1900000),
... }

>>> for city in sorted(cities, key=lambda city: city[1]):
...     print(city)
...
('Vancouver', 675000)
('Warsaw', 1790000)
('Vienna', 1900000)
('Berlin', 3800000)
('London', 8980000)
```

In this example, you use a `lambda` function that returns the population of the current city. The `sorted()` function uses this value as a sorting key.

By default, the `sorted()` function sorts the items in ascending order. You can use the `reverse` argument to sort in descending order:

```
Python >_
>>> for city in sorted(cities, key=lambda city: city[1], reverse=True):
...     print(city)
...
('London', 8980000)
('Berlin', 3800000)
('Vienna', 1900000)
('Warsaw', 1790000)
('Vancouver', 675000)
```

Setting `reverse` to `True` makes the function sort the items in descending order. Now, the most populated city is at the top of the output.

# Exploring Other Set Capabilities

Python sets work seamlessly with some built-in functions and operators. Among the most popular are the `len()` function and the membership operators `in` and `not in`, which provide efficient ways to work with sets.

**Note:** To learn more about the `len()` function and the membership operators, check out the following tutorials:

- [Using the `len\(\)` Function in Python](#)
- [Python's `in` and `not in` Operators: Check for Membership](#)

Whether you want to quickly check how many elements a set has or verify if an element is in a set, these two tools have you covered.

## Finding the Number of Elements With `len()`

To find out how many elements a set contains, simply call the `len()` function with your set as an argument:

Python

```
>>> fruits = {"apple", "mango", "orange"}  
>>> len(fruits)  
3
```

Because sets don't allow duplicates, you'll always get the count of distinct items, making `len()` a quick way to verify the uniqueness of your data.

## Running Membership Tests on Sets

You can use the Python `in` and `not in` operators to check whether an element belongs to a collection or container. These checks are known as **membership tests** and can be pretty useful in real-world coding.

Because Python sets are implemented as hash tables, it turns out that they're very efficient for membership tests, especially compared to `list` objects. So, when you need to perform frequent membership tests on a collection, you'll benefit from using a set.

The following [script](#) compares the execution speed of membership tests run on a list and a set:

```
Python                                         membership.py

import random
import time

numbers_list = list(range(1_000_000))
numbers_set = set(range(1_000_000))
number_to_check = [random.randint(0, 999_999) for _ in range(1_000)]

start = time.perf_counter()
for number in number_to_check:
    _ = number in numbers_list
end = time.perf_counter()
print(f"List membership check took {end - start:.4f} seconds")

start = time.perf_counter()
for number in number_to_check:
    _ = number in numbers_set
end = time.perf_counter()
print(f"Set membership check took {end - start:.4f} seconds")
```

Sets shine when you need to run several membership checks. In this example, you generate a list and a set of one million numbers each. Then, you perform 1000 membership tests on each data type in a loop. Go ahead and run the script on your computer to check the speed gain with the set variation.