

Dictionaries in Python

by Leodanis Pozo Ramos  48m  33 Comments

 [basics](#) [python](#)

Mark as Completed



Share 

Table of Contents

- [Getting Started With Python Dictionaries](#)
- [Creating Dictionaries in Python](#)
 - [Dictionary Literals](#)
 - [The dict\(\) Constructor](#)

— FREE Email Series —

 **Python Tricks** 

```
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 the `.fromkeys()` Class Method
- Accessing Dictionary Values
- Populating Dictionaries Incrementally
 - Assigning Keys Manually
 - Adding Keys in a for Loop
 - Building Dictionaries With Comprehensions
- Exploring the `dict` Class Methods
 - Retrieving Data From Dictionaries
 - Adding Key-Value Pairs and Updating Dictionaries
 - Removing Data From Dictionaries
- Using Operators With Dictionaries
 - Membership: `in` and `not in`
 - Equality and Inequality: `==` and `!=`
 - Union and Augmented Union: `|` and `|=`
- Use Built-in Functions With Dictionaries
 - Checking for Truthy Data in Dictionaries: `all()` and `any()`
 - Determining the Number of Dictionary Items: `len()`
 - Finding Minimum and Maximum Values: `min()` and `max()`
 - Sorting Dictionaries by Keys, Values, and Items: `sorted()`
 - Summing Dictionary Values: `sum()`
- Iterating Over Dictionaries
 - Traversing Dictionaries by Keys
 - Iterating Over Dictionary Values
 - Looping Through Dictionary Items
- Exploring Existing Dictionary-Like Classes
- Creating Custom Dictionary-Like Classes
- Conclusion
- Frequently Asked Questions

projects python stdlib testing tools
web-dev web-scraping

```
... app.py
> 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 Dictionaries
- Creating Dictionaries in Python
- Accessing Dictionary Values
- Populating Dictionaries Incrementally
- Exploring the `dict` Class Methods
- Using Operators With Dictionaries
- Use Built-in Functions With Dictionaries
- Iterating Over Dictionaries
- Exploring Existing Dictionary-Like Classes
- Creating Custom Dictionary-Like Classes

[Remove ads](#)

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: [Using Dictionaries in Python](#)

Python dictionaries are a powerful built-in data type that allows you to store key-value pairs for efficient data retrieval and manipulation. Learning about them is essential for developers who want to process data efficiently. In this tutorial, you'll explore how to create dictionaries using literals and the `dict()` constructor, as well as how to use Python's operators and built-in functions to manipulate them.

By learning about Python dictionaries, you'll be able to access values through key lookups and modify dictionary content using various methods. This knowledge will help you in data processing, configuration management, and dealing with JSON and CSV data.

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

- A dictionary in Python is a **mutable collection of key-value pairs** that allows for efficient data retrieval using unique keys.
- Both `dict()` and `{}` can create dictionaries in Python. Use `{}` for **concise syntax** and `dict()` for **dynamic creation** from iterable objects.
- `dict()` is a **class** used to create dictionaries. However, it's **commonly called a built-in function** in Python.
- `.__dict__` is a **special attribute** in Python that holds an object's **writable attributes** in a dictionary.
- Python `dict` is implemented as a **hashmap**, which allows for **fast key lookups**.

[Mark as Completed](#) [Share](#)

Recommended Video Course

[Using Dictionaries in Python](#)



Join Real Python and Unlock Learning Paths, Courses, Live Q&As, and More:
[Become a Python Expert »](#)

To get the most out of this tutorial, you should be familiar with basic Python syntax and concepts such as [variables](#), [loops](#), and [built-in functions](#). Some experience with [basic Python data types](#) will also be helpful.

Get Your Code: Click here to download the free sample code that you'll use to learn about dictionaries in Python.

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

Interactive Quiz

Dictionaries in Python



In this quiz, you'll test your understanding of Python's dict data type. By working through this quiz, you'll revisit how to create and manipulate dictionaries, how to use Python's operators and built-in functions with them, and how they're implemented for efficient data retrieval.

Getting Started With Python Dictionaries

Dictionaries are one of Python's most important and useful built-in data types. They provide a mutable collection of key-value pairs that lets you efficiently access and mutate values through their corresponding keys:

Python

```
>>> config = {  
...     "color": "green",  
...     "width": 42,  
...     "height": 100,  
...     "font": "Courier",  
... }
```

```
>>> # Access a value through its key
>>> config["color"]
'green'

>>> # Update a value
>>> config["font"] = "Helvetica"
>>> config
{
    'color': 'green',
    'width': 42,
    'height': 100,
    'font': 'Helvetica'
}
```

A Python dictionary consists of a collection of key-value pairs, where each key corresponds to its associated value. In this example, "color" is a key, and "green" is the associated value.

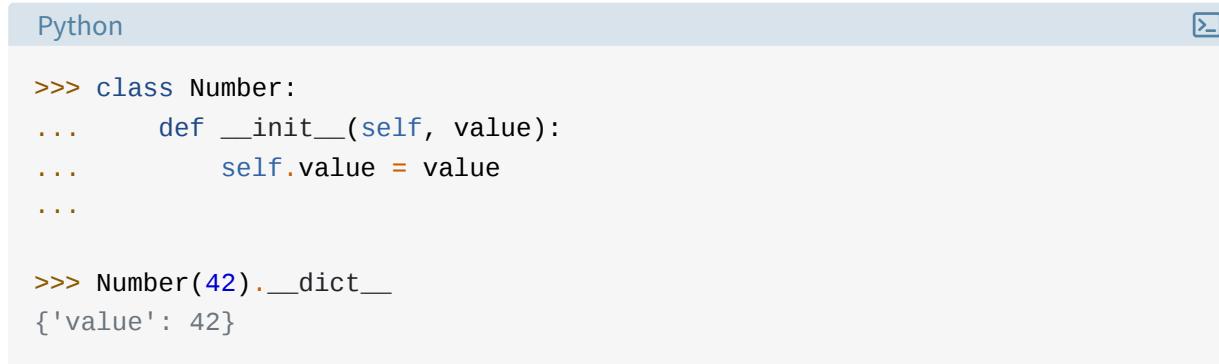
Dictionaries are a fundamental part of Python. You'll find them behind core concepts like scopes and namespaces as seen with the [built-in functions](#) `globals()` and `locals()`:

Python

```
>>> globals()
{
    '__name__': '__main__',
    '__doc__': None,
    '__package__': None,
    ...
}
```

The `globals()` function returns a dictionary containing key-value pairs that map names to objects that live in your current global scope.

Python also uses dictionaries to support the internal implementation of [classes](#). Consider the following demo class:



```
Python

>>> class Number:
...     def __init__(self, value):
...         self.value = value
...
...
>>> Number(42).__dict__
{'value': 42}
```

The `.__dict__` special attribute is a dictionary that maps attribute names to their corresponding values in Python classes and objects. This implementation makes attribute and method lookup fast and efficient in [object-oriented](#) code.

You can use dictionaries to approach many programming tasks in your Python code. They come in handy when processing [CSV](#) and [JSON](#) files, working with databases, loading configuration files, and more.

Python's dictionaries have the following characteristics:

- **Mutable:** The dictionary values can be updated [in place](#).
- **Dynamic:** Dictionaries can grow and shrink as needed.
- **Efficient:** They're implemented as [hash tables](#), which allows for fast key lookup.
- **Ordered:** Starting with [Python 3.7](#), dictionaries keep their items in the same [order](#) they were inserted.

The keys of a dictionary have a couple of restrictions. They need to be:

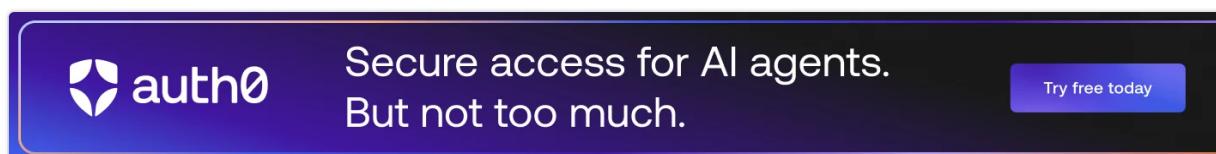
- **Hashable:** This means that you can't use unhashable objects like lists as dictionary keys.
- **Unique:** This means that your dictionaries won't have duplicate keys.

In contrast, the values in a dictionary aren't restricted. They can be of any Python type, including other dictionaries, which makes it possible to have nested dictionaries.

It's important to note that dictionaries are collections of pairs. So, you can't insert a key without its corresponding value or vice versa. Since they come as a pair, you always have to insert a key with its corresponding value.

Note: In some situations, you may want to add keys to a dictionary without deciding what the associated value should be. In those cases, you can use the `.setdefault()` method to create keys with a default or placeholder value.

In practice, you can use a dictionary when you need an efficient and mutable data structure that maps keys to values. In the following sections, you'll learn how to create and use dictionaries in your Python code.

An advertisement for Auth0. It features the Auth0 logo (a white 'a' inside a blue hexagon) and the text "Secure access for AI agents. But not too much." in white. A blue button on the right says "Try free today".

auth0 Secure access for AI agents. But not too much. Try free today

[i Remove ads](#)

Creating Dictionaries in Python

You can create Python dictionaries in a couple of ways, depending on your needs. The most common way is to use dictionary **literals**, which are a comma-separated series of key-value pairs in curly braces. The second way is to use the `dict()` **constructor**, which lets you create dictionaries from **iterables** of key-value pairs, other mappings, or a series of keyword arguments. It also lets you create empty dictionaries when you call it without arguments.

In the following sections, you'll dive deeper into how to create Python dictionaries using literals and the `dict()` constructor.

Dictionary Literals

You can define a dictionary by enclosing a comma-separated series of key-value pairs in curly braces ({}). To separate the keys from their values, you need to use a colon (:). Here's the syntax for a dictionary literal:

Python Syntax

```
{  
    <key_1>: <value_1>,  
    <key_2>: <value_2>,  
    ...  
    <key_N>: <value_N>,  
}
```

The keys and values are completely optional, which means that you can use an empty pair of curly braces to create an empty dictionary. Then, you have the keys, a colon, and the value associated with the current key. To separate the pairs, you use a comma.

The keys must be [hashable](#) objects like numbers, strings, or tuples. Being hashable means they can be passed to a hash function. A [hash function](#) takes data of arbitrary size and maps it to a fixed-size value called a **hash value**—or just **hash**—which is used for table lookup and comparison. In Python, the built-in immutable data types are hashable, and the mutable types are unhashable.

Note: Python [sets](#) also use curly braces to define their literals, but they enclose individual elements rather than key-value pairs. To create an empty set, you need to use `set()` instead of an empty pair of curly braces because this syntax is reserved for empty dictionaries.

The following code defines a dictionary that maps cities or states to the names of their corresponding Major League Baseball (MLB) teams:

Python



```
>>> MLB_teams = {
...     "Colorado": "Rockies",
...     "Chicago": "White Sox",
...     "Boston": "Red Sox",
...     "Minnesota": "Twins",
...     "Milwaukee": "Brewers",
...     "Seattle": "Mariners",
... }
```

You can only use hashable Python objects as dictionary keys. The following example shows a dictionary with integer, float, and Boolean objects used as keys:

```
Python >

>>> {42: "aaa", 2.78: "bbb", True: "ccc"}
{42: 'aaa', 2.78: 'bbb', True: 'ccc'}
```

You can even use objects like data types and functions as keys:

```
Python >

>>> types = {int: 1, float: 2, bool: 3}
>>> types
{<class 'int'>: 1, <class 'float'>: 2, <class 'bool'>: 3}

>>> types[float]
2
>>> types[bool]
3
```

However, you can't use unhashable objects as keys. If you try to, then you'll get an error:

```
Python >

>>> {[1, 2]: "A list as a key? Hmm..."}
Traceback (most recent call last):
```

```
...
TypeError: unhashable type: 'list'
```

Python lists are unhashable because any changes to their content would change their hash value, violating the requirement that hash values must remain constant for hashable types. In practice, you can't use any [mutable](#) data type as a key in a dictionary. This means that lists, sets, and dictionaries themselves aren't allowed.

If you need to use [sequences](#) as dictionary keys, then you can use tuples because tuples are immutable:

```
Python
>>> a_dict = {(1, 1): "a", (1, 2): "b", (2, 1): "c", (2, 2): "d"}

>>> a_dict[(1, 1)]
'a'
>>> a_dict[(2, 1)]
'c'
```

It's important to note that even though tuples are immutable, they can contain mutable objects. You can't use a tuple that contains mutable objects as a dictionary key:

```
Python
>>> {(1, [1, 1]): "a"}
Traceback (most recent call last):
...
TypeError: unhashable type: 'list'
```

In this example, the tuple that you try to use as a dictionary key contains a list. As a result, the tuple isn't hashable anymore, so you get an error.

Duplicate keys aren't allowed in Python's `dict` data type. Because of this restriction, when you assign a value to an existing key, you won't add a second instance of the key. Instead,

you'll replace the previously associated value with a new one.

For example, say that a given city has a second MLB team. You may try to add the second team by assigning it to the same key:

Python

```
>>> MLB_teams["Chicago"] = "Cubs"
>>> MLB_teams
{
    'Colorado': 'Rockies',
    'Chicago': 'Cubs',
    'Boston': 'Red Sox',
    'Minnesota': 'Twins',
    'Milwaukee': 'Brewers',
    'Seattle': 'Mariners',
}
```

In this example, you try to add a new key-value pair for the second MLB team in Chicago.

However, what happens is that you replace the old team name ("White Sox") with the new one ("Cubs").

Similarly, if you specify a key a second time during the creation of a dictionary, the second occurrence will override the first:

Python

```
>>> MLB_teams = {
...     "Colorado": "Rockies",
...     "Chicago": "White Sox",
...     "Chicago": "Cubs",
...     "Boston": "Red Sox",
...     "Minnesota": "Twins",
...     "Milwaukee": "Brewers",
...     "Seattle": "Mariners",
... }
```

```
>>> MLB_teams
{
    'Colorado': 'Rockies',
    'Chicago': 'Cubs', # "White Sox" is not present
    'Boston': 'Red Sox',
    'Minnesota': 'Twins',
    'Milwaukee': 'Brewers',
    'Seattle': 'Mariners'
}
```

In this example, your dictionary ends up containing the "Chicago": "Cubs" pair because you inserted it after "Chicago": "white Sox" with the same key.

Unlike dictionary keys, there are no restrictions for dictionary values. Literally none at all. A dictionary value can be any type of object, including mutable types like lists and dictionaries, as well as user-defined objects:

```
Python
>>> class Point:
...     def __init__(self, x, y):
...         self.x = x
...         self.y = y
...
>>> {
...     "colors": ["red", "green", "blue"],
...     "plugins": {"py_code", "dev_sugar", "fasting_py"},
...     "timeout": 3,
...     "position": Point(42, 21),
... }
```

In this example, you create a dictionary with a list, a set, an integer, and a custom object as values. All these objects work because values have no restrictions.

There's also no restriction against a particular value appearing in a dictionary multiple times:

```
Python >
>>> {0: "a", 1: "a", 2: "a", 3: "a"}
{0: 'a', 1: 'a', 2: 'a', 3: 'a'}
```

In this example, your dictionary contains multiple instances of the letter a as a value. This is completely okay because values don't have the restriction of needing to be unique.



**Master Real-World Python Skills
With a Community of Experts**
Level Up With Unlimited Access to Our Vast Library
of Python Tutorials and Video Lessons

Watch Now »

 Remove ads

The `dict()` Constructor

You can also build dictionaries with the `dict()` constructor. The arguments to `dict()` can be a series of keyword arguments, another mapping, or an iterable of key-value pairs. Here are the constructor's signatures:

```
Python Syntax >
dict()
dict(**kwargs)
dict(mapping, **kwargs)
dict(iterable, **kwargs)
```

If you call the `dict()` constructor without arguments, then you get an empty dictionary:

```
Python >
>>> dict()
```

```
{}
```

In most cases, you'll use an empty pair of curly braces to create empty dictionaries.

However, in some situations, using the constructor might be more explicit.

If the keys of your dictionary are strings representing valid Python [identifiers](#), then you can specify them as keyword arguments. Here's how you'd create the `MLB_teams` dictionary with this approach:

Python

```
>>> MLB_teams = dict(  
...     Colorado="Rockies",  
...     Chicago="White Sox",  
...     Boston="Red Sox",  
...     Minnesota="Twins",  
...     Milwaukee="Brewers",  
...     Seattle="Mariners",  
... )  
  
>>> MLB_teams  
{  
    'Colorado': 'Rockies',  
    'Chicago': 'White Sox',  
    'Boston': 'Red Sox',  
    'Minnesota': 'Twins',  
    'Milwaukee': 'Brewers',  
    'Seattle': 'Mariners'  
}
```

Again, to build a dictionary using keyword arguments, the keys must be strings holding valid Python names. Otherwise, they won't work as argument names. This is a syntactical restriction of Python.

You can also create a dictionary from an iterable of key-value pairs. Here's how you can build the `MLB_teams` dictionary this way:

```
>>> MLB_teams = dict(  
...     [  
...         ("Colorado", "Rockies"),  
...         ("Chicago", "White Sox"),  
...         ("Boston", "Red Sox"),  
...         ("Minnesota", "Twins"),  
...         ("Milwaukee", "Brewers"),  
...         ("Seattle", "Mariners"),  
...     ]  
... )  
  
>>> MLB_teams  
{  
    'Colorado': 'Rockies',  
    'Chicago': 'White Sox',  
    'Boston': 'Red Sox',  
    'Minnesota': 'Twins',  
    'Milwaukee': 'Brewers',  
    'Seattle': 'Mariners'  
}
```

In this example, you build the dictionary using a list of two-item tuples. The first item acts as the key, and the second is the associated value.

A cool way to create dictionaries from sequences of values is to combine them with the built-in `zip()` function and then call `dict()` as shown below:

```
>>> places = [  
...     "Colorado",  
...     "Chicago",  
...     "Boston",  
...     "Minnesota",  
...     "Milwaukee",  
...     "Seattle",
```

```
... ]  
  
>>> teams = [  
...     "Rockies",  
...     "White Sox",  
...     "Red Sox",  
...     "Twins",  
...     "Brewers",  
...     "Mariners",  
... ]  
  
>>> dict(zip(places, teams))  
{  
    'Colorado': 'Rockies',  
    'Chicago': 'White Sox',  
    'Boston': 'Red Sox',  
    'Minnesota': 'Twins',  
    'Milwaukee': 'Brewers',  
    'Seattle': 'Mariners'  
}
```

The `zip()` function takes one or more iterables as arguments and yields tuples that combine items from each iterable. Note that your original data must be stored in ordered sequences for this technique to work correctly because the order is essential. Otherwise, you can end up with a dictionary that maps keys to values incorrectly.

Using the `.fromkeys()` Class Method

The `dict` data type has a class method called `.fromkeys()` that lets you create new dictionaries from an iterable of keys and a default value. The method's signature looks like the following:

Python Syntax

```
.fromkeys(iterable, value=None, /)
```

The `iterable` argument provides the keys that you want to include in your dictionary. Even though the input iterable can have duplicate items, the final dictionary will have unique keys as usual.

The `value` argument allows you to define an appropriate default value for all the keys. This argument defaults to `None`, which can serve as a good default value in several scenarios. Here's an example of how to create a new dictionary with the `.fromkeys()` method:

```
Python >
>>> inventory = dict.fromkeys(["apple", "orange", "banana", "mango"], 0)
>>> inventory
{'apple': 0, 'orange': 0, 'banana': 0, 'mango': 0}
```

In this example, you create a dictionary to store an inventory of fruits. Initially, you have the list of fruits in stock but don't have the corresponding amounts. So, you use `0` as the default amount in the call to `.fromkeys()`.



[i Remove ads](#)

Accessing Dictionary Values

Once you've created a dictionary, you can access its content by keys. To retrieve a value from a dictionary, you can specify its corresponding key in square brackets `[]` after the dictionary name:

```
Python >
>>> MLB_teams["Minnesota"]
'Twins'
```

```
>>> MLB_teams["Colorado"]
'Rockies'
```

You can subscript a dictionary using specific keys to get their associated values. Key lookup in dictionaries is quite an efficient operation because dictionaries are implemented as hash tables.

If you refer to a key that isn't in the dictionary, then Python raises an exception:

```
Python
>>> MLB_teams["Indianapolis"]
Traceback (most recent call last):
...
KeyError: 'Indianapolis'
```

When you try to access a key that doesn't exist in a dictionary, you get a `KeyError` exception.

Now say that you have the following dictionary with a person's data:

```
Python
>>> person = {
...     "first_name": "John",
...     "last_name": "Doe",
...     "age": 35,
...     "spouse": "Jane",
...     "children": ["Ralph", "Betty", "Bob"],
...     "pets": {"dog": "Frieda", "cat": "Sox"},
... }
```

This dictionary contains a list and dictionary as part of its values. To access the nested list elements, you can use the corresponding key and then the desired index. To access a key-value pair in a nested dictionary, you can use the outer key and then the inner:

```
Python
```

```
>>> person["children"][0]
'Ralph'
>>> person["children"][2]
'Bob'

>>> person["pets"]["dog"]
'Frieda'
>>> person["pets"]["cat"]
'Sox'
```

Using the key and the index, you can access items in nested lists. Similarly, using the outer and inner keys, you can access values in nested dictionaries. Then, the nesting level will define how many keys or indices you'll have to use.

Populating Dictionaries Incrementally

Python dictionaries are dynamically sized data structures. This means that you can add key-value pairs to your dictionaries dynamically, and Python will take care of increasing the dictionary size for you. This characteristic is helpful because it lets you dynamically populate dictionaries with data.

When populating dictionaries, there are three common techniques that you can use. You can:

1. Assign keys manually
2. Add keys in a for loop
3. Build a dictionary with a comprehension

In the following sections, you'll learn how to use these techniques to populate your dictionaries in Python.

Assigning Keys Manually

Sometimes, you start by creating an empty dictionary with an empty pair of curly braces. Then, you start adding new key-value pairs one at a time. Consider the following example where you populate a dictionary with a person's data:

```
Python >

>>> person = {}

>>> person["first_name"] = "John"
>>> person["last_name"] = "Doe"
>>> person["age"] = 35
>>> person["spouse"] = "Jane"
>>> person["children"] = ["Ralph", "Betty", "Bob"]
>>> person["pets"] = {"dog": "Frieda", "cat": "Sox"}


>>> person
{
    'first_name': 'John',
    'last_name': 'Doe',
    'age': 35,
    'spouse': 'Jane',
    'children': ['Ralph', 'Betty', 'Bob'],
    'pets': {'dog': 'Frieda', 'cat': 'Sox'}
}
```

You can populate your dictionaries manually with new key-value pairs by assigning values to new keys. Internally, Python will create the key-value pair for you. It's important to remember that keys are unique. If you assign a new value to an existing key, then the old value will be lost.



[i Remove ads](#)

Adding Keys in a for Loop

You'll also find situations where a `for` loop is a good approach for populating an empty dictionary with new data. For example, say that you have a `range` of numbers and want to create a dictionary that maps each number to its corresponding square value. To create and populate the dictionary, you can use the following code:

Python

```
>>> squares = {}

>>> for integer in range(1, 10):
...     squares[integer] = integer**2
...

>>> squares
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

In this example, you first create an empty dictionary using a pair of curly braces. Then, you use a loop to iterate over a range of integer numbers. Inside the loop, you populate the dictionary using the numbers as keys and the square values as the corresponding values.

Building Dictionaries With Comprehensions

Python has dictionary comprehensions, which is another great tool for creating and populating dictionaries with concise syntax. Here's how you can create your square dictionary with a comprehension:

Python

```
>>> squares = {integer: integer**2 for integer in range(1, 10)}
>>> squares
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

Dictionary comprehensions are expressions that build and return a new dictionary. In this example, you use a comprehension to create a dictionary that maps numbers to their square values.

Note: To learn more about dictionary comprehensions, check out the [Python Dictionary Comprehensions: How and When to Use Them](#) tutorial.

Comprehensions provide a powerful way to create new dictionaries, transform and filter existing dictionaries, and more. They're a great tool for you to have under your belt.

Exploring the `dict` Class Methods

Python dictionaries have several methods that you can call to perform common actions like accessing keys, values, and items. You'll also find methods for updating and removing values. In the following sections, you'll learn about these methods and how to use them in your Python code.

Retrieving Data From Dictionaries

To get started, you'll learn about methods you can use to access the data stored in an existing dictionary. You'll also learn about methods for getting a single key and retrieving all the values, keys, and pairs from a dictionary. These methods are useful in real-world Python programming.

Getting Individual Keys: `.get(key, default=None)`

The `.get()` method provides a convenient way to retrieve the value associated with a key without checking whether the key exists beforehand. The key you want to search for is the first argument to `.get()`. The second argument, which is optional, is a default value that will be used if the target key doesn't exist in the dictionary. Note that the default value of `default` is `None`:

Python

```
>>> inventory = {"apple": 100, "orange": 80, "banana": 100}

>>> inventory.get("apple")
100
>>> print(inventory.get("mango"))
None
```

If the target key exists in the dictionary, then you get the corresponding value. If the key isn't found in the dictionary and the optional `default` argument is specified, then you get `None` as a result.

You can also provide a convenient value to default:

Python

```
>>> inventory.get("mango", 0)
0
```

In this example, the "mango" key isn't in the `inventory` dictionary. Because of this, you get the custom default value (0) as a result.

Retrieving All the Values: `.values()`

The `.values()` method returns a [dictionary view](#) object, which provides a dynamic view of the values in a dictionary:

Python

```
>>> inventory = {"apple": 100, "orange": 80, "banana": 100}

>>> inventory.values()
dict_values([100, 80, 100])
```

The `dict_values` object contains all the values in `inventory`. Note that any duplicate values will be returned as many times as they occur.

Accessing All the Keys: `.keys()`

The `.keys()` method returns a dictionary view object with a dynamic view of the keys in the target dictionary:

```
Python >

>>> inventory = {"apple": 100, "orange": 80, "banana": 100}

>>> inventory.keys()
dict_keys(['apple', 'orange', 'banana'])
```

Again, the view object `dict_keys` contains all the keys in the `inventory` dictionary. Since dictionary keys are unique, you won't get any duplicate keys.

Getting All the Items or Key-Value Pairs: `.items()`

The `.items()` method returns a dictionary view containing tuples of keys and values. The first item in each tuple is the key, while the second item is the associated value:

```
Python >

>>> inventory = {"apple": 100, "orange": 80, "banana": 100}

>>> inventory.items()
dict_items([('apple', 100), ('orange', 80), ('banana', 100)])
```

The `dict_items` view object contains the key-value pairs of your `inventory` dictionary as two-item tuples of the form `(key, value)`.



I ❤ Python



Shop
now »

Remove ads

Adding Key-Value Pairs and Updating Dictionaries

Python's built-in `dict` data type also has methods for adding and updating key-value pairs. For this purpose, you have the `.setdefault()` and `.update()` methods. You'll learn about them in the following sections.

Setting One Key: `.setdefault(key, default=None)`

The `.setdefault()` method lets you set default values to keys. If key is in the dictionary, then the method returns the associated value. If key isn't in the dictionary, it's inserted with default as its associated value. Then, it returns default:

Python

```
>>> inventory = {"apple": 100, "orange": 80}

>>> inventory.setdefault("apple")
100
>>> print(inventory.setdefault("mango"))
None
>>> inventory
{'apple': 100, 'orange': 80, 'mango': None}
>>> inventory.setdefault("banana", 0)
0
>>> inventory
{'apple': 100, 'orange': 80, 'mango': None, 'banana': 0}
```

When you call `.setdefault()` with an existing key, you get the associated value. If the key is missing, you get `None`—which is the default value—and a new key-value pair is inserted. If

the key is missing and you provide a custom default value, then you get the custom default and a new key-value pair.

Updating a Dictionary: `.update([other])`

The `.update()` method merges a dictionary with another dictionary or with an iterable of key-value pairs. If `other` is a dictionary, then `a_dict.update(other)` merges the entries from `other` into `a_dict`. For each key in `other`, you can have one of the following results:

- If the key isn't present in `a_dict`, then the key-value pair from `other` is added to `a_dict`.
- If the key is present in `a_dict`, then the corresponding value in `a_dict` is updated to the value in `other`.

Here's an example showing two dictionaries merged together:

```
Python

>>> config = {
...     "color": "green",
...     "width": 42,
...     "height": 100,
...     "font": "Courier",
... }

>>> user_config = {
...     "path": "/home",
...     "color": "red",
...     "font": "Arial",
...     "position": (200, 100),
... }

>>> config.update(user_config)

>>> config
{
```

```
    'color': 'red',
    'width': 42,
    'height': 100,
    'font': 'Arial',
    'path': '/home',
    'position': (200, 100)
}
```

In this example, you update the `config` dictionary with content from the `user_config` dictionary. Note how the existing keys were updated while the missing ones were added to the end of `config`.

The other argument may also be a sequence of key-value pairs:

Python

```
>>> config.update([("width", 200), ("api_key", 1234)])
>>> config
{
    'color': 'red',
    'width': 200,
    'height': 100,
    'font': 'Arial',
    'path': '/home',
    'position': (200, 100),
    'api_key': 1234
}
```

Here, you pass a list of tuples as an argument to `.update()`. The method updates the existing keys or adds new keys as needed.

Finally, you can also call `.update()` with keyword arguments:

Python

```
>>> config.update(color="yellow", script="__main__.py")
>>> config
```

```
{  
    'color': 'yellow',  
    'width': 200,  
    'height': 100,  
    'font': 'Arial',  
    'path': '/home',  
    'position': (200, 100),  
    'api_key': 1234,  
    'script': '__main__.py'  
}
```

In this example, you call `.update()` with keyword arguments, and the method updates the existing keys or adds new keys as needed.

Removing Data From Dictionaries

Removing key-value pairs is another common operation that you may need to perform on your dictionaries. To do this, the `dict` class provides a few useful methods. In the following sections, you'll learn about these methods and how they work.

Removing Keys: `.pop(key[, default])`

The `.pop()` method removes key-value pairs by keys. If the key exists, then the method returns its associated value. On the other hand, if the key doesn't exist and `default` isn't provided, then you get a `KeyError`. Otherwise, you get the `default` value:

Python

```
>>> inventory = {"apple": 100, "orange": 80, "banana": 100}  
  
>>> inventory.pop("apple")  
100  
>>> inventory  
{'orange': 80, 'banana': 100}
```

```
>>> inventory.pop("mango")
Traceback (most recent call last):
...
KeyError: 'mango'

>>> inventory.pop("mango", 0)
0
```

If the target key isn't present in the dictionary, and the optional default argument is specified, then that value is returned and no exception is raised.

When you want to both delete an item and retain its value, you'll commonly use `.pop()`. If you just want to remove the item, then you typically go with the `del` statement:

```
Python
```

```
>>> del inventory["banana"]
>>> inventory
{'orange': 80}
```

In this example, you remove the "banana" key and its associated value without returning the value as `.pop()` does.

Deleting Items: `.popitem()`

The `.popitem()` method removes a key-value pair from a dictionary. This method returns the removed pair as a tuple of the form `(key, value)`. The pairs are removed in [LIFO](#) (last-in, first-out) order:

```
Python
```

```
>>> inventory = {"apple": 100, "orange": 80, "banana": 100}

>>> inventory.popitem()
('banana', 100)

>>> inventory
```

```
{'apple': 100, 'orange': 80}
```

```
>>> inventory.popitem()
('orange', 80)
>>> inventory
{'apple': 100}

>>> inventory.popitem()
('apple', 100)
>>> inventory
{}
```

Calling `.popitem()` removes a key-value pair from the dictionary and returns it as a two-item tuple. The first item is the key, and the second is the value. Note that the items are removed from right to left, starting with the last item added.

Note: In Python versions prior to 3.6, `.popitem()` returned an arbitrary key-value pair because Python dictionaries were [unordered](#) before this version.

If the dictionary is empty, then `.popitem()` raises a `KeyError` exception:

Python

```
>>> inventory.popitem()
Traceback (most recent call last):
...
KeyError: 'popitem(): dictionary is empty'
```

When you call `.popitem()` on an empty dictionary, you get a `KeyError` because there are no items to remove.

Clearing Dictionaries: `.clear()`

The `.clear()` method removes all the items from a dictionary:

```
>>> inventory = {"apple": 100, "orange": 80, "banana": 100}
>>> inventory
{'apple': 100, 'orange': 80, 'banana': 100}
>>> inventory.clear()
>>> inventory
{}
```

Calling the `.clear()` method on an existing dictionary will remove all the current key-value pairs from the dictionary.



[i Remove ads](#)

Using Operators With Dictionaries

There are a few Python operators you can use with dictionaries. The most notable ones are the membership, equality, and union operators. In the following sections, you'll learn how these operators work with dictionaries by coding and running some quick examples.

Membership: `in` and `not in`

The membership operators `in` and `not in` allow you to determine whether a given key, value, or item is in a dictionary, depending on the target iterable you use.

Note: To learn more about membership tests, check out [Python's "in" and "not in" Operators: Check for Membership](#).

For example, to check whether:

- A key is in a dictionary, you can use the dictionary itself or the `.keys()` method to provide the target iterable
- A value is in a dictionary, you can use the `.values()` method to provide the target iterable
- An item is in a dictionary, you can use the `.items()` method to provide the target iterable

To illustrate, say that you want to check whether a given city is in your `MLB_teams` dictionary.

To do this, you can use the `in` and `not in` operator with the dictionary itself or with the `.keys()` method:

Python

```
>>> MLB_teams = {  
...     "Colorado": "Rockies",  
...     "Chicago": "White Sox",  
...     "Boston": "Red Sox",  
...     "Minnesota": "Twins",  
...     "Milwaukee": "Brewers",  
...     "Seattle": "Mariners",  
... }  
  
>>> "Milwaukee" in MLB_teams  
True  
>>> "Indianapolis" in MLB_teams  
False  
>>> "Indianapolis" not in MLB_teams  
True  
  
>>> "Milwaukee" in MLB_teams.keys()  
True  
>>> "Indianapolis" in MLB_teams.keys()  
False  
>>> "Indianapolis" not in MLB_teams.keys()  
True
```

In the first membership test, you check whether Milwaukee is included in the `MLB_teams` dictionary. Because this city is in the dictionary, you get `True` as a result. Then, you check whether Indianapolis is a member of the dictionary, which returns `False`.

In this first series of examples, you use the dictionary as the target iterable for the `in` and `not in` operators. In the second series of examples, you use `.keys()`. As you can see, both techniques work the same. However, using `.keys()` in membership is redundant and slightly less efficient than using the dictionary directly.

For an execution time comparison, click to open the collapsible section below and run the script on your computer:

Compare Membership Test on `dict` vs `dict.keys()`

Show/Hide

You can also use the `in` and `not in` operators with the `.values()` method to determine whether a given value is in your dictionary:

Python

```
>>> "Rockies" in MLB_teams.values()
True
>>> "Rockies" not in MLB_teams.values()
False
```

In this example, you use the `.values()` method to provide the target iterable for the membership test. This is how to know if a given team is in your dictionary.

Finally, in some situations, you may want to know whether a key-value pair is in the target dictionary. To figure this out, you can use the membership operators with the `.items()` method:

Python

```
>>> ("Boston", "Red Sox") in MLB_teams.items()
True
>>> ("Boston", "Red Sox") not in MLB_teams.items()
False
```

Note that in this example, you use a tuple containing the key-value pair as the value to check. Then, you use the `.items()` method to provide the target iterable.

Equality and Inequality: `==` and `!=`

The equality (`==`) and inequality (`!=`) operators also work with dictionaries. These operators disregard element order when you use them with dictionaries, which is different from what happens with lists, for example:

```
Python >_>
>>> [1, 2, 3] == [3, 2, 1]
False
>>> {1: 1, 2: 2, 3: 3} == {3: 3, 2: 2, 1: 1}
True

>>> [1, 2, 3] != [3, 2, 1]
True
>>> {1: 1, 2: 2, 3: 3} != {3: 3, 2: 2, 1: 1}
False
```

When you compare a list using the equality operator, the result depends on both the content and the order. In contrast, when you compare two dictionaries that contain the same series of key-value pairs, the order of those pairs isn't considered. The inequality operator when used with dictionaries doesn't consider the order of pairs either.



Python

IS GOOD
FOR YOU



*Shop
now »*

Remove ads

Union and Augmented Union: | and |=

The union operator (|) creates a new dictionary by merging the keys and values of two initial dictionaries. The values of the dictionary to the right of the operator take precedence when both dictionaries share keys:

Python

```
>>> default_config = {  
...     "color": "green",  
...     "width": 42,  
...     "height": 100,  
...     "font": "Courier",  
... }  
  
>>> user_config = {  
...     "path": "/home",  
...     "color": "red",  
...     "font": "Arial",  
...     "position": (200, 100),  
... }  
  
>>> config = default_config | user_config  
>>> config  
{  
    'color': 'red',  
    'width': 42,  
    'height': 100,  
    'font': 'Arial',  
    'path': '/home',
```

```
        'position': (200, 100)
    }
```

In this example, you merge the `default_config` and `user_config` dictionaries to build the final `config` dictionary using the `union` operator.

Note that the `"color"` and `"font"` keys are common to both initial dictionaries, `default_config` and `user_config`. After the union, the values associated with these keys in `user_config` prevail. The key-value pairs that didn't exist in `default_config` are added to the end of the new dictionary.

Similarly, the augmented union operator (`|=`) updates an existing dictionary with key-value pairs from another dictionary, `mapping`, or `iterable` of key-value pairs. Again, when the operands share keys, the values from the right-hand side operand take priority:

```
Python
>>> config = {
...     "color": "green",
...     "width": 42,
...     "height": 100,
...     "font": "Courier",
... }

>>> user_config = {
...     "path": "/home",
...     "color": "red",
...     "font": "Arial",
...     "position": (200, 100),
... }

>>> config |= user_config
>>> config
{
    'color': 'red',
    'width': 42,
```

```
    'height': 100,  
    'font': 'Arial',  
    'path': '/home',  
    'position': (200, 100)  
}
```

In this new version of the config dictionary, you don't create a new dictionary for the final configuration. Instead, you update the existing dictionary with the content of user_config using the augmented union operator. In a sense, the augmented union operator works like the `.update()` method, updating an existing dictionary with the content of another.

Use Built-in Functions With Dictionaries

In Python, you'll find several built-in functions that you can use for processing or working with dictionaries. Here's a quick summary of some of these functions:

Function	Description
<code>all()</code>	Returns <code>True</code> if all the items in an iterable are truthy and <code>False</code> otherwise.
<code>any()</code>	Returns <code>True</code> if at least one element in the iterable is truthy and <code>False</code> otherwise.
<code>len()</code>	Returns an integer representing the number of items in the input object.
<code>max()</code>	Returns the largest value in an iterable or series of arguments.
<code>min()</code>	Returns the smallest value in an iterable or series of arguments.
<code>sorted()</code>	Returns a new sorted list of the elements in the iterable.

sum()	Returns the sum of a start value and the values in the input iterable from left to right.
-------	---

As you can see, all these functions have different goals. Also, you can use them with different dictionary components. In the following sections, you'll learn about using these functions to process Python dictionaries.

Checking for Truthy Data in Dictionaries: all() and any()

To start off, say that you have a dictionary that maps products to their amounts. You want to know whether all of the products are stocked. To figure this out, you can use the `all()` function with the dictionary values as a target:

```
Python >

>>> inventory = {"apple": 100, "orange": 80, "banana": 100, "mango": 200}

>>> all(inventory.values())
True

>>> # Update the stock
>>> inventory["mango"] = 0
>>> all(inventory.values())
False
```

In the first call to `all()`, you get `True` because all product amounts differ from 0. In the second example, you get `False` because you're out of mangoes. You can use the `any()` function in a similar fashion.

Note: To learn more about `all()` and `any()`, check out the following tutorials:

- [Python's `all\(\)`: Check Your Iterables for Truthiness](#)

- How to Use any() in Python

You can use these functions with keys as well. To do this, you can use either the dictionary directly or the `.keys()` method. Finally, using these functions with items doesn't make sense because the `.items()` method returns non-empty tuples.

Determining the Number of Dictionary Items: `len()`

Sometimes, you need to know the number of key-value pairs in an existing dictionary. The built-in `len()` function returns exactly that number:

Python

```
>>> MLB_teams = {  
...     "Colorado": "Rockies",  
...     "Chicago": "White Sox",  
...     "Boston": "Red Sox",  
...     "Minnesota": "Twins",  
...     "Milwaukee": "Brewers",  
...     "Seattle": "Mariners",  
... }  
  
>>> len(MLB_teams)  
6
```

When you use a dictionary as an argument for `len()`, the function returns the number of items in the dictionary. In this example, the input dictionary has six key-value pairs, so you get 6 as a result.



UNIQUE SWAG FOR  www.nerdlettering.com

 Remove ads

Finding Minimum and Maximum Values: `min()` and `max()`

If you ever need to find the minimum and maximum value stored in a dictionary, then you can use the built-in `min()` and `max()` functions:

Python

```
>>> computer_parts = {  
...     "CPU": 299.99,  
...     "Motherboard": 149.99,  
...     "RAM": 89.99,  
...     "GPU": 499.99,  
...     "SSD": 129.99,  
...     "Power Supply": 79.99,  
...     "Case": 99.99,  
...     "Cooling System": 59.99,  
... }  
  
>>> min(computer_parts.values())  
59.99  
>>> max(computer_parts.values())  
499.99
```

In this example, you use the `min()` and `max()` functions to find the lower and higher prices with the `.values()` method. You can also use the functions with dictionary keys and even with items. However, note that these functions are mostly used with numeric values.

Sorting Dictionaries by Keys, Values, and Items: `sorted()`

Sorting the items of a dictionary may be another common requirement. To do this, you can use the built-in `sorted()` function. To illustrate, say that you have a dictionary matching student names with their average grades and you want to sort the data by grades.

Here's how you can do this sorting:

Python

```
>>> students = {
...     "Alice": 89.5,
...     "Bob": 76.0,
...     "Charlie": 92.3,
...     "Diana": 84.7,
...     "Ethan": 88.9,
...     "Fiona": 95.6,
...     "George": 73.4,
...     "Hannah": 81.2,
... }

>>> dict(sorted(students.items(), key=lambda item: item[1]))
{
    'George': 73.4,
    'Bob': 76.0,
    'Hannah': 81.2,
    'Diana': 84.7,
    'Ethan': 88.9,
    'Alice': 89.5,
    'Charlie': 92.3,
    'Fiona': 95.6
}

>>> dict(sorted(students.items(), key=lambda item: item[1], reverse=True))
{
    'Fiona': 95.6,
    'Charlie': 92.3,
    'Alice': 89.5,
    'Ethan': 88.9,
    'Diana': 84.7,
    'Hannah': 81.2,
    'Bob': 76.0,
    'George': 73.4
}
```

The `sorted()` function returns a list of sorted values, so you wrap its call with `dict()` to build a new sorted dictionary. In the first call, you sort the items by value in ascending order.

To do this, you use a `lambda` function that takes a two-value tuple as an argument and returns the second item, which has an index of `1`.

In the second call to `sorted()`, you set the `reverse` argument to `True` so that the function returns a list of items stored in reverse order.

Note: To dive deeper into sorting dictionaries, check out the [Sorting a Python Dictionary: Values, Keys, and More](#) tutorial.

You can also sort the dictionary by its keys:

```
Python >

>>> dict(sorted(students.items(), key=lambda item: item[0]))
{
    'Alice': 89.5,
    'Bob': 76.0,
    'Charlie': 92.3,
    'Diana': 84.7,
    'Ethan': 88.9,
    'Fiona': 95.6,
    'George': 73.4,
    'Hannah': 81.2
}
```

In this example, you sort the dictionary by keys using a `lambda` function that returns the first value in the input tuple.

Finally, you can also use `sorted()` to sort the keys and values:

```
Python >

>>> sorted(students)
['Alice', 'Bob', 'Charlie', 'Diana', 'Ethan', 'Fiona', 'George', 'Hannah']
```

```
>>> sorted(students.values())
[73.4, 76.0, 81.2, 84.7, 88.9, 89.5, 92.3, 95.6]
```

In the first call to `sorted()`, you use the dictionary as an argument. This results in a list of sorted keys. Next, you use the `.values()` method to get a list of sorted values.

Summing Dictionary Values: `sum()`

You can also use the built-in `sum()` function with dictionaries. For example, you can use the function to sum up numeric dictionary values or keys.

Note: To learn more about `sum()`, check out Python's [sum\(\): The Pythonic Way to Sum Values](#).

To illustrate, say that you have a dictionary containing daily sales data and want to know the average daily sales. In this scenario, you can do something like the following:

Python

```
>>> daily_sales = {
...     "Monday": 1500,
...     "Tuesday": 1750,
...     "Wednesday": 1600,
...     "Thursday": 1800,
...     "Friday": 2000,
...     "Saturday": 2200,
...     "Sunday": 2100,
... }

>>> sum(daily_sales.values()) / len(daily_sales)
1850.0
```

In this example, you use the `sum()` function to calculate the total sales. To do this, you use the `.values()` method. Then, you compute the average with the help of `len()`.

[Remove ads](#)

Iterating Over Dictionaries

Iterating over data collections, including dictionaries, is a common task in programming. In this sense, Python dictionaries are pretty versatile, allowing you to iterate over their keys, values, and items.

Note: To learn more about dictionary iteration, check out the [How to Iterate Through a Dictionary in Python](#) tutorial.

In the following sections, you'll learn the basics of iterating over Python dictionaries and their components. To kick things off, you'll start by iterating over dictionary keys.

Traversing Dictionaries by Keys

There are two different ways you can iterate over the keys of a dictionary. You can either use the dictionary directly, or use the `.keys()` method. The following examples show how to use these two approaches:

Python

```
>>> students = {  
...     "Alice": 89.5,  
...     "Bob": 76.0,  
...     "Charlie": 92.3,  
...     "Diana": 84.7,  
...     "Ethan": 88.9,  
...     "Fiona": 95.6,  
...     "George": 73.4,
```

```
...     "Hannah": 81.2,
... }

>>> for student in students:
...     print(student)
...
Alice
Bob
Charlie
Diana
Ethan
Fiona
George
Hannah

>>> for student in students.keys():
...     print(student)
...
Alice
Bob
Charlie
Diana
Ethan
Fiona
George
Hannah
```

In these examples, you first iterate over the keys of a dictionary using the dictionary directly in the loop header. In the second loop, you use the `.keys()` method to iterate over the keys. Both loops are equivalent. The second loop is more explicit and readable, but it can be less efficient than the first loop because of the additional method call.

Note that in both loops, you can access the dictionary values as well:

Python



```
>>> for student in students:  
...     print(student, "->", students[student])  
...  
Alice -> 89.5  
Bob -> 76.0  
Charlie -> 92.3  
Diana -> 84.7  
Ethan -> 88.9  
Fiona -> 95.6  
George -> 73.4  
Hannah -> 81.2
```

To access the values in this type of iteration, you can use the original dictionary and a key lookup operation, as shown in the highlighted line.

Iterating Over Dictionary Values

When it comes to iterating through dictionary values, you can use the `.values()` method to feed the loop. To illustrate, say that you're working with the `MLB_teams` dictionary and need to iterate over the team names only:

Python

```
>>> MLB_teams = {  
...     "Colorado": "Rockies",  
...     "Chicago": "White Sox",  
...     "Boston": "Red Sox",  
...     "Minnesota": "Twins",  
...     "Milwaukee": "Brewers",  
...     "Seattle": "Mariners",  
... }  
  
>>> for team in MLB_teams.values():  
...     print(team)  
...  
Rockies
```

```
White Sox
Red Sox
Twins
Brewers
Mariners
```

To iterate over the values of a dictionary, you can use the `.values()` method. In this example, you iterate over the registered MLB teams one by one. Note that when you use the `.values()` method, you can't access the dictionary keys.

Looping Through Dictionary Items

Finally, in many cases, you'll need to iterate over both keys and values in a Python dictionary. In this case, the recommended and most Pythonic approach is to use the `.items()` method:

Python

```
>>> for place, team in MLB_teams.items():
...     print(place, "->", team)
...
Colorado -> Rockies
Chicago -> White Sox
Boston -> Red Sox
Minnesota -> Twins
Milwaukee -> Brewers
Seattle -> Mariners
```

When iterating over keys and values this way, you typically use a tuple of loop variables. The first variable will get the key, while the second will get the associated value. In this example, you have the `place` and `team` variables, which make the code clear and readable.

Exploring Existing Dictionary-Like Classes

In the Python standard library, you'll find a few dictionary-like classes that have been adapted to perform specific tasks. The most notable examples are the following:

Class	Description
<code>OrderedDict</code>	A dictionary subclass specially designed to remember the order of items, which is defined by the insertion order of keys.
<code>Counter</code>	A dictionary subclass specially designed to provide efficient counting capabilities out of the box.
<code>defaultdict</code>	A dictionary subclass specially designed to handle missing keys in dictionaries.

All these classes and a few others are available in the `collections` module found in the Python standard library.

`OrderedDict` isn't that useful anymore because since Python 3.6, dictionaries keep their items in the same insertion order. However, you may find some interesting differences between `dict` and `OrderedDict` that can help you [decide which dictionary](#) best suits your needs.

The `Counter` class provides an efficient tool convenient for counting objects:

```
Python
>>> from collections import Counter

>>> Counter("mississippi")
Counter({'i': 4, 's': 4, 'p': 2, 'm': 1})
```

In this example, you use `Counter` to count the letters in a string. The resulting dictionary's keys are the letters, while the values are the number of occurrences of each letter. Note that

the items in a Counter instance are sorted in descending order out of the box, which can be useful for building rankings.

The defaultdict class automatically creates a new key and generates a default value for it when you try to access or modify a missing key. To illustrate, say that you have the following data in a list of tuples:

Python

```
>>> employees = [
...     ("Sales", "John"),
...     ("Sales", "Martin"),
...     ("Accounting", "Kate"),
...     ("Marketing", "Elizabeth"),
...     ("Marketing", "Linda"),
... ]
```

You want to create a dictionary that uses the departments as keys. Each key should map a list of people working in the department. Here's how you can do this quickly with defaultdict, which is an invaluable tool when you want to group elements together:

Python

```
>>> from collections import defaultdict

>>> departments = defaultdict(list)

>>> for department, employee in employees:
...     departments[department].append(employee)
...
>>> departments
defaultdict(<class 'list'>,
{
    'Sales': ['John', 'Martin'],
    'Accounting': ['Kate'],
    'Marketing': ['Elizabeth', 'Linda']}
```

```
    }  
}
```

In this example, you create a defaultdict called departments and use a [for loop](#) to iterate through your employees list. The line `departments[department].append(employee)` creates the keys for the departments, initializes them to an empty list if necessary, and then appends the employees to each department.



[Learn Python »](#)

[i Remove ads](#)

Creating Custom Dictionary-Like Classes

You can also create custom dictionary-like classes in Python. To do this, you can inherit from one of the following classes:

1. The built-in `dict` class
2. The `collections.UserDict` class

The first approach may lead to some issues, but it can work in situations where you want to add functionality that doesn't imply changing the core functionality of `dict`. The second approach is more reliable and safe, and you can use it in most cases.

Note: To dive deeper into how to create custom dictionary-like classes, check out the [Custom Python Dictionaries: Inheriting From `dict` vs `UserDict`](#) tutorial.

To illustrate, say that you want to create a dictionary-like class that has in-place sorting capabilities. Here's a class that does that: