**15 things you should know about Dictionaries in Python**

Guidelines to use dictionaries in Python

[[](https://amandaiglesiasmoreno.medium.com/?source=post_page-----44c55e75405c--------------------------------)](https://amandaiglesiasmoreno.medium.com/?source=post_page-----44c55e75405c--------------------------------)

[[Towards Data Science](https://towardsdatascience.com/?source=post_page-----44c55e75405c--------------------------------)](https://towardsdatascience.com/?source=post_page-----44c55e75405c--------------------------------)

[Amanda Iglesias Moreno](https://amandaiglesiasmoreno.medium.com/?source=post_page-----44c55e75405c--------------------------------)

·

Follow

Published in

[Towards Data Science](https://towardsdatascience.com/?source=post_page-----44c55e75405c--------------------------------)

·

9 min read

·

Feb 23, 2020

610

6

A close-up of a computer code

Description automatically generated

**1. What is a Python dictionary?**

A dictionary is an **unordered** and **mutable** Python **container** that stores mappings of unique **keys to values**. Dictionaries are written with curly brackets ({}), including **key-value** pairs separated by commas (,). A colon (:) separates each **key** from its **value**.

Three dictionaries are shown below, containing the population of the 5 largest German cities, list of products, and student’s grades.

**2. Create a dictionary with dict() constructor**

Dictionaries can also be created with the built-in function **dict(\*\*kwarg)**. This function takes an arbitrary number of **keywords arguments** (arguments preceded by an identifier **kwarg=value**) as input.

We can also create a dictionary using **another dictionary** in combination with**keyword arguments** (**dict(mapping, \*\*kwarg)**) as follows:

Alternatively, we can construct a dictionary using an iterable (e.g. **list of tuples**). Each tuple must contain two objects. The first object becomes the **key** and the second becomes the **value** of the **dictionary**.

Lastly, we can create a dictionary using two lists. First, we have to build an **iterator of tuples**using **zip(\*iterables)** function. Then, we employ the **dict([iterable, \*\*kwarg])** function to construct the dictionary, as we did previously.

**3. Access values in a dictionary**

To access dictionary **values**, we cannot use a numeric index (as we do with lists or tuples), since the dictionaries are **unordered** containers. Instead, we enclose the **key** using square brackets([]). If we try to access a value using an undefined **key,** a **KeyError** is raised.

To avoid getting an exception with undefined keys, we can use the method **dict.get(key[, default])**. This method returnsthe **value** for**key**if **key** is in the **dictionary**, else returns default. If default is not provided, it returns **None** (but never raises an exception).

**4. Insert elements in a dictionary**

To insert an element in a dictionary, we can use square brackets as follows:

To insert multiple items at once, we can use the method **dict.update([other])**. This method updates the dictionary with the **key/value pairs** from other, overwriting existing **keys**.

As shown above, the **.update()** method accepts as an argument not only another **dictionary,** but also a **list of tuples** or **keyword arguments**. This method modifies the dictionary in-place, returning **None**.

**5. Change elements in a dictionary**

We can change the **value** of an item by accessing the **key**using square brackets ([]). To modify multiple values at once, we can use the **.update()**method, since this function overwrites existing **keys**.

Subsequently, we increase the price of a sofa 100 units, and we modify the grades of two students.

**6. Remove elements in a dictionary**

To remove an element in a dictionary, we can use either the **del dict[key]** keyword or the **dict.pop(key[, default])**method.

The **del dict[key]** keyword removes the given element from the dictionary, raising a **KeyError** if **key** does not exists.

If **key** exists in the dictionary, the **dict.pop(key[, default])** method removes the **item** with the given **key**from the dictionary and returns its **value**. On the contrary, if **key** does not exist in the dictionary, the method returns the **default** value. If no **default** value is provided and **key** does not exist, the **.pop()**method will raise an **exception** (**KeyError**).

**7. Check if a key exists**

To check whether a **key** exists in a **dictionary**, we have to use a **membership operator**. Membership operators are used to test whether a value is found in a sequence (e.g. strings, lists, tuples, sets, or dictionaries). There are two membership operators, as explained below.

* **in** → Evaluates to true if the object on the left side **is**included in the object on the right side.
* **not in** → Evaluates to true if the object on the left side **is not** included in the object on the right side.

As shown above, membership operators (**in** and **not in**) can be used to check whether a key exists in a dictionary, but they can also be used with other sequences in the following manner.

**8. Copy a dictionary**

To copy a dictionary, we can simply use the **dict.copy()** method. This method returns a **shallow copy** of the dictionary. We have to be careful with **shallow copies**, since if your dictionary contains another **container-objects** like lists, tuples, or sets, they will be referenced again and not duplicated.

To avoid this problem, we can create a **deep copy**using **copy.deepcopy(x)** function (defined in the **copy** module) as follows:

The difference between **shallow copies** and **deep copies**is only relevant when the dictionary contains other objects like lists, since those objects will be referenced instead of duplicated (**shallow copy**). To create a fully independent clone of the original dictionary, we have to make a**deep copy**.

\*\* For Shallow Copy, other object that is not ‘list’ category, will not have impact on original dictionary if change the copy one. Only object that is ‘list’ type.

If you want to know more about how to copy a dictionary, you can read the following article where the differences between **shallow copies** and **deep copies** are explained in detail.

**[Python : How to copy a dictionary | Shallow Copy vs Deep Copy](https://thispointer.com/python-how-to-copy-a-dictionary-shallow-copy-vs-deep-copy/?source=post_page-----44c55e75405c--------------------------------" \t "_blank)**

[In this article we will discuss how to create a shallow and deep copy of dictionary in Python. Python's dictionary…](https://thispointer.com/python-how-to-copy-a-dictionary-shallow-copy-vs-deep-copy/?source=post_page-----44c55e75405c--------------------------------" \t "_blank)

[thispointer.com](https://thispointer.com/python-how-to-copy-a-dictionary-shallow-copy-vs-deep-copy/?source=post_page-----44c55e75405c--------------------------------" \t "_blank)

It is important to bear in mind that the **= operator**does not make a copy of the dictionary. It is just another name to refer to the same dictionary, meaning any modification to the new dictionary is reflected in the original one.

**9. Determine the length of the dictionary**

To determine how many **key-value** pairs the dictionary contains, we can use the **len()**function. This function returns the number of items of an object. The input of the function can be a dictionary, but also another type of sequence such as a string, list, tuple, or set.

**10. Loop through a dictionary**

**Iterating through keys**

To iterate over the **keys**, we can use the dictionary directly in a **for** loop as follows:

Alternatively, we can use the **dict.keys()** method. This method returns a view object, containing the**keys** of the dictionary.

**Iterating through values**

If you just need to work with the **values** of a dictionary, then you can use the **dict.values()**method in a **for** loop. This method returns a view object that contains the **values** of the dictionary.

We can compute how many people live in the 5 largest German cities using **dict.values()** method as follows:

As we can observe, almost 9 million people live in the 5 largest German cities.

**Iterating through items**

When you’re working with dictionaries, it’s likely that you need to use the **keys** and the **values**. To loop through both, you can use the **dict.items()** method. This method returns a view object, containing **key-value** pairs as a list of tuples.

We can determine the student with the lowest test score using the **dict.items()** method in combination with a**for loop**as follows:

As shown above, Normando is the student with the lowest test score (2.5).

**11. Dictionary comprehensions**

Python **for-loops** are very handy in dealing with repetitive programming tasks; however, there is another alternative to achieve the same results in a more efficient way: **dictionary comprehensions.**

**Dictionary comprehensions** allow the creation of a dictionary using an elegant and simple syntax: **{key: value for vars in iterable}.**In addition, they are faster than traditional **for-loops**.

We can filter the products with a price lower than 100 euros using both a traditional **for-loop** and a **dictionary comprehension**.

As we can observe, **dictionary comprehensions** provide the same results as traditional **for-loops** in a more elegant way.

**12. Nested dictionaries**

**Nested dictionaries** are dictionaries that contain other dictionaries. We can create a **nested dictionary** in the same way we create a normal dictionary using curly brackets ({}).

The following **nested dictionary** contains information about 5 famous works of art. As we can observe, the **values** of the dictionary are other dictionaries as well.

We can also create the prior **nested dictionary** using the **dict()** constructor, passing the **key: value** pairs as **keyword arguments.**

To access elements in a nested dictionary, we specify the keys using multiple square brackets ([]).

If you want to know more about **nested dictionaries**, you can read the following article where, how to work with **nested dictionaries**(e.g. update items, change elements, and loop though)is explained in detail.

**[Python Nested Dictionary - Learn By Example](https://www.learnbyexample.org/python-nested-dictionary/?source=post_page-----44c55e75405c--------------------------------" \t "_blank)**

[A dictionary can contain another dictionary, which in turn can contain dictionaries themselves, and so on to arbitrary…](https://www.learnbyexample.org/python-nested-dictionary/?source=post_page-----44c55e75405c--------------------------------" \t "_blank)

[www.learnbyexample.org](https://www.learnbyexample.org/python-nested-dictionary/?source=post_page-----44c55e75405c--------------------------------" \t "_blank)

**13. Alternative containers : OrderedDict, defaultdict, and Counter**

The **collections** module provides alternative container datatypes to built-in Python containers. Three dictionary subclasses contained in the **collections** module that are pretty handy when working with Python are: (1)**OrderedDict**, (2)**defaultdict**, and (3)**Counter**.

**OrderedDict**

**OrderedDict** consists of a dictionary that remembers the order in which its contents are added. In Python 3.6+ dictionaries are also **insertion ordered,**meaning they remember the order of items inserted. However, to guarantee element order across other Python versions, we have to use **OrderedDict** containers.

As shown above, **OrderedDict** accepts dictionary methods and functions. Moreover, elements can be inserted, changed, or deleted in the same way as with normal dictionaries.

**defaultdict**

**Defaultdicts** are a dictionary subclass that assign a **default value** when a key is missing (it has not been set yet). They never raise a **KeyError**, if we try to access an item that is not available in the dictionary, instead a new entry is created.

**Defaultdicts** take a function as an argument, and initialize the missing key with the value returned by the function. In the example below, the keys are initialized with different values, depending on the function employed as first argument.

As we can observe, we can pass a **dictionary**or **keywords** as second argument (optional) to initialize the **defaultdict** container.

**Counter**

A **Counter** is a dictionary subclass for counting hastable objects. The function returns a Counter object, where elements are stored as **keys** and their counts are stored as **values**. Using this function, we can easily count the elements of a list, as shown below.

As shown above, we can easily obtain the most frequent elements with the **.most\_common([n])** method. This method returns a list of the n most common elements and their counts.

**14. Create a Pandas DataFrame from a dictionary.**

A Pandas **DataFrame** is a two-dimensional tabular data where each **row**represents an observation and each **column** a variable. A Pandas DataFrame can be created using the **pandas.DataFrame** constructor. This function accepts as input various python containers (e.g. lists, dictionaries, or numpy arrays). However, in this article, we explain only the ways to create a DataFrame that involve the use of dictionaries.

**Create a DataFrame from a dictionary**

We can create a **DataFrame** from a **dictionary,** where the **keys** represent column names, and the **values**represent column data in the following manner:

A screenshot of a computer

Description automatically generated

As we can observe, the default index is just the row number (an integer index beginning at 0). We can modify these indexes by passing the index list to the DataFrame constructor.

A screenshot of a computer

Description automatically generated

**Create a DataFrame from a list of dictionaries**

A **list of dictionaries** can also be used to create a **DataFrame**, where the**keys** represent column names. As before, we can change indexes by passing the index list to the **DataFrame** function.

A screenshot of a graph

Description automatically generated

**15. Functions in Pandas that use dictionaries**

There are several functions in Pandas that use dictionaries as input values, for example, **pandas.DataFrame.rename** and **pandas.DataFrame.replace.**

**pandas.DataFrame.rename**

This function returns a DataFrame with renamed axis labels. We can use a **dictionary** as input where**keys**refer to the old names and **values** to the new ones. Labels not contained in the dictionary remain unchanged.

A screenshot of a graph

Description automatically generated

A table with numbers and text

Description automatically generated

As shown above, we can change **index labels**, providing a **dictionary** to the index parameter. Alternatively, we can modify column names providing the **dictionary** to the **column** parameter.

**pandas.DataFrame.replace**

This function replaces values of the **DataFrame** with other values dynamically. We can use a dictionary with the replace function to modify the **DataFrame**where **keys** represent existing entries, and **values** replacement entries.

A screenshot of a graph

Description automatically generated

A table with numbers and text

Description automatically generated

Article finished! 🍀 As you can see, dictionaries are a really useful tool in Python. I hope this article serves you as a guideline for taking full advantage of them when coding in Python. 💪