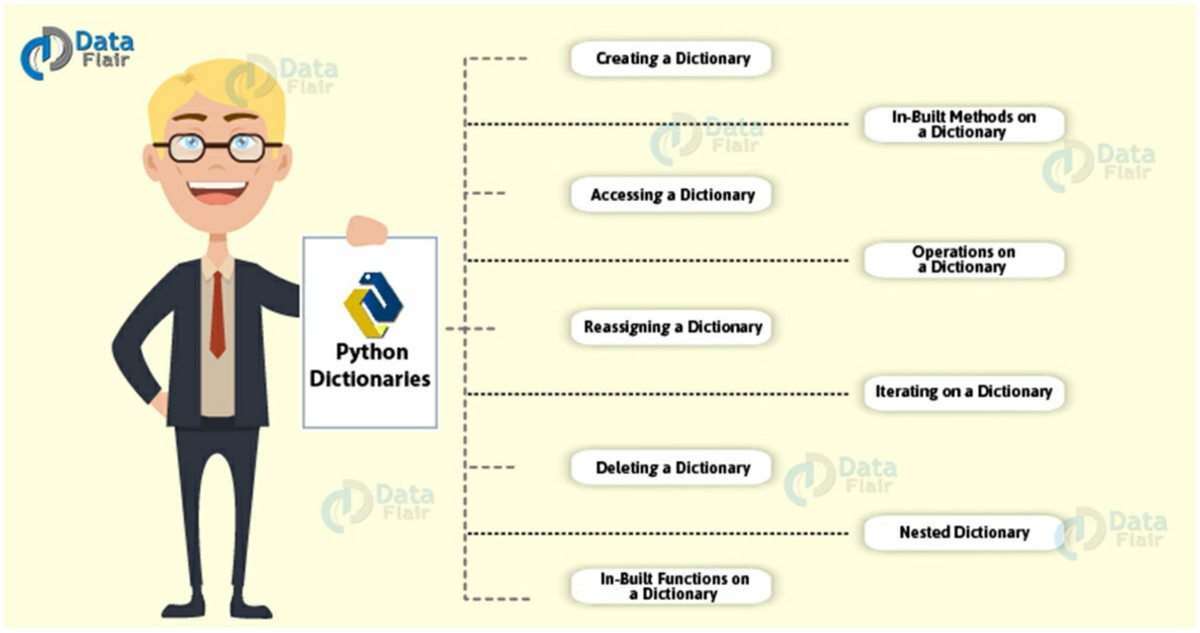
Python Dictionary with Methods, Functions and Dictionary Operations

data-flair.training/blogs/python-dictionary

Contents

1. Python Dictionary Tutorial with Examples

In the last few lessons, we have learned about some Python constructs likelists and tuples. Today, we will have a word about Python dictionary which is another type of data structure in Python.



Python Dictionary

2. An Introduction to Python Dictionaries

A real-life dictionary holds words and their meanings. As you can imagine, likewise, a dictionary holds key-value pairs. Let’s look at how to create one.

3. How to Create a Dictionary in Python

Creating a Python Dictionary is easy as pie. Separate keys from values with a colon(:), and a pair from another by a comma(,). Finally, put it all in curly braces.

```
>>>{'PB&J':'Peanut Butter and Jelly','PJ':'Pajamas'}
```

```
{‘PB&J’: ‘Peanut Butter and Jelly’, ‘PJ’: ‘Pajamas’}
```

Optionally, you can put the dictionary in a variable. If you want to use it later in the program, you must do this.

```
>>> lingo={'PB&J':'Peanut Butter and Jelly','PJ':'Pajamas'}
```

To create an empty dictionary, simply use curly braces and then assign it to a variable

```
>>> dict2={}
>>> dict2
```

{}

>>>type(dict2)

<class ‘dict’>

a. Dictionary Comprehension

You can also create a Python dict using comprehension. This is the same thing that you’ve learned in your math class. To do this, follow an expression pair with a for-statement loop in python. Finally, put it all in curly braces.

1. >>> mydict={x*x:x for x in range(8)}
2. >>> mydict

{0: 0, 1: 1, 4: 2, 9: 3, 16: 4, 25: 5, 36: 6, 49: 7}

In the above code, we created a dictionary to hold squares of numbers from 0 to 7 as keys, and numbers 0-1 as values.

b. Dictionaries with mixed keys

It isn’t necessary to use the same kind of keys (or values) for a dictionary.

1. >>> dict3={1:'carrots','two':[1,2,3]}
2. >>> dict3

{1: ‘carrots’, ‘two’: [1, 2, 3]}

As you can see here, a key or a value can be anything from an integer to a list.

c. dict()

Using the dict() function, you can convert a compatible combination of constructs into a dictionary.

>>>dict(([1,2],[2,4],[3,6]))

{1: 2, 2: 4, 3: 6}

However, this function takes only one argument. So if you pass it three lists, you must pass them inside a list or a tuple. Otherwise, it throws an error.

1. >>>dict([1,2],[2,4],[3,6])
2. Traceback(most recent call last):
3. File "<pyshell#121>", line 1, in <module>

dict([1,2],[2,4],[3,6])

TypeError: dict expected at most 1 arguments, got 3

d. Declaring one key more than once

Now, let’s try declaring one key more than once and see what happens.

1. >>> mydict2={1:2,1:3,1:4,2:4}
2. >>> mydict2

{1: 4, 2: 4}

As you can see, 1:2 was replaced by 1:3, which was then replaced by 1:4. This shows us that a dictionary cannot contain the same key twice.

e. Declaring an empty dictionary and adding elements later

When you don’t know what key-value pairs go in your dictionary, you can just create an empty Python dict, and add pairs later.

1. >>> animals={}
2. >>>type(animals)

<class ‘dict’>

1. >>> animals[1]='dog'
2. >>> animals[2]='cat'
3. >>> animals[3]='ferret'
4. >>> animals

{1: ‘dog’, 2: ‘cat’, 3: ‘ferret’}

Any query on Python Dictionay yet? Leave a comment.

4. Accessing a Dictionary

a. Accessing the entire dictionary

To get the entire dictionary at once, type its name in the shell.

>>> dict3

{1: ‘carrots’, ‘two’: [1, 2, 3]}

b. Accessing a value

To access an item from a list or a tuple, we use its index in square brackets. This is the python syntax to be followed. However, a Python dictionary is unordered. So to get a value from it, you need to put its key in square brackets.

To get the square root of 36 from the above dictionary, we write the following code.

>>> mydict[36]

6

c. get()

The Python dictionary get() function takes a key as an argument and returns the corresponding value.

>>> mydict.get(49)

7

d. When the Python dictionary keys doesn’t exist

If the key you’re searching for doesn’t exist, let’s see what happens.

1. >>> mydict[48]
2. Traceback(most recent call last):
3. File "<pyshell#125>", line 1, in <module>

mydict[48]

KeyError: 48

Using the key in square brackets gives us a KeyError. Now let’s see what the get() method returns in such a situation.

1. >>> mydict.get(48)
2. >>>

As you can see, this didn’t print anything. Let’s put it in the print statement to find out what’s going on.

>>>print(mydict.get(48))

None

So we see, when the key doesn’t exist, the get() function returns the value None. We discussed about it earlier, and know that it indicates absence of value.

5. Reassigning a Python Dictionary

A dictionary is mutable. This means that we can change it or add new items without having to reassign all of it.

a. Updating the Value of an Existing Key

If the key already exists in the dictionary, you can reassign its value using square brackets.

Let’s take a new Python dictionary for this.

>>> dict4={1:1,2:2,3:3}

Now, let’s try updating the value for the key 2.

1. >>> dict4[2]=4
2. >>> dict4

{1: 1, 2: 4, 3: 3}

b. Adding a new key

However, if the key doesn’t already exist in the dictionary, then it adds a new one.

1. >>> dict4[4]=6
2. >>> dict4

{1: 1, 2: 4, 3: 3, 4: 6}

A dictionary cannot be sliced.

6. Deleting a Python Dictionary

You can delete an entire dictionary. Also, unlike a tuple, a Python dictionary is mutable. So you can also delete a part of it.

a. Deleting an entire Python dictionary

To delete the whole Python dict, simply use its name after the keyword ‘del’.

1. >>> del dict4
2. >>> dict4
3. Traceback(most recent call last):
4. File "<pyshell#138>", line 1, in <module>

dict4

NameError: name ‘dict4’ is not defined

b. Deleting a single key-value pair

To delete just one key-value pair, use the keyword ‘del’ with the key of the pair to delete.

Now let’s first reassign dict4 for this example.

>>> dict4={1:1,2:2,3:3,4:4}

Now, let’s delete the pair 2:2

1. >>> del dict4[2]
2. >>> dict4

{1: 1, 3: 3, 4: 4}

A few other methods allow us to delete an item from a dictionary in Python. We will see those in section 8.

7. In-Built Functions on a Dictionary

A function is a procedure that can be applied on a construct to get a value. Furthermore, it doesn’t modify the construct. Python gives us a few functions that we can apply on a dictionary. Take a look.

a. len()

The len() function returns the length of the dictionary in Python. Every key-value pair adds 1 to the length.

```
>>>len(dict4)
```

```
3
```

An empty Python dictionary has a length of 0.

```
>>>len({})
```

```
0
```

b. any()

Like it is with lists an tuples, the any() function returns True if even one key in a dictionary has a Boolean value of True.

```
>>>any({False:False,'':''})
```

```
False
```

```
>>>any({True:False,"":""})
```

```
True
```

```
>>>any({'1':'','':''})
```

```
True
```

c. all()

Unlike the any() function, all() returns True only if all the keys in the dictionary have a Boolean value of True.

```
>>>all({1:2,2:'',"":3})
```

```
False
```

d. sorted()

Like it is with lists and tuples, the sorted() function returns a sorted sequence of the keys in the dictionary. The sorting is in ascending order, and doesn’t modify the original Python dictionary.

But to see its effect, let’s first modify dict4.

```
>>> dict4={3:3,1:1,4:4}
```

Now, let’s apply the sorted() function on it.

```
>>>sorted(dict4)
```

```
[1, 3, 4]
```

Now, let’s try printing the dictionary dict4 again.

```
>>> dict4
```

```
{3: 3, 1: 1, 4: 4}
```

As you can see, the original Python dictionary wasn’t modified.

This function returns the keys in a sorted list. To prove this, let’s see what the type() function returns.

```
>>>type(sorted(dict4))
```

```
<class ‘list’>
```

This proves that sorted() returns a list.

8. In-Built Methods on a Python Dictionary

A method is a set of instructions to execute on a construct, and it may modify the construct. To do this, a method must be called on the construct. Now, let’s look at available methods for dictionaries.

Let’s use dict4 for this example.

```
>>> dict4
```

```
{3: 3, 1: 1, 4: 4}
```

a. keys()

The keys() method returns a list of keys in a Python dictionary.

```
>>> dict4.keys()
```

```
dict_keys([3, 1, 4])
```

b. values()

Likewise, the values() method returns a list of values in the dictionary.

```
>>> dict4.values()
```

```
dict_values([3, 1, 4])
```

c. items()

This method returns a list of key-value pairs.

```
>>> dict4.items()

dict_items([(3, 3), (1, 1), (4, 4)])
```

d. get()

We first saw this function in section 4c. Now, let’s dig a bit deeper.

It takes one to two arguments. While the first is the key to search for, the second is the value to return if the key isn’t found. The default value for this second argument is None.

```
>>> dict4.get(3,0)

3

>>> dict4.get(5,0)

0
```

Since the key 5 wasn’t in the dictionary, it returned 0, like we specified.

Any doubt yet in Python Dictionary? Leave a comment.

e. clear()

The clear function’s purpose is obvious. It empties the dictionary.

1. >>> dict4.clear()
2. >>> dict4

```
{}
```

Let’s reassign it though for further examples.

```
>>> dict4={3:3,1:1,4:4}
```

f. copy()

First, let’s see what shallow and deep copies are. A shallow copy only copies contents, and the new construct points to the same memory location. But a deep copy copies contents, and the new construct points to a different location. The copy() method creates a shallow copy of the dictionary.

1. >>> newdict=dict4.copy()
2. >>> newdict

```
{3: 3, 1: 1, 4: 4}
```

g. pop()

This method is used to remove and display an item from the dictionary. It takes one to two arguments. The first is the key to be deleted, while the second is the value that’s returned if the key isn’t found.

>>> newdict.pop(4)

4

Now, let’s try deleting the pair 4:4.

>>> newdict.pop(5,-1)

-1

However, unlike the get() method, this has no default None value for the second parameter.

1. >>> newdict.pop(5)
2. Traceback(most recent call last):
3. File "<pyshell#191>", line 1, in <module>

newdict.pop(5)

KeyError: 5

As you can see, it raised a KeyError when it couldn’t find the key.

h. popitem()

Let’s first reassign the dictionary newdict.

>>> newdict={3:3,1:1,4:4,7:7,9:9}

Now, we’ll try calling popitem() on this.

>>> newdict.popitem()

(9, 9)

It popped the pair 9:9.

Let’s restart the shell and reassign the dictionary again and see which pair is popped.

1. ================================RESTART: Shell ================================
2. >>> newdict={3:3,1:1,4:4,7:7,9:9}
3. >>> newdict.popitem()

(9, 9)

As you can see, the same pair was popped. We can interpret from this that the internal logic of the popitem() method is such that for a particular dictionary, it always pops pairs in the same order.

i. fromkeys()

This method creates a new dictionary from an existing one. To take an example, we try to create a new dictionary from newdict. While the first argument takes a set of keys from the dictionary, the second takes a value to assign to all those keys. But the second argument is optional.

```
>>> newdict.fromkeys({1,2,3,4,7},0)
```

```
{1: 0, 2: 0, 3: 0, 4: 0, 7: 0}
```

However, the keys don’t have to be in a set.

```
>>> newdict.fromkeys((1,2,3,4,7),0)
```

```
{1: 0, 2: 0, 3: 0, 4: 0, 7: 0}
```

Like we’ve seen so far, the default value for the second argument is None.

```
>>> newdict.fromkeys({'1','2','3','4','7'})
```

```
{‘4’: None, ‘7’: None, ‘3’: None, ‘1’: None, ‘2’: None}
```

j. update()

The update() method takes another dictionary as argument. Then it updates the dictionary to hold values from the other dictionary that it doesn’t already.

1. >>> dict1={1:1,2:2}
2. >>> dict2={2:2,3:3}
3. >>> dict1.update(dict2)
4. >>> dict1

```
{1: 1, 2: 2, 3: 3}
```

9. Operations on a Python Dictionary

We learned about different classes of operators in Python earlier. Let’s now see which of those can we apply on dictionaries.

a. Membership

We can apply the ‘in’ and ‘not in’ operators on a dictionary to check whether it contains a certain key.

For this example, we’ll work on dict1.

```
>>> dict1
```

```
{1: 1, 2: 2, 3: 3}
```

```
>>>2 in dict1
```

```
True
```

```
>>>4 in dict1
```

```
False
```

2 is a key in dict1, but 4 isn’t. Hence, it returns True and False for them, correspondingly.

10. Python Iterate Dictionary

When in a for loop, you can also iterate on a dictionary like on a list, tuple, or set.

>>> dict4

{1: 1, 3: 3, 4: 4}

1. >>>for i in dict4
2. print(dict4[i]*2)

2

6

8

The above code, for every key in the dictionary, multiplies its value by 2, and then prints it.

11. Nested Dictionary

Finally, let’s look at nested dictionaries. You can also place a dictionary as a value within a dictionary.

1. >>> dict1={4:{1:2,2:4},8:16}
2. >>> dict1

{4: {1: 2, 2: 4}, 8: 16}

To get the value for the key 4, write the following code.

>>> dict1[4]

{1: 2, 2: 4}

However, you can’t place it as a key, because that throws an error.

1. >>> dict1={{1:2,2:4}:8,8:16}
2. Traceback(most recent call last):
3. File "<pyshell#227>", line 1, in <module>

dict1={{1:2,2:4}:8,8:16}

TypeError: unhashable type: ‘dict’

If you like the tutorial on Python Dictionary, Please comment.

12. Conclusion

In today’s lesson, we took a deep look at Python Dictionary. We first saw how to create, access, reassign, and delete a dictionary or its elements. Then we looked at built-in functions and methods for dictionaries. After that, we learned about the operations that we can perform on them. Lastly, we learned about nested dictionaries and how to iterate on a Python dictionary.