Theory: Dictionary methods

41 minutes 4 / 12 problems solved

Start practicing

You already know how to create a dictionary and access its items. In this topic, you are going to learn about other features of dictionaries.

§1. Alternative dictionary creation

You know that there are two ways to create a dictionary. Using *curly braces* with a comma-separated list of `key: value` pairs or the `dict` constructor. We will learn about the `fromkeys` method that creates a new dictionary with specified keys and values. This is the syntax for this method:

```
dict.fromkeys(keys, value)
```

The `keys` parameter is a sequence of elements that will become the keys of a new dictionary. The `value` parameter is optional and defaults to `None`, but the user can specify a value for all keys in the dictionary. Look at the example below:

```
planets = {'Venus', 'Earth', 'Jupiter'}

# initializing by default with None
planets_dict = dict.fromkeys(planets)
print(planets_dict)  # {'Jupiter': None, 'Venus': None, 'Earth': None}

# initializing with a value
value = 'planet'
planets_dict = dict.fromkeys(planets, value)


print(planets_dict)  # {'Earth': 'planet', 'Venus': 'planet', 'Jupiter': 'planet'}



# changing the value of 'Jupiter'

planets_dict['Jupiter'] = "giant " + planets_dict['Jupiter']

print(planets_dict)

 # {'Earth': 'planet', 'Venus': 'planet', 'Jupiter': 'giant planet'}
```

The word was added successfully! But now we want to create a dictionary that would store the names of the satellites for those planets. Some planets have several satellites, some do not have them at all, so it is more convenient to use a list as a value.

```
# some satellites of the Solar System
satellites = ['Moon', 'Io','Europa']

# initializing with an empty list
planets_dict = dict.fromkeys(planets, [])
print(planets_dict)  # {'Jupiter': [], 'Venus': [], 'Earth': []}
```

Let's add the items from the `satellites` list to the corresponding planets. Look, this is what happened to our dictionary:

```
planets_dict['Earth'].append(satellites[0])
planets_dict['Jupiter'].append(satellites[1])
planets_dict['Jupiter'].append(satellites[2])
print(planets_dict)

# {'Jupiter': ['Moon', 'Io', 'Europa'], 'Venus': ['Moon', 'Io', 'Europa'], 'Earth'
: ['Moon', 'Io', 'Europa']}
```

Current topic:

- ✓ Dictionary methods — Stage 1

Topic depends on:

- ✓ Dictionary 6★ — Stage 1
- ✓ Arguments 7★ — Stage 1
- ✓ For loop 12★ — Stage 1

Topic is required for:

- Operations with dictionary — Stage 1
- Shelve
- Queries and filters

Table of contents:

We see that all the elements of the `satellites` list have been assigned to all planets in our dictionary. This happened because the `fromkeys` method assigns the same object to all keys. While referring to different keys of the `planets_dict` dictionary, we are still referring to the same list. The difference from the previous example is that if we use mutable objects (a list, a dictionary) as values, all changes will also apply to our dictionary. The solution is to use the dictionary comprehension:

```
planets_dict = {key: [] for key in planets}
```

More details on this operation will be provided in another topic on dictionary operations.

§2. Adding items

Suppose we want to add items to an existing dictionary. You know one way to do it — define a new key and a new value: `existing_dict['new key'] = 'new value'`. But there is another way — use the `update` method. The method updates the dictionary with new elements from another dictionary or an iterable of key-value pairs.

Let's create a dictionary and define months as keys, and the average temperature for this month as values. So we have the following `testable` dictionary:

```
testable = {'September': '16°C', 'December': '-10°C'}
another_dictionary = {'June': '21°C'}

# adding items from another dictionary
testable.update(another_dictionary)

print(testable)  # {'September': '16°C', 'December': '-10°C', 'June': '21°C'}

# adding a key-value pair
testable.update(October='10°C')

print(testable)


# {'September': '16°C', 'December': '-10°C', 'June': '21°C', 'October': '10°C'}
```

If the specified key already exists in the dictionary, the method will update the key with the new value:

```
testable = {'September': '16°C', 'December': '-10°C'}
testable.update(December='-20°C')

print(testable)  # {'September': '16°C', 'December': '-20°C'}
```

§3. Getting and removing items

We learned how to create a dictionary and add elements to it. But what if we need to get some value from the dictionary or also remove an item? The following methods will help you deal with different tasks depending on your needs.

1. Get a value from the dictionary by a key.

As you remember, we can access the value in a dictionary by a key:

```
testable = {}
testable['September'] = '16°C'

print(testable['September'])  # 16°C
```

However, if you try to access a non-existent key, you will get a `KeyError`:

```
print(testable['June'])  # throws a KeyError
```

To avoid the `KeyError`, we can use the `get` method that returns `None` if the specified key is not in the dictionary:

```
# 'get' method does not throw an error
print(testable.get('September'))  # 16°C
print(testable.get('June'))  # None
```

With the `get` method, we can also define the default value that will be returned:

```
print(testable.get('June', 'no temperature'))  # no temperature
```

2. Remove the key from the dictionary and return the value using the `pop` *method.*

If the specified key was found in the dictionary, then the method will remove it and return the value:

```
testable = {'September': '16°C', 'December': '-10°C'}
return_value = testable.pop('December')

print(return_value)  # -10°C
print(testable)  # {'September': '16°C'}
```

If the key was not found, a `KeyError` will appear:

```
testable.pop('July')  # throws a KeyError
```

To get rid of it, we can provide a default argument, and it will return this default value:

```
return_value = testable.pop('July', 'no temperature')
print(return_value)  #  no temperature
```

3. Remove and return the last item (key, value) added to the dictionary using the `popitem` *method:*

```
testable = {'September': '16°C', 'December': '-10°C'}
return_value = testable.popitem()

print(return_value)  # ('December', '-10°C')
print(testable)  # {'September': '16°C'}
```

Pay attention, if the dictionary is empty, a `KeyError` will appear:

```
testable = {}
return_value = testable.popitem()
# KeyError: 'popitem(): dictionary is empty'
```

Before Python 3.7, the `popitem` method removes and returns a *random item* from the dictionary, not *the last one added*.

§4. Cleaning the dictionary

All the methods described above return a value or an item (key, value) upon removing, but sometimes this is not we want. There are two ways that remove an item from the dictionary (they do not return anything) or the entire dictionary at once.

1. Delete (remove from a dictionary) a value by its key with the `del` *keyword:*

```
testable = {'September': '16°C', 'December': '-10°C', 'July': '23°C'}

# this will remove both the key and the value from dictionary object
del testable['September']
print(testable)  # {'December': '-10°C', 'July': '23°C'}

# throws a KeyError, because there's no such key in the dictionary
del testable['May']


# throws a KeyError as we've already deleted the object by the key

del testable['September']



# deletes the whole dictionary

del testable
```

2. *Remove all the items and return an empty dictionary* using the `clear` method:

```
testable = {'September': '16°C', 'December': '-10°C', 'July': '23°C'}

testable.clear()  # remove all elements
print(testable)   # {}
```

§5. Differences in removal methods

You may wonder, is there any difference between `dict = {}` and `dict.clear()` ? Let's say we have another variable that refers to the same dictionary:

```
testable = {'December': '-10°C', 'July': '23°C'}
another_testable = testable
```

Then, the `dict = {}` just creates a new empty dictionary and assigns it to our variable. Let's go back to the example above and assign an empty dictionary to `testable`:

```
testable = {}
print(testable)  # {}
print(another_testable)  # {'December': '-10°C', 'July': '23°C'}
```

`another_testable` still points to the original dictionary with the same elements, so it doesn't change.

In contrast, the `clear` method will clear the dictionary as well as all the objects referring to it:

```
testable = {'December': '-10°C', 'July': '23°C'}

testable.clear()
print(testable)  # {}
print(another_testable)  # {}
```

§6. Recap

What have we learned in this topic?

- a new `fromkeys` method for alternative dictionary creation; we also found out its peculiarities,
- different methods to access elements and/or remove them by key (`get` , `pop`), as well as by adding order (`popitem`),
- discovered how to add new items to the dictionary with the `update` method,
- the `del` keyword and how to use it,
- got acquainted with the features of clearing the dictionary (`dict = {}` and `dict.clear()`).

If you want to see more information on dictionaries, don't forget to check out the Python documentation.

Report a typo

284 users liked this theory. **0** didn't like it. **What about you?**

Start practicing

This content was created 10 months ago and updated 12 days ago. Share your feedback below in comments to help us improve it!

Comments (3) Hints (0) Useful links (0) Show discussion