Theory: Dictionary

16 minutes 9 / 9 problems solved

Start practicing

Imagine that you're a birdwatcher sitting in the park and counting birds that you see. You've observed a dozen pigeons, 5 sparrows, and even one red crossbill! Now, suppose that you want to store these observations for later use. You need to remember exactly how many birds of each kind you've seen. So, a simple list with numbers won't do because you won't be able to tell which number refers to which bird. You need a data type that can associate one thing with another: in our case, the name of the bird with the number of observations.

Luckily, Python has such a type — **dictionary** (`dict`). You can picture a real dictionary — a large book with definitions for a lot of words. The definition contains two parts: the word itself (let's call it a **key**) and the definition for it (a **value**). In our birdwatcher example, the keys are names of the birds ("pigeon", "sparrow", and "red crossbill") and the values are how many birds of that kind we've seen (12, 5 and 1, respectively).

In programming, dictionaries work in a similar way: if we want to store an object, we need to select some key for it and put our object as a value for that key into our dictionary.

§1. Dictionary creation

A dictionary consists of a collection of key-value pairs. Each key-value pair maps the key to its associated value. If you already know the values needed, then the easiest way to create a dictionary is to use the **curly braces** with a comma-separated list of `key: value` pairs. If you want to create an empty dictionary, you can do so with the help of curly braces as well. Note that values in a dictionary can be of **different types**.

```
1  birds = {"pigeon": 12, "sparrow": 5, "red crossbill": 1}
2
prices = {'espresso': 5.0, 'americano': 8.0, 'latte': 10, 'pastry': 'various price
s'}
3  empty_dict = {}
4
5  print(type(birds))  # <class 'dict'>
6  print(type(prices))  # <class 'dict'>
7  print(type(empty_dict))  # <class 'dict'>
```

Another way to create a dictionary is to use the `dict` constructor.

```
1  another_empty_dict = dict()  # using the dict constructor
2
3  print(type(another_empty_dict))  # <class 'dict'>
```

When creating a non-empty dictionary, a dict constructor can take a dictionary as an argument, and / or future dictionary keys as arguments with assigned values, as in the example:

```
1  # note that the future dictionary keys are listed without quotes
2
prices_with_constr = dict({'espresso': 5.0}, americano=8.0, latte=10, pastry='vari
ous prices')
3
4
print(prices_with_constr)  # {'espresso': 5.0, 'americano': 8.0, 'latte': 10, 'pas
try': 'various prices'}
```

When we give the dict constructor dictionary keys with assigned values, as `dict(americano=8.0)`, the left part of the expression is treated like a variable, so it can't be an integer, a string in quotes, a list, a multiword expression, etc. That is, the following lines will give you an error:

Current topic:

- ✓ Dictionary — Stage 1 — 6★

Topic depends on:

- ✓ Integer arithmetic — Stage 1 — 17★
- ✓ Variables — Stage 1 — 17★

Topic is required for:

- ✓ Collections module
- ✓ Dictionary methods — Stage 1
- ✓ Defaultdict and Counter
- Copy of an object
- Working with CSV
- Requests: retrieving data
- Requests: manipulating data
- XML in Python
- Series
- Data types in NumPy
- ✓ Hashable

Table of contents:

↑ Dictionary
§1. Dictionary creation
§2. Accessing the items
§3. Choosing the keys
§4. Recap
Feedback & Comments

```
d1 = dict(888=8.0)
d2 = dict("americano"=8.0)
d3 = dict(["americano", "filter"]=8.0)
d4 = dict(the best americano=8.0)
```

Finally, you can create a nested dictionary. It's a collection of dictionaries inside one single dictionary.

```
# a nested dictionary example
my_pets = {'dog': {'name': 'Dolly', 'breed': 'collie'},
           'cat': {'name': 'Fluffy', 'breed': 'maine coon'}}

# another nested dictionary example
# note that keys of the outer dictionary are numbers
digits = {1: {'Word': 'one', 'Roman': 'I'},
          2: {'Word': 'two', 'Roman': 'II'},
          3: {'Word': 'three', 'Roman': 'III'},

          4: {'Word': 'four', 'Roman': 'IV'},

          5: {'Word': 'five', 'Roman': 'V'}}
```

§2. Accessing the items

The syntax for **accessing** an item is quite simple — square brackets `[]` with a key between them. This approach works both for adding objects to a dictionary and for reading them from there:

```
my_pet = {}

# add 3 keys and their values into the dictionary
my_pet['name'] = 'Dolly'
my_pet['animal'] = 'dog'
my_pet['breed'] = 'collie'

print(my_pet)  # {'name': 'Dolly', 'animal': 'dog', 'breed': 'collie'}


# get information from the dictionary about an added item

print(my_pet['name'])  # Dolly
```

When working with a nested dictionary, getting the right value may be a little harder. As in our example, there are different levels and you need to stop at the right depth.

```
# our nested dictionary once again:
my_pets = {'dog': {'name': 'Dolly', 'breed': 'collie'},
           'cat': {'name': 'Fluffy', 'breed': 'maine coon'}}

print(my_pets['cat'])  # {'name': 'Fluffy', 'breed': 'maine coon'}

print(my_pets['cat']['breed'])  # maine coon
```

§3. Choosing the keys

You can save objects of any type in a dictionary, but not all of them qualify as a key. You need a good, **unique** key for each object in your collection. Still, this is not the only restriction on dictionary keys and we will cover them later. For now, safely use numbers and strings.

When a key has already been added to your dictionary, its old value will be overridden:

```

trilogy = {'IV': 'Star Wars', 'V': 'The Empire Strikes Back', 'VI': 'Return of the
 Jedi'}
    print(trilogy['IV'])  # Star Wars

    trilogy['IV'] = 'A New Hope'
    print(trilogy['IV'])  # A New Hope
```

In Python 3.7 and up, dictionaries do maintain the insertion order for values they store, but in previous versions it is not neccessarily so:

```
alphabet = {}
alphabet['alpha'] = 1
alphabet['beta'] = 2

print(alphabet)
# Python 3.8 output: {'alpha': 1, 'beta': 2}
```

§4. Recap

In this topic we've covered some basics for the dictionary data type in Python:

- how to create a dictionary,
- what is a nested dictionary,
- how to manage dictionary items: keys and values.

In the following lesson you'll get acquainted with basic operations on dictionaries, but first, let's practice some tasks, so you would feel confident using this data type!