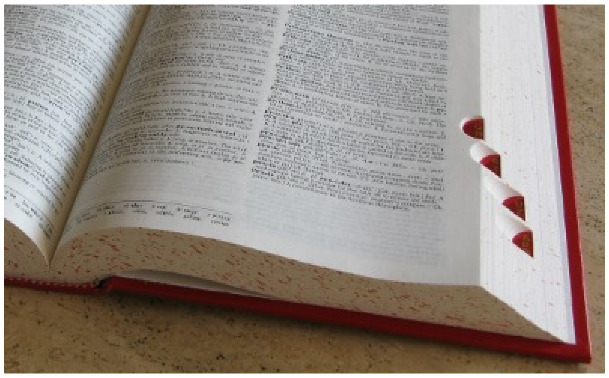
DICTIONARIES

INTRODUCTION



We have already become acquainted with lists in the previous chapter. In this chapter of our online Python course we will present the dictionaries and the operators and methods on dictionaries. Python programs or scripts without lists and dictionaries are nearly inconceivable. Dictionaries and their powerful implementations are part of what makes Python so effective and superior. Like lists they can be easily changed, can be shrunk and grown ad libitum at run time. They shrink and grow without the necessity of making copies. Dictionaries can be contained in lists and vice versa.

But what's the difference between lists and dictionaries? A list is an ordered sequence of objects, whereas dictionaries are unordered sets. But the main difference is that items in dictionaries are accessed via keys and not via their position.

More theoretically, we can say that dictionaries are the Python implementation of an abstract data type, known in computer science as an associative array. Associative arrays consist - like dictionaries of (key, value) pairs, such that each possible key appears at most once in the collection. Any key of the dictionary is associated (or mapped) to a value. The values of a dictionary can be any Python data type. So dictionaries are unordered key-value-pairs. Dictionaries are implemented as hash tables, and that is the reason why they are known as "Hashes" in the programming language Perl.

Dictionaries don't support the sequence operation of the sequence data types like strings, tuples and lists. Dictionaries belong to the built-in mapping type, but so far they are the sole representative of this kind!

At the end of this chapter, we will demonstrate how a dictionary can be turned into one list, containing (key,value)-tuples or two lists, i.e. one with the keys and one with the values. This transformation can be done reversely as well.

EXAMPLES OF DICTIONARIES

Our first example is a dictionary with cities located in the US and Canada and their corresponding population. We have taken those numbers out the "List of North American cities by population" from Wikipedia (https://en.wikipedia.org/wiki/List_of_North_American_cities_by_population)

```
>>> city_population = {"New York City":8550405, "Los
Angeles":3971883, "Toronto":2731571, "Chicago":2720546,
"Houston":2296224, "Montreal":1704694, "Calgary":1239220,
"Vancouver":631486, "Boston":667137}
```

If we want to get the population of one of those cities, all we have to do is to use the name of the city as an index:

```
>>> city_population["New York City"]
8550405
>>> city_population["Toronto"]
2731571
>>> city_population["Boston"]
667137
```

What happens, if we try to access a key, i.e. a city, which is not contained in the dictionary? We raise a KeyError:

```
>>> city["Detroit"]
Traceback (most recent call last):
  File "<stdin>", line 1, in
KeyError: 'Detroit'
>>>
```

If you look at the way, we have defined our dictionary, you might get the impression that we have an ordering in our dictionary, i.e. the first one "New York City", the second one "Los Angeles" and so on. But be aware of the fact that there is no ordering in dictionaries. That's the reason, why the output of the city dictionary, doesn't reflect the "original ordering":

```
>>> city
{'Toronto': 2615060, 'Ottawa': 883391, 'Los Angeles': 3792621,
'Chicago': 2695598, 'New York City': 8175133, 'Boston': 62600,
'Washington': 632323, 'Montreal': 11854442}
```

Therefore it is neither possible to access an element of the dictionary by a number, like we did with lists:

```
>>> city[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in
KeyError: 0
>>>
```

It is very easy to add another entry to an existing dictionary:

```
>>> city["Halifax"] = 390096
>>> city
{'Toronto': 2615060, 'Ottawa': 883391, 'Los Angeles': 3792621,
'Chicago': 2695598, 'New York City': 8175133, 'Halifax': 390096,
'Boston': 62600, 'Washington': 632323, 'Montreal': 11854442}
>>>
```

So, it's possible to create a dictionary incrementally by starting with an empty dictionary. We haven't mentioned so far, how to define an empty one. It can be done by using a empty pair of brackets. The following defines an empty dictionary, called city:

```
>>> city = {}
>>> city
{}
```

Looking at our first examples with the cities and their population numbers, you might have got the wrong impression that the values in the dictionaries have to be different. The values can be the same, as you can see in the following example. In honour to the patron saint of Python "Monty Python", we'll have now some special food dictionaries. What's Python without "ham", "egg" and "spam"?

```
>>> food = {"ham" : "yes", "egg" : "yes", "spam" : "no" }
>>> food
{'egg': 'yes', 'ham': 'yes', 'spam': 'no'}
>>> food["spam"] = "yes"
>>> food
{'egg': 'yes', 'ham': 'yes', 'spam': 'yes'}
>>>
```

Our next example is a simple English-German dictionary:

```
en_de = {"red" : "rot", "green" : "grün", "blue" : "blau",
"yellow":"gelb"}
print(en_de)
print(en_de["red"])
```

What about having another language dictionary, let's say German-French?

```
de_fr = {"rot" : "rouge", "grün" : "vert", "blau" : "bleu",
"gelb":"jaune"}
```

Now it's even possible to translate from English to French, even though we don't have an English-French-dictionary. de_fr[en_de["red"]] gives us the French word for "red", i.e. "rouge":

```
en_de = {"red" : "rot", "green" : "grün", "blue" : "blau",
"yellow":"gelb"}
print(en_de)
print(en_de["red"])
de_fr = {"rot" : "rouge", "grün" : "vert", "blau" : "bleu",
"gelb":"jaune"}
print("The French word for red is: " + de_fr[en_de["red"]])
```

The output of the previous script:

```
{'blue': 'blau', 'green': 'grün', 'yellow': 'gelb', 'red': 'rot'}
rot
The French word for red is: rouge
```

We can use arbitrary types as values in a dictionary, but there is a restriction for the keys. Only immutable data types can be used as keys, i.e. no lists or dictionaries can be used:
If you use a mutable data type as a key, you get an error message:

```
>>> dic = { [1,2,3]:"abc"}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: list objects are unhashable
```

Tuple as keys are okay, as you can see in the following example:

```
>>> dic = { (1,2,3):"abc", 3.1415:"abc"}
>>> dic
{3.1415: 'abc', (1, 2, 3): 'abc'}
```

Let's improve our examples with the natural language dictionaries a bit. We create a dictionary of dictionaries:

```
en_de = {"red" : "rot", "green" : "grün", "blue" : "blau",
"yellow":"gelb"}
de_fr = {"rot" : "rouge", "grün" : "vert", "blau" : "bleu",
"gelb":"jaune"}

dictionaries = {"en_de" : en_de, "de_fr" : de_fr }
print(dictionaries["de_fr"]["blau"])
```

OPERATORS ON DICTIONARIES

Operator	Explanation
len(d)	returns the number of stored entries, i.e. the number of (key,value) pairs.
del d[k]	deletes the key k together with his value
k in d	True, if a key k exists in the dictionary d
k not in d	True, if a key k doesn't exist in the dictionary d

Examples:
The following dictionary contains a mapping from latin characters to morsecode.

```
morse = {
"A" : ".-",
"B" : "-...",
"C" : "-.-.",
"D" : "-..",
"E" : ".",
"F" : "..-.",
"G" : "--.",
"H" : "....",
"I" : "..",
"J" : ".---",
"K" : "-.-",
"L" : ".-..",
"M" : "--",
"N" : "-.",
"O" : "---",
"P" : ".--.",
"Q" : "--.-",
"R" : ".-.",
"S" : "...",
"T" : "-",
"U" : "..-",
"V" : "...-",
"W" : ".--",
"X" : "-..-",
"Y" : "-.--",
"Z" : "--..",
"0" : "-----",
"1" : ".----",
"2" : "..---",
"3" : "...--",
"4" : "....-",
"5" : ".....",
"6" : "-....",
"7" : "--...",
"8" : "---..",
"9" : "----.",
"." : ".-.-.-",
"," : "--..--"
}
```

If you save this dictionary as morsecode.py, you can easily follow the following examples. At first you have to import this dictionary:

```
from morsecode import morse
```

The numbers of characters contained in this dictionary can be determined by calling the len function:

```
>>> len(morse)
38
```

The dictionary contains only upper case characters, so that "a" returns False for example:

```
>>> "a" in morse
False
>>> "A" in morse
True
>>> "a" not in morse
True
```

POP() AND POPITEM()

POP

Lists can be used as stacks and the operator pop() is used to take an element from the stack. So far, so good, for lists, but does it make sense to have a pop() method for dictionaries. After all a dict is not a sequence data type, i.e. there is no ordering and no indexing. Therefore, pop() is defined differently with dictionaries. Keys and values are implemented in an arbitrary order, which is not random, but depends on the implementation.
If D is a dictionary, then D.pop(k) removes the key k with its value from the dictionary D and returns the corresponding value as the return value, i.e. D[k].
If the key is not found a KeyError is raised:

```
>>> en_de = {"Austria":"Vienna", "Switzerland":"Bern",
"Germany":"Berlin", "Netherlands":"Amsterdam"}
>>> capitals = {"Austria":"Vienna", "Germany":"Berlin",
"Netherlands":"Amsterdam"}>>> capital = capitals.pop("Austria")
>>> print(capital)
Vienna
>>> print(capitals)
{'Netherlands': 'Amsterdam', 'Germany': 'Berlin'}
>>> capital = capitals.pop("Switzerland")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Switzerland'
>>>
```

If we try to find out the capital of Switzerland in the previous example, we raise a KeyError. To prevent these errors, there is an elegant way. The method pop() has an optional second parameter, which can be used as a default value:

```
>>> capital = capitals.pop("Switzerland", "Bern")
>>> print(capital)
Bern
>>> capital = capitals.pop("France", "Paris")
>>> print(capital)
Paris
>>> capital = capitals.pop("Germany", "München")
>>> print(capital)
Berlin
>>>
```

POPITEM

popitem() is a method of dict, which doesn't take any parameter and removes and returns an arbitrary (key,value) pair as a 2-tuple. If popitem() is applied on an empty dictionary, a KeyError will be raised.

```
>>> capitals = {"Springfield":"Illinois", "Augusta":"Maine",
"Boston": "Massachusetts", "Lansing":"Michigan", "Albany":"New
York", "Olympia":"Washington", "Toronto":"Ontario"}
>>> (city, state) = capitals.popitem()
>>> (city, state)
('Olympia', 'Washington')
>>> print(capitals.popitem())
('Albany', 'New York')
>>> print(capitals.popitem())
('Boston', 'Massachusetts')
>>> print(capitals.popitem())
('Lansing', 'Michigan')
>>> print(capitals.popitem())
('Toronto', 'Ontario')
>>>
```

ACCESSING NON EXISTING KEYS

If you try to access a key which doesn't exist, you will get an error message:

```
>>> locations = {"Toronto" : "Ontario", "Vancouver":"British
Columbia"}
>>> locations["Ottawa"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Ottawa'
```

You can prevent this by using the "in" operator:

```
>>> if "Ottawa" in locations: print(locations["Ottawa"])
...
>>> if "Toronto" in locations: print(locations["Toronto"])
...
Ontario
```

Another method to access the values via the key consists in using the get() method. get() is not raising an error, if an index doesn't exist. In this case it will return None. It's also possible to set a default value, which will be returned, if an index doesn't exit:

```
>>> proj_language = {"proj1":"Python", "proj2":"Perl",
"proj3":"Java"}
>>> proj_language["proj1"]
'Python'
>>> proj_language["proj4"]
Traceback (most recent call last):
  File "<stdin>", line 1, in
KeyError: 'proj4'
>>> proj_language.get("proj2")
'Perl'
>>> proj_language.get("proj4")
>>> print(proj_language.get("proj4"))
None
>>> # setting a default value:
>>> proj_language.get("proj4", "Python")
'Python'
>>>
```

IMPORTANT METHODS

A dictionary can be copied with the method copy():

```
>>> w = words.copy()
>>> words["cat"]="chat"
>>> print(w)
{'house': 'Haus', 'cat': 'Katze'}
>>> print(words)
{'house': 'Haus', 'cat': 'chat'}
```

This copy is a shallow and not a deep copy. If a value is a complex data type like a list for example, in-place changes in this object have effects on the copy as well:

```
# -*- coding: utf-8 -*-

trainings = { "course1":{"title":"Python Training Course for
Beginners",
                         "location":"Frankfurt",
                         "trainer":"Steve G. Snake"},
              "course2":{"title":"Intermediate Python Training",
                         "location":"Berlin",
                         "trainer":"Ella M. Charming"},
              "course3":{"title":"Python Text Processing Course",
                         "location":"München",
                         "trainer":"Monica A. Snowdon"}
              }

trainings2 = trainings.copy()

trainings["course2"]["title"] = "Perl Training Course for
Beginners"
print(trainings2)
```

If we check the output, we can see that the title of course2 has been changed not only in the dictionary training but in trainings2 as well:

```
{'course2': {'trainer': 'Ella M. Charming', 'name': 'Perl Training
Course for Beginners', 'location': 'Berlin'}, 'course3':
{'trainer': 'Monica A. Snowdon', 'name': 'Python Text Processing
Course', 'location': 'München'}, 'course1': {'trainer': 'Steve G.
Snake', 'name': 'Python Training Course for Beginners',
'location': 'Frankfurt'}}
```

Everything works the way you expect it, if you assign a new value, i.e. a new object, to a key:

```
trainings = { "course1":{"title":"Python Training Course for
Beginners",
                         "location":"Frankfurt",
                         "trainer":"Steve G. Snake"},
              "course2":{"title":"Intermediate Python Training",
                         "location":"Berlin",
                         "trainer":"Ella M. Charming"},
              "course3":{"title":"Python Text Processing Course",
                         "location":"München",
                         "trainer":"Monica A. Snowdon"}
              }

trainings2 = trainings.copy()

trainings["course2"] = {"title":"Perl Seminar for Beginners",
                         "location":"Ulm",
                              "trainer":"James D. Morgan"}
print(trainings2["course2"])
```

The statements *print(trainings2["course2"])* outputs still the original Python course:

```
{'trainer': 'Ella M. Charming', 'location': 'Berlin', 'title':
'Intermediate Python Training'}
```

If we want to understand the reason for this behaviour, we recommend our chapter "Shallow and Deep Copy".

The content of a dictionary can be cleared with the method clear(). The dictionary is not deleted, but set to an empty dictionary:

```
>>> w.clear()
>>> print(w)
{}
```

UPDATE: MERGING DICTIONARIES

What about concatenating dictionaries, like we did with lists? There is someting similar for dictionaries: the update method
update() merges the keys and values of one dictionary into another, overwriting values of the same key:

```
>>> knowledge = {"Frank": {"Perl"}, "Monica":{"C","C++"}}
>>> knowledge2 = {"Guido":{"Python"}, "Frank":{"Perl", "Python"}}
>>> knowledge.update(knowledge2)
>>> knowledge
{'Frank': {'Python', 'Perl'}, 'Guido': {'Python'}, 'Monica': {'C',
'C++'}}
```

ITERATING OVER A DICTIONARY

No method is needed to iterate over the keys of a dictionary:

```
>>> d = {"a":123, "b":34, "c":304, "d":99}
>>> for key in d:
...     print(key)
```

```
...
b
c
a
d
>>>
```

But it's possible to use the method keys(), but we will get the same result:

```
>>> for key in d.keys():
...     print(key)
...
b
c
a
d
>>>
```

The method values() is a convenient way for iterating directly over the values:

```
>>> for value in d.values():
...     print(value)
...
34
304
123
99
>>>
```

The above loop is logically equivalent to the following one:

```
for key in d:
        print(d[key])
```

We said logically, because the second way is less efficient!

If you are familiar with the timeit possibility of ipython, you can measure the time used for the two alternatives:

```
In [5]: %%timeit  d = {"a":123, "b":34, "c":304, "d":99}
for key in d.keys():
    x=d[key]
   ...:
1000000 loops, best of 3: 225 ns per loop

In [6]: %%timeit  d = {"a":123, "b":34, "c":304, "d":99}
for value in d.values():
    x=value
   ...:
```

```
10000000 loops, best of 3: 164 ns per loop

In [7]:
```

CONNECTION BETWEEN LISTS AND DICTIONARIES

If you have worked for a while with Python, nearly inevitably the moment will come, when you want or have to convert lists into dictionaries or vice versa. It wouldn't be too hard to write a function doing this. But Python wouldn't be Python, if it didn't provide such functionalities.

If we have a dictionary

```
D = {"list":"Liste",
"dictionary":"Wörterbuch",
"function":"Funktion"}
```

we could turn this into a list with two-tuples:

```
L = [("list","Liste"), ("dictionary","Wörterbuch"),
("function","Funktion")]
```

The list L and the dictionary D contain the same content, i.e. the information content, or to express ot sententiously "The entropy of L and D is the same". Of course, the information is harder to retrieve from the list L than from the dictionary D. To find a certain key in L, we would have to browse through the tuples of the list and compare the first components of the tuples with the key we are looking for. This search is implicitly and extremely efficiently implemented for dictionaries.

LISTS FROM DICTIONARIES

It's possible to create lists from dictionaries by using the methods items(), keys() and values(). As the name implies the method keys() creates a list, which consists solely of the keys of the dictionary. values() produces a list consisting of the values. items() can be used to create a list consisting of 2-tuples of (key,value)-pairs:

```
>>> w = {"house":"Haus", "cat":"", "red":"rot"}
>>> items_view = w.items()
>>> items = list(items_view)
>>> items
```

```
[('house', 'Haus'), ('cat', ''), ('red', 'rot')]
>>>
>>> keys_view = w.keys()
>>> keys = list(keys_view)
>>> keys
['house', 'cat', 'red']
>>>
>>> values_view = w.values()
>>> values = list(values_view)
>>> values
['Haus', '', 'rot']
>>> values_view
dict_values(['Haus', '', 'rot'])
>>> items_view
dict_items([('house', 'Haus'), ('cat', ''), ('red', 'rot')])
>>> keys_view
dict_keys(['house', 'cat', 'red'])
>>>
```

If we apply the method items() to a dictionary, we don't get a list back, as it used to be the case in Python 2, but a so-called items view. The items view can be turned into a list by applying the list function. We have no information loss by turning a dictionary into an item view or an items list, i.e. it is possible to recreate the original dictionary from the view created by items(). Even though this list of 2-tuples has the same entropy, i.e. the information content is the same, the efficiency of both approaches is completely different. The dictionary data type provides highly efficient methods to access, delete and change elements of the dictionary, while in the case of lists these functions have to be implemented by the programmer.

TURN LISTS INTO DICTIONARIES

Now we will turn our attention to the art of cooking, but don't be afraid, this remains a python course and not a cooking course. We want to show you, how to turn lists into dictionaries, if these lists satisfy certain conditions.
We have two lists, one containing the dishes and the other one the corresponding countries:

```
>>> dishes = ["pizza", "sauerkraut", "paella", "hamburger"]
>>> countries = ["Italy", "Germany", "Spain", "USA"]
```

Now we will create a dictionary, which assigns a dish, a country-specific dish, to a country; please forgive us for resorting to the common prejudices. For this purpose we need the function zip(). The name zip was well chosen, because the two lists get combined like a zipper. The result is a list iterator. This means that we have to wrap a list() casting function around the zip call to get a list so that we can see what is going on:

```
>>> country_specialities_iterator = zip(countries, dishes)
>>> country_specialities_iterator
<zip object at 0x7fa5f7cad408>
```

```
>>> country_specialities = list(countr
countries                          country_specialities_iterator
>>> country_specialities = list(country_specialities_iterator)
>>> print(country_specialities)
[('Italy', 'pizza'), ('Germany', 'sauerkraut'), ('Spain',
'paella'), ('USA', 'hamburger')]
>>>
```

Now our country-specific dishes are in a list form, - i.e. a list of two-tuples, where the first components are seen as keys and the second components as values - which can be automatically turned into a dictionary by casting it with dict().

```
>>> country_specialities_dict = dict(country_specialities)
>>> print(country_specialities_dict)
{'USA': 'hamburger', 'Germany': 'sauerkraut', 'Spain': 'paella',
'Italy': 'pizza'}
>>>
```

Yet, this is very inefficient, because we created a list of 2-tuples to turn this list into a dict. This can be done directly by applying dict to zip:

```
>>> dishes = ["pizza", "sauerkraut", "paella", "hamburger"]
>>> countries = ["Italy", "Germany", "Spain", "USA"]
>>> dict(zip(countries, dishes))
{'USA': 'hamburger', 'Germany': 'sauerkraut', 'Spain': 'paella',
'Italy': 'pizza'}
>>>
```

There is still one question concerning the function zip(). What happens, if one of the two argument lists contains more elements than the other one?

It's easy to answer: The superfluous elements, which cannot be paired, will be ignored:

```
>>> dishes = ["pizza", "sauerkraut", "paella", "hamburger"]
>>> countries = ["Italy", "Germany", "Spain", "USA","
Switzerland"]
>>> country_specialities = list(zip(countries, dishes))
>>> country_specialities_dict = dict(country_specialities)
>>> print(country_specialities_dict)
{'Germany': 'sauerkraut', 'Italy': 'pizza', 'USA': 'hamburger',
'Spain': 'paella'}
>>>
```

So in this course, we will not answer the burning question, what the national dish of Switzerland is.

EVERYTHING IN ONE STEP

Normally, we recommend not to implement too many steps in one programming expression, though it looks more impressive and the code is more compact. Using "talking" variable names in intermediate steps can enhance legibility. Though it might be alluring to create our previous dictionary just in one go:

```
>>> country_specialities_dict = dict(list(zip(["pizza",
"sauerkraut", "paella", "hamburger"], ["Italy", "Germany",
"Spain", "USA"," Switzerland"])))
>>> print(country_specialities_dict)
{'paella': 'Spain', 'hamburger': 'USA', 'sauerkraut': 'Germany',
'pizza': 'Italy'}
>>>
```

On the other hand, the code in the previous script is gilding the lily:

```
dishes = ["pizza", "sauerkraut", "paella", "hamburger"]
countries = ["Italy", "Germany", "Spain", "USA"]
country_specialities_zip = zip(dishes,countries)
print(list(country_specialities_zip))
country_specialities_list = list(country_specialities_zip)
country_specialities_dict = dict(country_specialities_list)
print(country_specialities_dict)
```

We get the same result, as if we would have called it in one go.

DANGER LURKING

Especialy for those migrating from Python 2.x to Python 3.x: zip() used to return a list, now it's returning an iterator. You have to keep in mind that iterators exhaust themselves, if they are used. You can see this in the following interactive session:

```
>>> l1 = ["a","b","c"]
>>> l2 = [1,2,3]
>>> c = zip(l1, l2)
>>> for i in c:
...     print(i)
...
('a', 1)
('b', 2)
('c', 3)
>>> for i in c:
...     print(i)
...
```

This effect can be seen by calling the list casting operator as well:

```
>>> l1 = ["a","b","c"]
>>> l2 = [1,2,3]
>>> c = zip(l1,l2)
>>> z1 = list(c)
>>> z2 = list(c)
>>> print(z1)
[('a', 1), ('b', 2), ('c', 3)]
>>> print(z2)
[]
```

As an exercise, you may muse about the following script.

```
dishes = ["pizza", "sauerkraut", "paella", "hamburger"]
countries = ["Italy", "Germany", "Spain", "USA"]
country_specialities_zip = zip(dishes,countries)
print(list(country_specialities_zip))
country_specialities_list = list(country_specialities_zip)
country_specialities_dict = dict(country_specialities_list)
print(country_specialities_dict)
```

If you start this script, you will see that the dictionary you want to create will be empty:

```
$ python3 tricky_code.py
[('pizza', 'Italy'), ('sauerkraut', 'Germany'), ('paella',
'Spain'), ('hamburger', 'USA')]
{}
$
```