IIT Madras
BSc Degree

Home Lesson-6.2

Lesson-6.1

Dictionaries

Introduction

Let us assume that we want to store the following information in Python:

Country	Capital
Brazil	Brasilia
Russia	Moscow
India	New Delhi
China	Beijing
South Africa	Cape Town

A minor geographical observation: South Africa has three capitals; we have only mentioned the legislative capital for convenience. A geopolitical point: these five countries form a part of a block called BRICS [refer].

Coming back to Python, a dictionary is possibly the most interesting data structure offered by Python. It is basically a look-up table. This is how we would store the details of the BRiCS nations and their capitals:

```
brics = {
            'Brazil': 'Brasilia',
            'Russia': 'Moscow',
            'India': 'New Delhi',
            'China': 'Beijing',
            'South Africa': 'Cape Town'
        }
```

A dictionary is a collection of key-value pairs. In the code given above, `brics` is a dictionary. It has countries mapped to their respective capitals. For instance, `'India'` is mapped to `'New Delhi'`. Here, `'India'` is the key and `'New Delhi'` is the value. That is, the country is the key and its capital is the value. A dictionary object is of type `dict`:

```
print(type(brics))
print(isinstance(brics, dict))
```

To access the value corresponding to a given key, we do the following:

```
print(brics['India'], 'is the capital of', 'India')
print(brics['China'], 'is the capital of', 'China')
```

The value corresponding to a given key can be updated:

```
# Moving to a different capital for South Africa
brics['South Africa'] = 'Pretoria'
# Or we could also store all three capitals
brics['South Africa'] = ('Pretoria', 'Cape Town', 'Bloemfontein')
```

New key-value pairs can be added to a dictionary. Let us expand the horizons of our dictionary to include countries outside the BRICS nations. It no longer makes sense to call this `brics`, so let us create a new dictionary called `globe` which starts off as a copy of `brics`. Recall the `copy` method that we used to copy lists. A similar method is defined for dictionaries:

```
brics = {
            'Brazil': 'Brasilia',
            'Russia': 'Moscow',
            'India': 'New Delhi',
            'China': 'Beijing',
            'South Africa': 'Cape Town'
        }
globe = brics.copy()
globe['Spain'] = 'Madrid'
```

Adding a new key-value pair is as simple as the statement given in line-9 of the code given above. Keys of a dictionary are unique. This means that a dictionary cannot have two or more identical keys mapped to different values. On the other hand, two different keys could have the same value. For example:

```
some_dict = {'key_1': 0, 'key_2': 0}
```

Trying to access a key that is not present in the dictionary will result in a `KeyError`:

```
##### Alarm! Wrong code snippet! ######
some_dict = {'0': 'zero', '1': 'one'}
print(some_dict[0])
##### Alarm! Wrong code snippet! ######
```

More Examples

The key of a dictionary can be any immutable object. There is a small catch here. We will return to this constraint in the next section. Let us look at different combinations key-value pairs that are possible beginning with the basic types: `int, str, float, bool`:

```
# int <> int
squares = {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
# str <> int
months = {'Jan': 31, 'March': 31, 'May': 31, 'Nov': 30}
# int <> str
roll_numbers = {1: 'CS001', 2: 'CS002', 3: 'CS003'}
# str <> str
names = {'Rohit': 'Sharma', 'Saina': 'Nehwal'}
# str <> float
constants = {'pi': 3.14, 'e': 2.71}
# float <> str
fractions = {0.5: 'half', 0.25: 'quarter', 0.3: 'one-third'}
# int <> bool
binary = {0: True, 1: False}
```

Next, we have dictionaries that have `list` and `tuple` as the type of their values:

```
# str <> list
outcomes = {'IND VS AUS': ['IND', 'AUS', 'IND', 'IND'], 'IND VS ENG': ['IND',
'ENG']}
# float <> tuple
bounds = {1.7: (1, 2), 4.3: (4, 5), -1.2: (-2, -1)}
```

Tuples can be keys, provided they don't contain any mutable objects within them:

```
# tuple <> list
T1, T2 = (0, 1), (1, 2)
random_numbers = {T1: [0.1, 0.4, 0.9], T2: [1.1, 1.9]}
```

Towards the end, we will look at an example where a tuple cannot be a key. Finally, the richness of dictionaries comes out in the following example:

```
# mixed
report_card = {
                'name': 'Ramanujan',
                'age': 18,
                'school': 'KV',
                'marks': (75, 80, 60, 95, 100)
              }
```

More on Keys

Earlier, it was mentioned that the keys of dictionaries have to be immutable. This statement is not entirely accurate. In this section, we will explore why. What happens if we use a list as a key?

```
##### Alarm! Wrong code snippet #####
some_list = [0, 1]
bad_dict = {some_list: 0}
##### Alarm! Wrong code snippet #####
```

It throws a `TypeError` with the following message: `unsashable type: 'list'`. A list cannot be a key in a dictionary; but the error message doesn't talk about immutability, instead it says that the `list` type is unhashable. A more accurate statement about keys in a dictionary is given below:

> The keys of a dictionary must be hashable.

To understand what we mean by the term hashable, we shall briefly look at the way Python implements dictionaries. The following section on hash tables is a bit involved and can be skipped.

Hash Tables

Python dictionaries are implemented using a data structure called a hash table. It is best to think about a hash table as a book-rack that has a number of rows. Picture the key-value pairs as books that are going to be stored in these racks. To access a book, we need to know the row number in which it is present. This is where the idea of a hash function comes in. The hash function is denoted by h and converts the key to the row number.

The hash function accepts a key k as input and returns a value, $h(k)$, as output. This is called the hash value. In our analogy, the hash value is synonymous with the rack number. Once we know the rack number, the book (key-value) stored in it can be easily retrieved. The description is somewhat naive, but you get the point.

Now, an object in Python is hashable if it has a hash value which never changes during its lifetime and can be compared to other objects. Most of the immutable objects that we have seen so far are hashable: `int, float, str, bool`. Mutable containers such as lists are not hashable. So, can we just go back to the original definition and claim that all immutable objects can be used as keys in dictionaries? No! Consider the following example:

```
##### Alarm! Wrong code snippet #####
some_tuple = ([0, 1], [2, 3])
bad_dict = {some_tuple: 0}
##### Alarm! Wrong code snippet #####
```

Though `some_tuple` is immutable, it contains a sequence of lists which are mutable. According to the Python documentation, immutable containers are hashable only if their elements are hashable. So, `some_tuple` is not hashable, and hence it cannot be used as a key! For a better explanation, check out the docs.

Iterating over Dictionaries

We can iterate over the keys of a dictionary:

```
squares = {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}    # key is a number, value is its square
for key in squares.keys():
    print(f'The square of {key} is {squares[key]}')
```

`squares.keys()` returns a sequence of keys over which we can iterate. Python makes things even more simple and lets us drop the `keys` method.

```
squares = {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}    # key is a number, value is its square
for key in squares:
    print(f'The square of {key} is {squares[key]}')
```

We can also iterate over the key-value pairs in a dictionary:

```
squares = {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}    # key is a number, value is its square
for key, value in squares.items():
    print(f'The square of {key} is {value}')
```

Growing a Dictionary

An empty dictionary can be defined in one of the following ways:

```
D1 = dict()
D1[0] = 1
D2 = { }
D2[0] = 1
```

Let us now solve the following problem:

> Accept a list of words as input and create a dictionary that maps words to their lengths.

Solution

```
words = ['interstellar', 'dunkirk', 'inception', 'tenet']
lengths = dict()
for word in words:
    lengths[word] = len(word)
print(lengths)
```

A piece of trivia: what is common among the words in the list `words`?

Mutability

Like lists dictionaries are mutable objects. To see the mutability of `dict` objects in action, consider the following code:

```
dict_1 = {'one': 1, 'two': 2, 'three': 3}
dict_2 = dict_1
dict_2['four'] = 4
print(dict_1, dict_2)
print(dict_1 is dict_2)
```

We see that `dict_2` is alias of `dict_1` and both point to the same object. If we want a new `dict` object with the same contents as `dict_1`, we could either use the `copy` method or the `dict` built-in function:

```
dict_1 = {'one': 1, 'two': 2, 'three': 3}
dict_2 = dict_1.copy()      # dict(dict_1) also works
dict_2['four'] = 4
print(dict_1, dict_2)
print(dict_1 is not dict_2)
```

The last line prints `True` which confirms that we have two different objects. So modifying one doesn't affect the other. But note that `copy` only produces a shallow copy. As long as the values are immutable, this doesn't matter. But if we have mutable values, then we have a problem:

```
dict_1 = {'one': [1], 'two': [1, 1], 'three': [1, 1, 1]}
dict_2 = dict_1.copy()
dict_2['one'].append(100)
print(dict_1, dict_2)
print(dict_1 is not dict_2)
print(dict_1['one'] is dict_2['one'])
```

Here, we see that the value corresponding to the key `'one'` in both dictionaries gets affected. This is because `dict_1['one']` and `dict_2['one']` are still the same object. This can be seen from the last statement of the code given above. To set this right, we need to do a deepcopy:

```
from copy import deepcopy
dict_1 = {'one': [1], 'two': [1, 1], 'three': [1, 1, 1]}
dict_2 = deepcopy(dict_1)
dict_2['one'].append(100)
print(dict_1, dict_2)
print(dict_1 is not dict_2)
print(dict_1['one'] is not dict_2['one'])
```