Python: Dictionaries

Introduction to Computer Programming
Bachelor in Data Science

Roberto Guidi

roberto.guidi@supsi.ch

Fall 2021

SUPSI University of Applied Sciences and Arts
of Southern Switzerland

Content created in collaboration with: L. Grossi, F. Landoni

Dictionaries

In Python, another data structure that can be used to store collections of items, in a completely different way compared to lists, are dictionaries.

They allow to store items by assigning to each items a key instead of an index (position).

Dictionaries are not sequences but rather a mapping type.

They are used mostly when the collection's items need to be associated with a label (key)

In other programming languages there are similar structures known as maps or associative arrays.

Dictionaries

The most important properties of dictionaries in Python are:

- They do not mantain a left-to-right order that we can rely on.
- Items are accessed by key: using a key (label) is possible to retrieve the item associated.
- Content can by any sort of object: it is possible to store in a dictionary any kind of object.
- Similar to lists, they are mutable, meaning that their content can be changed and they can grow or shrink if necessary.
- Keys must be unique: a key can be used only once in a dictionary.
- Keys must be immutable: we can use as key objects of immutable type such as numbers, strings, booleans or tuples.

Dictionaries: creation

In order to create a dictionary we use the curly braces `{ }`.

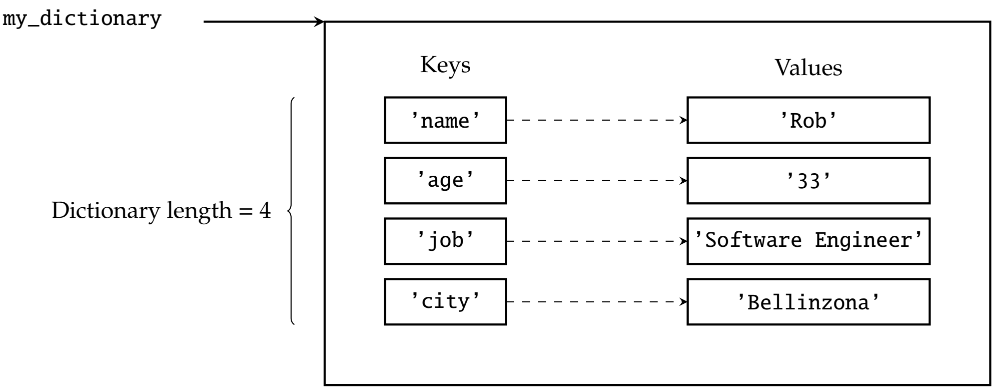
Inside the braces we can then specify couples of keys and values.

```
# an empty dictionary
D = {}

# a dictionary with 3 key-value couples
D = {"name": "Rob", "age": 33, "job": "Software Engineer"}
```

In the last example above we created a dictionary containing the following keys: name, age, job with the respective items associated.

Dictionaries: internal structure



my_dictionary

Dictionary length = 4

Keys	Values
'name'	'Rob'
'age'	'33'
'job'	'Software Engineer'
'city'	'Bellinzona'

The length of a dictionary is equal to the number of key:value pairs that it contains.
The statement `len(my_dictionary)` returns the current length if the dictionary.

Dictionaries: basic usage

In order to access an item of a dictionary we use the following notation:

```
dictionary_variable_name[key_name]
```

It is basically the same we used for lists, except that we specify a key instead of an index.

Example:

```
my_dictionary = {"name": "George", "credits": 200}
item = my_dictionary["name"]
item2 = my_dictionary["credits"]
```

Dictionaries: key assignment

To modify an element at a given key inside the dictionary it is possible to use the assignment operator =.

This operation practically allow us to change in-place the content of a dictionary.

Example:

```
>>> my_dictionary = {"name": "George", "credits": 200}
>>> my_dictionary["name"] = "John"
>>> my_dictionary
{'name': 'John', 'credits': 200}
```

Dictionaries: insert an item

The insertion an element at a new key inside the dictionary is possible by using the same key assignment notation just seen.

The only difference is that if the specified key is not existing, this will be added.

Example:

```
>>> my_dictionary = {}
>>> my_dictionary["name"] = "Obi-Wan"
>>> my_dictionary["age"] = 57
>>> my_dictionary
{’name’: ’Obi-Wan’, ’age’:57}
>>> my_dictionary["name"]
Obi-Wan
```

Dictionaries: other ways to create a dictionary

Other than the literal notation used so far to create a dictionary, there are also some other ways.

One possibility is to use the `dict()` function and passing the keys and values using the keyword arguments function syntax.

Example:

```
>>> my_dictionary = dict(name="Obi-Wan", age=57)
>>> my_dictionary
{'name': 'Obi-Wan', 'age':57}
```

Dictionaries: other ways to create a dictionary

It is also possible to create a dictionary starting from two lists containing respectively the keys and the items.

To do this we can use the function `zip()` in combination with `dict()`.

Example:

```
>>> my_keys = ["name", "age", "job"]
>>> my_items = ["Obi-Wan", 57, "jedi master"]
>>> my_dictionary = dict(zip(my_keys, my_items))
>>> my_dictionary
{'name': 'Obi-Wan', 'age':57, 'job': 'jedi master'}
```

If the two lists do not have the same number of element, the zip function will stop creating key-value pairs using the smallest list.

Dictionaries: removing an element

To delete an entry from the dictionary we can use the *del* keyword.

To use it just write the `del` keyword before the instruction to access a particular item as in the following example:

```
>>> my_dictionary = {"a": 1, "b": 5, "e":9}
>>> del my_dictionary["b"]
>>> my_dictionary
{'a': 1, 'e': 9}
```

Pay attention that if the given key is not existing, the deletion will cause an error.

Dictionaries: checking key existence

When dealing with dictionaries, it may happen that we try to access a key that is not existing.

If this occurs, we would end up with an error as shown below:

```
>>> my_dictionary = {"a": 1, "b": 5, "e":9}
>>> my_dictionary["d"]
Traceback (most recent call last):
  File "<input>", line 1, in <module>
KeyError: ’d’
```

In this case since we already knew the key wasn’t there, we might think that it is a programming error by the developer.

However, sometimes, for example when receiving data from external sources, it might be that we don’t exactly know when a key is there or not.

Dictionaries: checking key existence

In order to solve this problem we can use the membership expression `in`.

With this expression we can check (test) if a key is available before using it, thus avoiding the previous error.

```
>>> my_dictionary = {"a": 1, "b": 5, "e":9}
>>> "d" in my_dictionary
False

>>> if not "d" in my_dictionary:
        print("d is missing")
    else:
        print(f"there it is {my_dictionary["d"]}")
d is missing
```

Dictionaries: getting all keys

To obtain the list of keys present in a dictionary we can use its keys() method combined with the list() function.

```
>>> my_dictionary = {"a": 1, "b": 5, "e":9}
>>> keys = list(my_dictionary.keys())
>>> keys
[’a’, ’b’, ’e’]
```

Dictionaries: getting all values

To obtain the list of items present in a dictionary we can use its values() method combined with the list() function.

```
>>> my_dictionary = {"a": 1, "b": 5, "e":9}
>>> values = list(my_dictionary.values())
>>> values
[1, 5, 9]
```

Dictionaries: getting all key-value pairs

To obtain the list of key-value pairs present in a dictionary we can use its items() method combined with the list() function.

```
>>> my_dictionary = {"a": 1, "b": 5, "e":9}
>>> values = list(my_dictionary.items())
>>> values
[('a', 1), ('b', 5), ('e', 9)]
```

This operation returns a list of tuples.

Tuples are basically sequence structures similar to lists, but that are immutable and thus cannot be changed.

Dictionaries: checking value existence

It is possible to ask a dictionary if it contain a certain value by exploiting its `values()` method, combined with the membership expression `in`.

This works similarly as what seen before for keys.

```
>>> my_dictionary = {"a": 1, "b": 5, "e":9}
>>> 1 in my_dictionary.values()
True

>>> 2 in my_dictionary.values()
False
```

Dictionaries: iteration

Dictionaries can be used in loops in order to iterate over their content.

If we use a dictionary in a for loop, by default we are traversing its keys.

```
my_dictionary = {"a": 1, "b": 5, "e":9}
for x in my_dictionary:
    print(x)
# prints a b e
```

If we need to iterate over the actual values we can simply access the value at each iteration using the current key:

```
my_dictionary = {"a": 1, "b": 5, "e":9}
for x in my_dictionary:
    print(my_dictionary[x])
# prints 1 5 9
```

Dictionaries: sorting keys

As said before, dictionaries are not ordered sequences.

When creating a dictionary with keys in some order, keys may not be returned in the same order later.

Example:

```
>>> my_dictionary = {"a": 1, "b": 5, "e":9}
>>> my_dictionary
{’a’: 1, ’e’: 9, ’b’:5}
```

If for any reason we need to order the items in our dictionary we can use `keys()` and loops to solve the task.

Dictionaries: sorting keys

```
>>> my_dictionary = {"a": 1, "e": 5, "b":9}
>>> keys = list(my_dictionary.keys())
>>> keys
[’a’, ’e’, ’b’]

>>> keys.sort()
>>> keys
[’a’, ’b’, ’e’]

>>> for k in keys:
        print(f’{k} => {my_dictionary[k]}’)
a => 1
b => 9
e => 5
```

As seen in the example we need to perform a few operations to achieve our result.

Dictionaries: sorting keys

A simpler alternative to do the same thing is to use the sorted() function.

```
>>> my_dictionary = {"a": 1, "e": 5, "b":9}
>>> for k in sorted(my_dictionary):
        print(f'{k} => {my_dictionary[k]}')
a => 1
b => 9
e => 5
```

Used in this way, the sorted() function gives us back automatically the sorted keys of our dictionary.

Dictionaries methods

Along with the ones that we already treated, namely `keys(), values() and items()` there are some more dictionaries methods that we want to explore together.

In the following slides we will go through a selection of useful methods.

A description of all the dictionary methods is available on the Official Documentation

Dictionaries methods: pop

Similarly to what already seen for the lists, also dictionaries have a pop() method to remove items.

The pop() method removes the item at the given key and returns it.

If the key is not in the dictionary it will raise an error.

```
>>> my_dictionary = {"name": "Darth Vader", "clan": "sith"}
>>> clan = my_dictionary.pop("clan")
>>> my_dictionary
{'name': 'Darth Vader'}

>>> my_dictionary.pop("age")
Traceback (most recent call last):
  File "<input>", line 1, in <module>
KeyError: 'age'
```

Dictionaries methods: get

To avoid errors that occur when we try to access a key that do not exists we can use the method `get(key)`

This method returns the value associated with the key if it exists, otherwise it returns `None`, without raising errors.

```
>>> my_dictionary = {"name": "Darth Vader", "clan": "sith"}
>>> clan = my_dictionary.get("clan")
>>> clan
'sith'

>>> clan = my_dictionary.get("age")
>>> print(clan)
None
```

Dictionaries methods: update

Another useful operation is that to merge multiple dictionaries together.
To perform this action the update(dictionary) comes at hand.

```
>>> my_dictionary = {"name": "Darth Vader", "clan": "sith"}
>>> my_dictionary_2 = {"age": 40, "job": "lord"}
>>> my_dictionary.update(my_dictionary_2)
>>> my_dictionary
{’name’: ’Darth Vader’, ’clan’: ’sith’, ’age’: 40, ’job’: ’lord’}
```

When merging two dictionaries that have common keys, the key defined in the second dictionary will overwrite the one in the first.

```
>>> my_dictionary = {"name": "Darth Vader", "clan": "sith"}
>>> my_dictionary_2 = {"name": "Anakin", "job": "driver"}
>>> my_dictionary.update(my_dictionary_2)
>>> my_dictionary
{’name’: ’Anakin’, ’clan’: ’sith’, ’age’: 40, ’job’: ’driver’}
```

Nested Dictionaries

As seen for the list, also when working with dictionaries, we have the possibility to nest them one into the other. In the previous examples we just created a dictionary with the information about a film character.

If the information is more complex we can use nested dictionaries to better structure it.

In the following example we use nested dictionaries and lists to organize the information:

```
my_character = {"name": {"first": "Sheev", "last": "Palpatine"},
                "age": 57,
                "jobs": ["sith lord", "emperor", "republic chancellor"]}
```

Nested Dictionaries

If we try to analyze the created dictionary we can see that we are dealing with a more complex object than before.

The dictionary is still composed by three top level keys: `name, age, jobs`.

However, this time, the values associated with the keys are not only simple strings or numbers but also other dictionaries or lists.

This gives us the flexibility to build structures that allow us to work with compound objects.

Let's see in the next slide a few examples to better understand how that works.

Nested Dictionaries

```
my_character = {"name": {"first": "Sheev", "last": "Palpatine"},
                "age": 57,
                "jobs": ["sith lord", "emperor", "republic chancellor"]}

print(my_character["name"])
# prints {’first’: ’Sheev’, ’last’: ’Palpatine’}
# The result is a dictionary containing first and last names.

print(my_character["name"]["last"])
# prints ’Palpatine’
# we use the two keys to access the nested dictionary

print(my_character["jobs"])
# prints [’sith lord’, ’emperor’, ’republic chancellor’]
# The result is the nested list associated with the ’jobs’ key

print(my_character["jobs"][1])
# prints ’emperor’
# we access the second element of the nested jobs list
```

Summary

- Dictionaries
- Basic operations
- Keys and values
- Iteration
- Methods
- Nested dictionaries

Bibliography

- Learning Python 5th edition, Oreilly - Mark Lutz: Chapters 4, 8
- Python Crash Course, no starch press - Eric Matthes: Chapters 3, 4
- Python Official Documentation: https://docs.python.org/3/tutorial/
- LearnByExample: https://www.learnbyexample.org/python/