✦ Member-only story

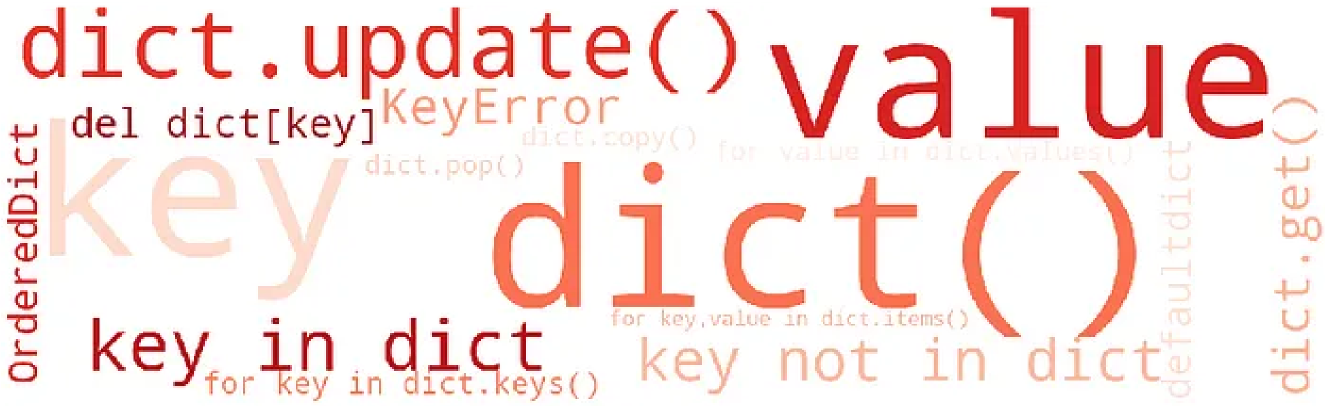
15 things you should know about Dictionaries in Python

Guidelines to use dictionaries in Python

Amanda Iglesias Moreno · Follow

Published in Towards Data Science · 9 min read · Feb 23, 2020

610 6



1. What is a Python dictionary?

A dictionary is an **unordered** and **mutable** Python **container** that stores mappings of unique **keys to values**. Dictionaries are written with curly brackets ({}), including **key-value** pairs separated by commas (,). A colon (:) separates each **key** from its **value**.

Three dictionaries are shown below, containing the population of the 5 largest German cities, list of products, and student’s grades.

```python
# dictionary containing the population of the 5 largest german cities
population = {'Berlin': 3748148, 'Hamburg': 1822445, 'Munich': 1471508, 'Cologne': 1085664, 'Fran

# dictionary containing a list of products' prices
products = {'table': 120, 'chair': 40, 'lamp': 14, 'bed': 250, 'mattress': 100}

# dictionary containing students grades
grades = {'Alba': 9.5, 'Eduardo': 10, 'Normando': 3.5, 'Helena': 6.5, 'Claudia': 7.5}
```

dictionaries.py hosted with ❤ by GitHub view raw

2. Create a dictionary with dict() constructor

Dictionaries can also be created with the built-in function **dict(**kwarg)**. This function takes an arbitrary number of **keywords arguments** (arguments preceded by an identifier **kwarg=value**) as input.

```python
# create a dictionary with dict() function using keyword arguments
# dictionary - ages of students
students_ages = dict(Amanda=27, Teresa=38, Paula=17, Mario=40)

print(students_ages)
# {'Amanda': 27, 'Teresa': 38, 'Paula': 17, 'Mario': 40}
```

dictionaries_dict_constructor.py hosted with ❤ by GitHub view raw

We can also create a dictionary using **another dictionary** in combination with **keyword arguments** (**dict(mapping, **kwarg)**) as follows:

```python
# create a dictionary with dict() function using another dictionary and keyword arguments
# dictionary - ages of students
students_ages = dict({'Amanda': 27, 'Teresa': 38}, Paula=17, Mario=40)

print(students_ages)
# {'Amanda': 27, 'Teresa': 38, 'Paula': 17, 'Mario': 40}
```

dictionaries_dict_constructor_2.py hosted with ❤ by GitHub view raw

Alternatively, we can construct a dictionary using an iterable (e.g. **list of tuples**). Each tuple must contain two objects. The first object becomes the **key** and the second becomes the **value** of the **dictionary**.

```python
# create a dictionary with dict() function using an iterable (list of tuples)
# dictionary - ages of students
students_ages = dict([('Amanda', 27), ('Teresa', 38), ('Paula', 17), ('Mario', 40)])

print(students_ages)
# {'Amanda': 27, 'Teresa': 38, 'Paula': 17, 'Mario': 40}
```

dictionaries_dict_constructor_3.py hosted with ❤ by GitHub view raw

Lastly, we can create a dictionary using two lists. First, we have to build an **iterator of tuples** using **zip(*iterables)** function. Then, we employ the **dict([iterable, **kwarg])** function to construct the dictionary, as we did previously.

```python
# create a dictionary using two list
students = ['Amanda', 'Teresa', 'Paula', 'Mario']
ages = [27, 38, 17, 40]

# zip method --> iterator of tuples --> dict method --> dictionary
students_ages = dict(zip(students, ages))

print(students_ages)
# {'Amanda': 27, 'Teresa': 38, 'Paula': 17, 'Mario': 40}
```

dictionaries_dict_constructor_4.py hosted with ❤ by GitHub view raw

3. Access values in a dictionary

To access dictionary **values**, we cannot use a numeric index (as we do with lists or tuples), since the dictionaries are **unordered** containers. Instead, we enclose the **key** using square brackets([]). If we try to access a value using an undefined **key**, a **KeyError** is raised.

To avoid getting an exception with undefined keys, we can use the method **dict.get(key[, default])**. This method returns the **value** for **key** if **key** is in the **dictionary**, else returns default. If default is not provided, it returns **None** (but never raises an exception).

```python
# access population
population['Munich']
# 1471508

# access a value using a numeric index
population[1]
# KeyError

# access population of Stuttgart
population['Stuttgart']
# KeyError

# access population of Stuttgart using .get() method without default value
print(population.get('Munich'))
# 1471508

# access population of Stuttgart using .get() method without default value
print(population.get('Stuttgart'))
# None

# access population of Stuttgart using .get() method with default value
print(population.get('Stuttgart', 'Not found'))
# Not found
```

dictionaries_access.py hosted with ❤ by GitHub view raw

4. Insert elements in a dictionary

To insert an element in a dictionary, we can use square brackets as follows:

```python
# add pillow to the products dictionary
products['pillow'] = 10

print(products)
# {'table': 120, 'chair': 40, 'lamp': 14, 'bed': 250, 'mattress': 100, 'pillow': 10}
```

dictionaries_insert_1.py hosted with ❤ by GitHub view raw

To insert multiple items at once, we can use the method **dict.update([other])**. This method updates the dictionary with the **key/value pairs** from other, overwriting existing **keys**.

```python
# add shelf and sofa to the products dictionary using another dictionary object
products.update({'shelf': 70, 'sofa': 300})

print(products)
#{'table': 120, 'chair': 40, 'lamp': 14, 'bed': 250, 'mattress': 100, 'pillow': 10, 'shelf': 70,

# add three new items to the grades dictionary using keyword arguments
grades.update(Violeta=5.5, Marco=6.5, Paola=8)

print(grades)
# {'Alba': 9.5, 'Eduardo': 10, 'Normando': 3.5, 'Helena': 6.5, 'Claudia': 7.5, 'Violeta': 5.5, 'N

# add two cities to the population dictionary using a list of tuples
population.update([('Stuttgart', 632743), ('Dusseldorf', 617280)])

print(population)
# {'Berlin': 3748148, 'Hamburg': 1822445, 'Munich': 1471508, 'Cologne': 1085664, 'Frankfurt': 75
```

dictionary_insert_2.py hosted with ❤ by GitHub view raw

As shown above, the **.update()** method accepts as an argument not only another **dictionary**, but also a **list of tuples** or **keyword arguments**. This method modifies the dictionary in-place, returning **None**.

5. Change elements in a dictionary

We can change the **value** of an item by accessing the **key** using square brackets ([]). To modify multiple values at once, we can use the **.update()**

method, since this function overwrites existing **keys**.

Subsequently, we increase the price of a sofa 100 units, and we modify the grades of two students.

```python
# increase the price of a sofa 100 units
print(products)
# {'table': 120, 'chair': 40, 'lamp': 14, 'bed': 250, 'mattress': 100, 'pillow': 10, 'shelf': 70,

products['sofa'] = 400

print(products)
#{'table': 120, 'chair': 40, 'lamp': 14, 'bed': 250, 'mattress': 100, 'pillow': 10, 'shelf': 70,

# modify the grades of two students
print(grades)
# {'Alba': 9.5, 'Eduardo': 10, 'Normando': 3.5, 'Helena': 6.5, 'Claudia': 7.5, 'Violeta': 5.5, 'N

grades.update({'Normando': 2.5, 'Violeta': 6})

print(grades)
#{'Alba': 9.5, 'Eduardo': 10, 'Normando': 2.5, 'Helena': 6.5, 'Claudia': 7.5, 'Violeta': 6, 'Mar
```

dictionaries_change.py hosted with ❤ by GitHub view raw

6. Remove elements in a dictionary

To remove an element in a dictionary, we can use either the **del dict[key]** keyword or the **dict.pop(key[, default])** method.

The **del dict[key]** keyword removes the given element from the dictionary, raising a **KeyError** if **key** does not exists.

```
print(population)
#{'Berlin': 3748148, 'Hamburg': 1822445, 'Munich': 1471508, 'Cologne': 1085664, 'Frankfurt': 753
# 'Dusseldorf': 617280}

# key does not exists
del population['Ingolstadt']
# KeyError

# key exists
# the element dusseldorf is removed
del population['Dusseldorf']

print(population)
#{'Berlin': 3748148, 'Hamburg': 1822445, 'Munich': 1471508, 'Cologne': 1085664, 'Frankfurt': 753
```

dictionaries_remove.py hosted with ❤ by GitHub view raw

If **key** exists in the dictionary, the **dict.pop(key[, default])** method removes the **item** with the given **key** from the dictionary and returns its **value**. On the contrary, if **key** does not exist in the dictionary, the method returns the **default** value. If no **default** value is provided and **key** does not exist, the **.pop()** method will raise an **exception** (**KeyError**).

```
print(population)
#{'Berlin': 3748148, 'Hamburg': 1822445, 'Munich': 1471508, 'Cologne': 1085664, 'Frankfurt': 753

# key exists - the item is removed and the value returned
population.pop('Stuttgart')
# 632743 - returned value

print(population)
#{'Berlin': 3748148, 'Hamburg': 1822445, 'Munich': 1471508, 'Cologne': 1085664, 'Frankfurt': 753

# key does not exists but default value is provided
population.pop('Ingolstadt', 'Value not found')
# 'Value not found' - returned value

# key does not exists and default value is NOT provided
population.pop('Garching')
# KeyError
```

dictionaries_remove2.py hosted with ❤ by GitHub view raw

7. Check if a key exists

To check whether a **key** exists in a **dictionary**, we have to use a **membership operator**. Membership operators are used to test whether a value is found in a sequence (e.g. strings, lists, tuples, sets, or dictionaries). There are two membership operators, as explained below.

- **in** → Evaluates to true if the object on the left side **is** included in the object on the right side.
- **not in** → Evaluates to true if the object on the left side **is not** included in the object on the right side.

```python
# population dictionary
print(population)
# {'Berlin': 3748148, 'Hamburg': 1822445, 'Munich': 1471508, 'Cologne': 1085664, 'Frankfurt': 75

# membership operators
# check if the key 'Ingolstadt' exists in the dictionary population
'Ingolstadt' in population
# False

# check if the key 'Munich' exists in the dictionary population
'Munich' in population
# True

# check if the key 'Ingolstadt' does not exist in the dictionary population
'Ingolstadt' not in population
# True

# check if the key 'Munich' does not exist in the dictionary population
'Munich' not in population
# False
```

dictionaries_membership.py hosted with ❤ by GitHub view raw

As shown above, membership operators (**in** and **not in**) can be used to check whether a key exists in a dictionary, but they can also be used with other sequences in the following manner.

```python
# membership operators - in / not in

# strings
'a' in 'Amanda'
# True

# lists
5 in [1, 2, 3, 4]
# False

# tuples
3 not in (1, 2)
# True

# sets
'Valencia' not in {'Barcelona', 'Valencia', 'Madrid'}
# False
```

dictionaries_membership_2.py hosted with ❤ by GitHub view raw

8. Copy a dictionary

To copy a dictionary, we can simply use the **dict.copy()** method. This method returns a **shallow copy** of the dictionary. We have to be careful with **shallow copies**, since if your dictionary contains another **container-objects** like lists, tuples, or sets, they will be referenced again and not duplicated.

```python
# dictionary with students heights
students = {'Marco': 173, 'Luis': 184, 'Andrea': 168}

# create a shallow copy
students_2 = students.copy()

# modify the height of luis in the shallow copy
students_2['Luis'] = 180

# the modification in students_2 is not observed in students since 180 is an int
print(students)
# {'Marco': 173, 'Luis': 184, 'Andrea': 168}

print(students_2)
# {'Marco': 173, 'Luis': 180, 'Andrea': 168}


# dictionary with students heights and weights
students_weights = {'Marco': [173, 70], 'Luis': [184, 80], 'Andrea': [168, 57]}

# create a shallow copy
students_weights_2 = students_weights.copy()

# modify the height of luis in the shallow copy
students_weights_2['Luis'][0] = 180

# the modification in students_weights_2 is observed in students_weights
# since the list containing the weight and height is referenced and not duplicated
print(students_weights)
# {'Marco': [173, 70], 'Luis': [180, 80], 'Andrea': [168, 57]}

# solution --> create a deepcopy of the dictionary
```

dictionaries_copy.py hosted with ❤ by GitHub   view raw

To avoid this problem, we can create a **deep copy** using **copy.deepcopy(x)** function (defined in the **copy** module) as follows:

```python
import copy

# dictionary with students heights and weights
students_weights = {'Marco': [173, 70], 'Luis': [184, 80], 'Andrea': [168, 57]}

# create a deep copy
students_weights_2 = copy.deepcopy(students_weights)

# modify the height of luis in the shallow copy
students_weights_2['Luis'][0] = 180

# the modification in students_weights_2 is NOT observed in students_weights
# since we are working with a deep copy

print(students_weights)
# {'Marco': [173, 70], 'Luis': [184, 80], 'Andrea': [168, 57]}

print(students_weights_2)
# {'Marco': [173, 70], 'Luis': [180, 80], 'Andrea': [168, 57]}
```

dictionaries_copy_2.py hosted with ❤ by GitHub view raw

The difference between **shallow copies** and **deep copies** is only relevant when the dictionary contains other objects like lists, since those objects will be referenced instead of duplicated (**shallow copy**). To create a fully independent clone of the original dictionary, we have to make a **deep copy**.

If you want to know more about how to copy a dictionary, you can read the following article where the differences between **shallow copies** and **deep copies** are explained in detail.

Python : How to copy a dictionary | Shallow Copy vs Deep Copy

In this article we will discuss how to create a shallow and deep copy of dictionary in Python. Python's dictionary...

thispointer.com

It is important to bear in mind that the = **operator** does not make a copy of the dictionary. It is just another name to refer to the same dictionary, meaning any modification to the new dictionary is reflected in the original one.

```python
# dictionary with calories in fruits
fruits = {'Orange': 50, 'Apple': 65, 'Avocado': 160, 'Pear': 75}

# copy the dictionary using = operators
fruits_2 = fruits

# modify fruits_2 (delete one item)
fruits_2.pop('Orange')

# the modification is reflected in fruits
print(fruits)
# {'Apple': 65, 'Avocado': 160, 'Pear': 75}
```

dictionary_copy_3.py hosted with ❤ by GitHub view raw

9. Determine the length of the dictionary

To determine how many **key-value** pairs the dictionary contains, we can use the **len()** function. This function returns the number of items of an object. The input of the function can be a dictionary, but also another type of sequence such as a string, list, tuple, or set.

```python
print(population)
# {'Berlin': 3748148, 'Hamburg': 1822445, 'Munich': 1471508, 'Cologne': 1085664, 'Frankfurt': 753

len(population)
# 5
```

dictionaries_len.py hosted with ❤ by GitHub view raw

10. Loop through a dictionary

Iterating through keys

To iterate over the **keys**, we can use the dictionary directly in a **for** loop as follows:

```python
# iterate through keys
for city in population:
    print(city)
# Berlin
# Hamburg
# Munich
# Cologne
# Frankfurt
```

dictionaries_iteration_keys.py hosted with ❤ by GitHub view raw

Alternatively, we can use the **dict.keys()** method. This method returns a view object, containing the **keys** of the dictionary.

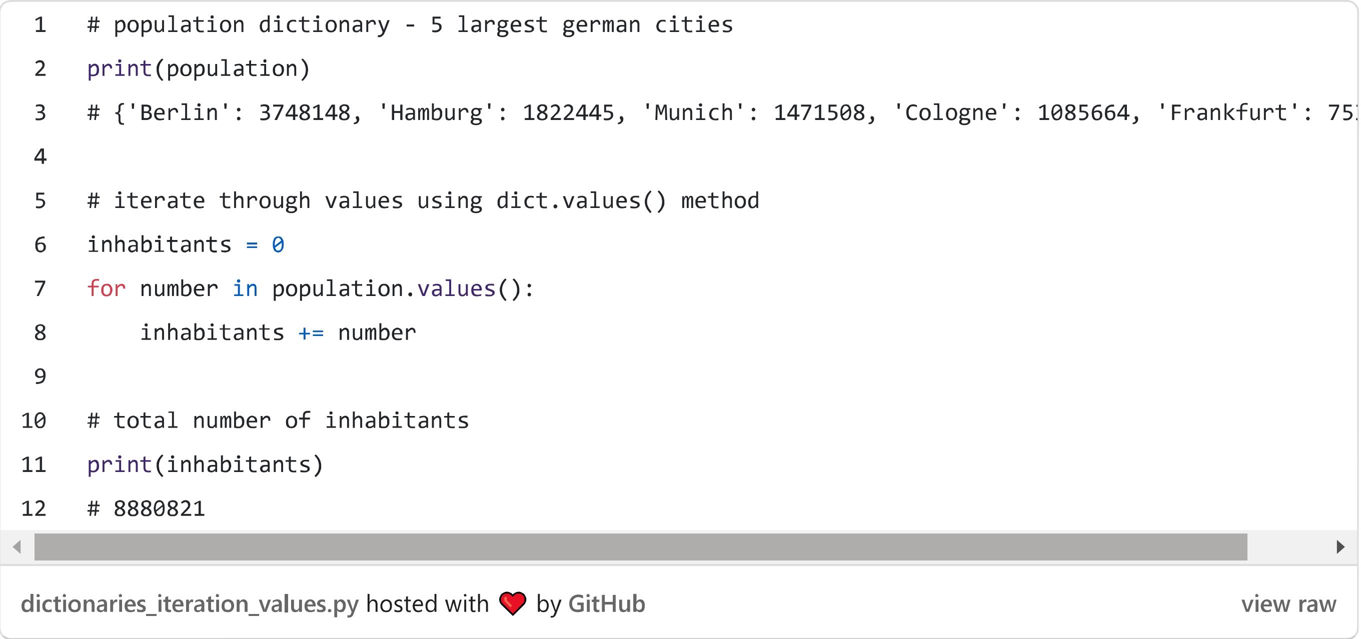
```python
# iterate through keys using dict.keys() method
for city in population.keys():
    print(city)
# Berlin
# Hamburg
# Munich
# Cologne
# Frankfurt
```

dictionaries_iteration_keys_2.py hosted with ❤ by GitHub view raw

Iterating through values

If you just need to work with the **values** of a dictionary, then you can use the **dict.values()** method in a **for** loop. This method returns a view object that contains the **values** of the dictionary.

We can compute how many people live in the 5 largest German cities using **dict.values()** method as follows:



```python
# population dictionary - 5 largest german cities
print(population)
# {'Berlin': 3748148, 'Hamburg': 1822445, 'Munich': 1471508, 'Cologne': 1085664, 'Frankfurt': 75

# iterate through values using dict.values() method
inhabitants = 0
for number in population.values():
    inhabitants += number

# total number of inhabitants
print(inhabitants)
# 8880821
```

dictionaries_iteration_values.py hosted with ❤ by GitHub view raw

As we can observe, almost 9 million people live in the 5 largest German cities.

Iterating through items

When you're working with dictionaries, it's likely that you need to use the **keys** and the **values.** To loop through both, you can use the **dict.items()** method. This method returns a view object, containing **key-value** pairs as a list of tuples.

We can determine the student with the lowest test score using the **dict.items()** method in combination with a **for loop** as follows:

```python
# students grades dictionary
print(grades)
# {'Alba': 9.5, 'Eduardo': 10, 'Normando': 2.5, 'Helena': 6.5, 'Claudia': 7.5, 'Violeta': 6, 'Ma

# dict.items() - dictionary view object containing key-value pairs as a list of tuples
grades.items()
# dict_items([('Alba', 9.5), ('Eduardo', 10), ('Normando', 2.5), ('Helena', 6.5), ('Claudia', 7.
#              ('Violeta', 6), ('Marco', 6.5), ('Paola', 8)])

# determine student with the lowest test score
min_grade = 10
min_student = ''
for student, grade in grades.items():
    if grade < min_grade:
        min_student = student
        min_grade = grade

print(min_student)
# Normando
```

dictionaries_iteration_items.py hosted with ❤ by GitHub view raw

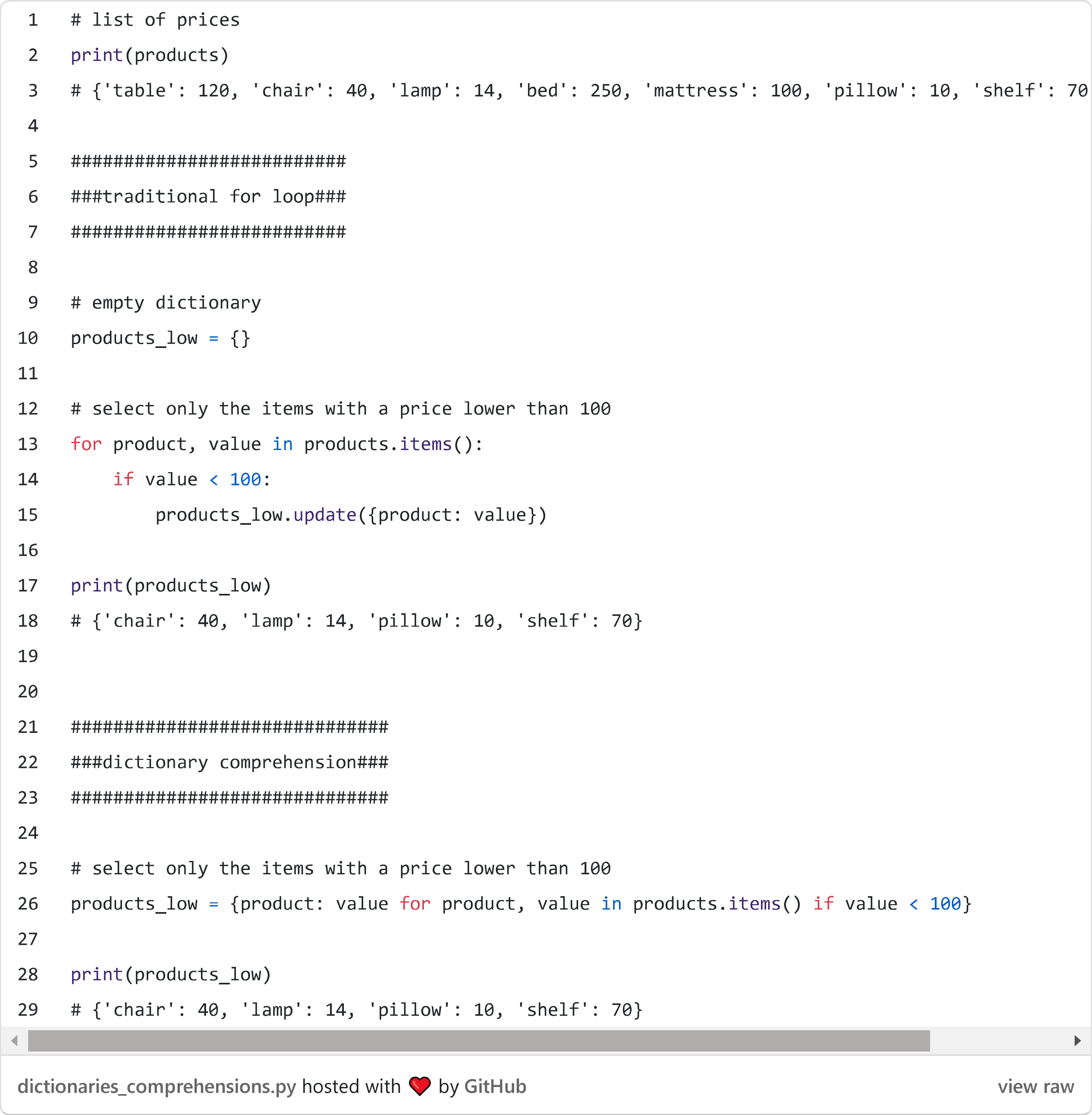
As shown above, Normando is the student with the lowest test score (2.5).

11. Dictionary comprehensions

Python **for-loops** are very handy in dealing with repetitive programming tasks; however, there is another alternative to achieve the same results in a more efficient way: **dictionary comprehensions.**

Dictionary comprehensions allow the creation of a dictionary using an elegant and simple syntax: **{key: value for vars in iterable}.** In addition, they are faster than traditional **for-loops**.

We can filter the products with a price lower than 100 euros using both a traditional **for-loop** and a **dictionary comprehension**.



```python
# list of prices
print(products)
# {'table': 120, 'chair': 40, 'lamp': 14, 'bed': 250, 'mattress': 100, 'pillow': 10, 'shelf': 70

##########################
###traditional for loop###
##########################

# empty dictionary
products_low = {}

# select only the items with a price lower than 100
for product, value in products.items():
    if value < 100:
        products_low.update({product: value})

print(products_low)
# {'chair': 40, 'lamp': 14, 'pillow': 10, 'shelf': 70}


##############################
###dictionary comprehension###
##############################

# select only the items with a price lower than 100
products_low = {product: value for product, value in products.items() if value < 100}

print(products_low)
# {'chair': 40, 'lamp': 14, 'pillow': 10, 'shelf': 70}
```

dictionaries_comprehensions.py hosted with ❤ by GitHub

view raw

As we can observe, **dictionary comprehensions** provide the same results as traditional **for-loops** in a more elegant way.

12. Nested dictionaries

Nested dictionaries are dictionaries that contain other dictionaries. We can create a **nested dictionary** in the same way we create a normal dictionary using curly brackets ({}).

The following **nested dictionary** contains information about 5 famous works of art. As we can observe, the **values** of the dictionary are other dictionaries as well.

```python
# nested dictionary containing information about famous works of art
works_of_art = {'The_Starry_Night': {'author': 'Van Gogh', 'year': 1889, 'style': 'post-impressio
                'The_Birth_of_Venus': {'author': 'Sandro Botticelli', 'year': 1480, 'style': 'ren
                'Guernica': {'author': 'Pablo Picasso', 'year': 1937, 'style': 'cubist'},
                'American_Gothic': {'author': 'Grant Wood', 'year': 1930, 'style': 'regionalism'}
                'The_Kiss': {'author': 'Gustav Klimt', 'year': 1908, 'style': 'art nouveau'}}
```

dictionaries_nested.py hosted with ❤ by GitHub view raw

We can also create the prior **nested dictionary** using the **dict()** constructor, passing the **key: value** pairs as **keyword arguments.**

```python
# nested dictionary with dict() constructor
works_of_art = dict(The_Starry_Night={'author': 'Van Gogh', 'year': 1889, 'style': 'post-impressi
                    The_Birth_of_Venus={'author': 'Sandro Botticelli', 'year': 1480, 'style': 're
                    Guernica={'author': 'Pablo Picasso', 'year': 1937, 'style': 'cubist'},
                    American_Gothic={'author': 'Grant Wood', 'year': 1930, 'style': 'regionalism'
                    The_Kiss={'author': 'Gustav Klimt', 'year': 1908, 'style': 'art nouveau'})
```

dictionaries_nested_dict.py hosted with ❤ by GitHub view raw

To access elements in a nested dictionary, we specify the keys using multiple square brackets ([]).

```python
# access elements in a nested dictionary
works_of_art['Guernica']['author']
# 'Pablo Picasso'

works_of_art['American_Gothic']['style']
# 'regionalism'
```

dictionaries_nested_access.py hosted with ❤ by GitHub view raw

If you want to know more about **nested dictionaries**, you can read the following article where, how to work with **nested dictionaries** (e.g. update items, change elements, and loop though) is explained in detail.

Python Nested Dictionary - Learn By Example

A dictionary can contain another dictionary, which in turn can contain dictionaries themselves, and so on to arbitrary...

www.learnbyexample.org

13. Alternative containers : OrderedDict, defaultdict, and Counter

The **collections** module provides alternative container datatypes to built-in Python containers. Three dictionary subclasses contained in the **collections** module that are pretty handy when working with Python are: (1)**OrderedDict**, (2)**defaultdict**, and (3)**Counter**.

OrderedDict

OrderedDict consists of a dictionary that remembers the order in which its contents are added. In Python 3.6+ dictionaries are also **insertion ordered,** meaning they remember the order of items inserted. However, to guarantee element order across other Python versions, we have to use **OrderedDict** containers.

```python
import collections

# create an OrderedDict of chemical elements
dictionary = collections.OrderedDict({'hydrogen': 1, 'helium': 2, 'carbon': 6, 'oxygen': 8})

# type OrderedDict
print(type(dictionary))
# <class 'collections.OrderedDict'>

# dictionary keys --> .keys() method
print(dictionary.keys())
# odict_keys(['hydrogen', 'helium', 'carbon', 'oxygen'])

# dictionary values --> .values() method
print(dictionary.values())
# odict_values([1, 2, 6, 8])

# insert a new element
dictionary['nitrogen'] = 7

# nitrogen last position since it is the last element added
print(dictionary)
# OrderedDict([('hydrogen', 1), ('helium', 2), ('carbon', 6), ('oxygen', 8), ('nitrogen', 7)])
```

dictionaries_ordereddict.py hosted with ❤ by GitHub view raw

As shown above, **OrderedDict** accepts dictionary methods and functions. Moreover, elements can be inserted, changed, or deleted in the same way as with normal dictionaries.

defaultdict

Defaultdicts are a dictionary subclass that assign a **default value** when a key is missing (it has not been set yet). They never raise a **KeyError**, if we try to access an item that is not available in the dictionary, instead a new entry is created.

Defaultdicts take a function as an argument, and initialize the missing key with the value returned by the function. In the example below, the keys are initialized with different values, depending on the function employed as first argument.

```python
import collections
import numpy as np

# missing key initialized with a 0
default_1 = collections.defaultdict(int)

default_1['missing_entry']
print(default_1)
# defaultdict(<class 'int'>, {'missing_entry': 0})

# missing key initialized with an empty list
default_2 = collections.defaultdict(list, {'a': 1, 'b': 2})

default_2['missing_entry']
print(default_2)
# defaultdict(<class 'list'>, {'a': 1, 'b': 2, 'missing_entry': []})

# missing key initialized with a string
default_3 = collections.defaultdict(lambda : 'Not given', a=1, b=2)

default_3['missing_entry']
print(default_3)
# defaultdict(<function <lambda> at 0x000001DEF6ADF730>, {'a': 1, 'b': 2, 'missing_entry': 'Not

# missing key initialized with a numpy array
default_4 = collections.defaultdict(lambda: np.zeros(2))

default_4['missing_entry']
print(default_4)
# defaultdict(<function <lambda> at 0x000001DEF6ADF950>, {'missing_entry': array([0., 0.])})
```

dictionaries_defaultdict.py hosted with ❤ by GitHub view raw

As we can observe, we can pass a **dictionary** or **keywords** as second argument (optional) to initialize the **defaultdict** container.

Counter

A **Counter** is a dictionary subclass for counting hastable objects. The function returns a Counter object, where elements are stored as **keys** and their counts are stored as **values**. Using this function, we can easily count the elements of a list, as shown below.

```python
import collections

# list containing letters
letters = ['a', 'a', 'c', 'a', 'a', 'b', 'c', 'a']

# count letters
counter = collections.Counter(letters)

print(counter)
# Counter({'a': 5, 'c': 2, 'b': 1})

# 2 most common elements
counter.most_common(2)
# [('a', 5), ('c', 2)]
```

dictionaries_counts.py hosted with ❤ by GitHub view raw

As shown above, we can easily obtain the most frequent elements with the **.most_common([n])** method. This method returns a list of the n most common elements and their counts.

14. Create a Pandas DataFrame from a dictionary.

A Pandas **DataFrame** is a two-dimensional tabular data where each **row** represents an observation and each **column** a variable. A Pandas DataFrame can be created using the **pandas.DataFrame** constructor. This function accepts as input various python containers (e.g. lists, dictionaries, or numpy arrays). However, in this article, we explain only the ways to create a DataFrame that involve the use of dictionaries.

Create a DataFrame from a dictionary

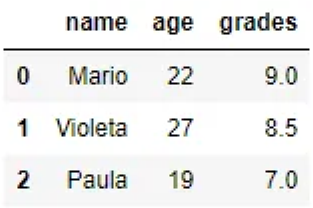
We can create a **DataFrame** from a **dictionary**, where the **keys** represent column names, and the **values** represent column data in the following manner:

```
import pandas as pd

# create a Pandas DataFrame from a dictionary - keys (column name) - value (column data)
df = pd.DataFrame({'name': ['Mario', 'Violeta', 'Paula'],
                   'age': [22, 27, 19],
                   'grades': [9, 8.5, 7]})

df
```

dictionaries_dataframe.py hosted with ❤ by GitHub view raw



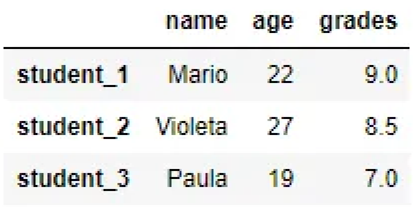
	name	age	grades
0	Mario	22	9.0
1	Violeta	27	8.5
2	Paula	19	7.0

As we can observe, the default index is just the row number (an integer index beginning at 0). We can modify these indexes by passing the index list to the DataFrame constructor.

```
# create a Pandas DataFrame from a dictionary - keys (column name) - value (column data) - with cu
df_index = pd.DataFrame({'name': ['Mario', 'Violeta', 'Paula'],
                   'age': [22, 27, 19],
                   'grades': [9, 8.5, 7]}, index=['student_1', 'student_2', 'student_3'])

df_index
```

dictionaries_dataframe_indexes.py hosted with ❤ by GitHub view raw



	name	age	grades
student_1	Mario	22	9.0
student_2	Violeta	27	8.5
student_3	Paula	19	7.0

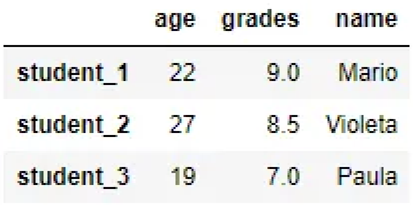
Create a DataFrame from a list of dictionaries

A **list of dictionaries** can also be used to create a **DataFrame**, where the **keys** represent column names. As before, we can change indexes by passing the index list to the **DataFrame** function.

```python
# create a Pandas DataFrame from a list of dictionaries - keys(column name) - with custom indexes
df_2 = pd.DataFrame([{'name': 'Mario', 'age': 22, 'grades':9},
                     {'name': 'Violeta', 'age': 27, 'grades':8.5},
                     {'name': 'Paula', 'age': 19, 'grades':7}], index=['student_1', 'student_2',

df_2
```

dictionaries_dataframe_list.py hosted with ❤ by GitHub   view raw



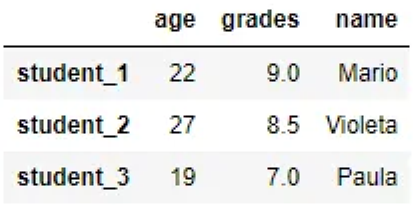
	age	grades	name
student_1	22	9.0	Mario
student_2	27	8.5	Violeta
student_3	19	7.0	Paula

15. Functions in Pandas that use dictionaries

There are several functions in Pandas that use dictionaries as input values, for example, **pandas.DataFrame.rename** and **pandas.DataFrame.replace.**

pandas.DataFrame.rename

This function returns a DataFrame with renamed axis labels. We can use a **dictionary** as input where **keys** refer to the old names and **values** to the new ones. Labels not contained in the dictionary remain unchanged.

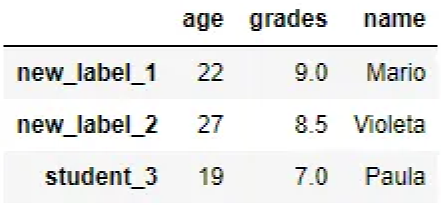


	age	grades	name
student_1	22	9.0	Mario
student_2	27	8.5	Violeta
student_3	19	7.0	Paula

```python
# change index labels in df_2
df_2.rename(index={'student_1': 'new_label_1', 'student_2': 'new_label_2'}, inplace=True)

df_2
```

dictionaries_dataframe_rename.py hosted with ❤ by GitHub view raw

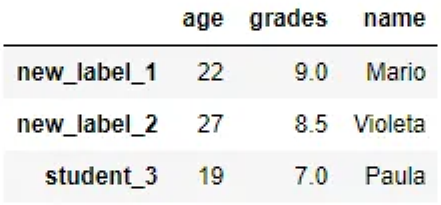


	age	grades	name
new_label_1	22	9.0	Mario
new_label_2	27	8.5	Violeta
student_3	19	7.0	Paula

As shown above, we can change **index labels**, providing a **dictionary** to the index parameter. Alternatively, we can modify column names providing the **dictionary** to the **column** parameter.

pandas.DataFrame.replace

This function replaces values of the **DataFrame** with other values dynamically. We can use a dictionary with the replace function to modify the **DataFrame** where **keys** represent existing entries, and **values** replacement entries.

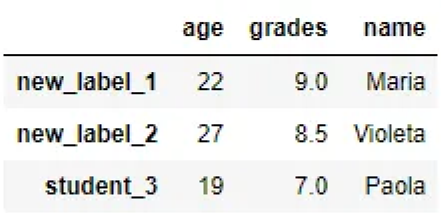


	age	grades	name
new_label_1	22	9.0	Mario
new_label_2	27	8.5	Violeta
student_3	19	7.0	Paula

```python
# replace Mario by Maria and Paula by Paola
df_2.replace({'Mario': 'Maria', 'Paula': 'Paola'}, inplace=True)

df_2
```

dictionaries_dataframe_replace.py hosted with ❤ by GitHub view raw



	age	grades	name
new_label_1	22	9.0	Maria
new_label_2	27	8.5	Violeta
student_3	19	7.0	Paola

Article finished! 🍀 As you can see, dictionaries are a really useful tool in Python. I hope this article serves you as a guideline for taking full advantage of them when coding in Python. 💪

Python Programming Python Dictionaries Pandas Data Science

Written by Amanda Iglesias Moreno

Follow

2.3K Followers · Writer for Towards Data Science

Data Scientist at Statista — Based in Hamburg 📍

More from Amanda Iglesias Moreno and Towards Data Science