Club Informatique
< ARTS & MÉTIERS />

Dictionaries in Python and Introduction to Pandas.

Dictionary

Dictionaries are used to store data values in key:value pairs.

A dictionary is a collection which is ordered*, changeable and do not allow duplicates.

Dictionaries are written with curly brackets, and have keys and values:

Example

Create and print a dictionary:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(thisdict)
```

Output :

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
```

Dictionary Items

Dictionary items are ordered, changeable, and does not allow duplicates.

Dictionary items are presented in key:value pairs, and can be referred to by using the key name.

Example

Print the "brand" value of the dictionary:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(thisdict["brand"])
```

Output : Ford

Ordered or Unordered?

When we say that dictionaries are ordered, it means that the items have a defined order, and that order will not change.

Unordered means that the items does not have a defined order, you cannot refer to an item by using an index.

Changeable

Dictionaries are changeable, meaning that we can change, add or remove items after the dictionary has been created.

Duplicates Not Allowed

Dictionaries cannot have two items with the same key:

Example

Duplicate values will overwrite existing values:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964,
  "year": 2020
}
print(thisdict)
```

Output

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 2020}
```

Dictionary Length

To determine how many items a dictionary has, use the `len()` function:

Example

Print the number of items in the dictionary:

```
print(len(thisdict))
```

This will print the value of 3

Dictionary Items - Data Types

The values in dictionary items can be of any data type:

Example

String, int, boolean, and list data types:

```
thisdict = {
  "brand": "Ford",
  "electric": False,
  "year": 1964,
  "colors": ["red", "white", "blue"]
}
```

Accessing Items

You can access the items of a dictionary by referring to its key name, inside square brackets:

Example

Get the value of the "model" key:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
x = thisdict["model"]
#
```

Output = > Mustang

There is also a method called `get()` that will give you the same result:

Example

Get the value of the "model" key:

```
x = thisdict.get("model")
```

Output = > Mustang

Get Keys

The `keys()` method will return a list of all the keys in the dictionary.

Example

Get a list of the keys:

```
x = thisdict.keys()
```

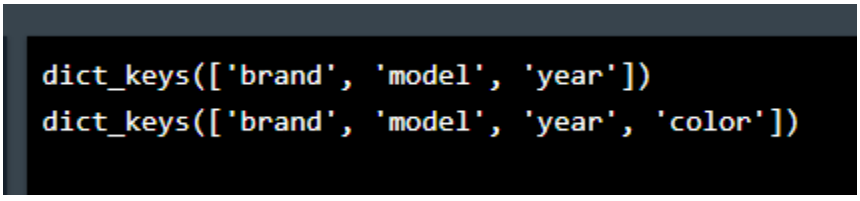
The list of the keys is a *view* of the dictionary, meaning that any changes done to the dictionary will be reflected in the keys list.

Example

Add a new item to the original dictionary, and see that the keys list gets updated as well:

```
car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}

x = car.keys()

print(x) #before the change

car["color"] = "white"

print(x) #after the change
```

Output :



Get Values

The `values()` method will return a list of all the values in the dictionary.

Example

Get a list of the values:

```
x = thisdict.values()
```

The list of the values is a *view* of the dictionary, meaning that any changes done to the dictionary will be reflected in the values list.

Example

Make a change in the original dictionary, and see that the values list gets updated as well:

```
car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}

x = car.values()

print(x) #before the change

car["year"] = 2020

print(x) #after the change
```

```
dict_values(['Ford', 'Mustang', 1964])
dict_values(['Ford', 'Mustang', 2020])
```

Example 2

Add a new item to the original dictionary, and see that the values list gets updated as well:

```
car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}

x = car.values()

print(x) #before the change

car["color"] = "red"

print(x) #after the change
```

Output

```
dict_values(['Ford', 'Mustang', 1964])
dict_values(['Ford', 'Mustang', 1964, 'red'])
```

Get Items

The `items()` method will return each item in a dictionary, as tuples in a list.

Example

Get a list of the key:value pairs

```
x = thisdict.items()
```

The returned list is a *view* of the items of the dictionary, meaning that any changes done to the dictionary will be reflected in the items list.

Example

Add a new item to the original dictionary, and see that the items list gets updated as well:

```
car = {
"brand": "Ford",
"model": "Mustang",
"year": 1964
}

x = car.items()

print(x) #before the change

car["color"] = "red"

print(x) #after the change
```

```
dict_items([('brand', 'Ford'), ('model', 'Mustang'), ('year', 1964)])
dict_items([('brand', 'Ford'), ('model', 'Mustang'), ('year', 1964), ('color', 'red')])
```

Check if Key Exists

To determine if a specified key is present in a dictionary use the `in` keyword:

Example

Check if "model" is present in the dictionary:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
if "model" in thisdict:
  print("Yes, 'model' is one of the keys in the thisdict dictionary")
```

In the same way, we can determine if a value exist.

Change Values

You can change the value of a specific item by referring to its key name:

Example

Change the "year" to 2018:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict["year"] = 2018
```

Update Dictionary

The `update()` method will update the dictionary with the items from a given argument. If the item does not exist, the item will be added.

The argument must be a dictionary, or an iterable object with key:value pairs.

Example

Update the "year" of the car by using the `update()` method:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.update({"year": 2020})

thisdict.update({"color": "red"})
```

Removing Items

There are several methods to remove items from a dictionary:

Example

The `pop()` method removes the item with the specified key name:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.pop("model")
print(thisdict)
```

```
{'brand': 'Ford', 'year': 1964}
```

Example

The `popitem()` method removes the last inserted item (in versions before 3.7, a random item is removed instead):

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.popitem()
print(thisdict)
```

```
{'brand': 'Ford', 'model': 'Mustang'}
```

Example

The `del` keyword removes the item with the specified key name:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
del thisdict["model"]
print(thisdict)
```

```
{'brand': 'Ford', 'year': 1964}
```

Example

The `del` keyword can also delete the dictionary completely:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
del thisdict
print(thisdict) #this will cause an error because "thisdict" no longer
exists.
```

```
Traceback (most recent call last):
  File "demo_dictionary_del3.py", line 7, in <module>
    print(thisdict) #this will cause an error because "thisdict" no longer exists.
NameError: name 'thisdict' is not defined
```

Example

The `clear()` method empties the dictionary:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
```

```
  "year": 1964
}
thisdict.clear()
print(thisdict) #output => {}
```

Loop Through a Dictionary

You can loop through a dictionary by using a `for` loop.

When looping through a dictionary, the return value are the *keys* of the dictionary, but there are methods to return the *values* as well.

Example

Print all key names in the dictionary, one by one:

```
for x in thisdict:
  print(x)
```

```
brand
model
year
```

Example

Print all *values* in the dictionary, one by one:

```
for x in thisdict:
  print(thisdict[x])
```

```
Ford
Mustang
1964
```

Example

You can also use the `values()` method to return values of a dictionary:

```
for x in thisdict.values():
  print(x)
```

```
Ford
Mustang
1964
```

Example

You can use the `keys()` method to return the keys of a dictionary:

```
for x in thisdict.keys():
  print(x)
```

```
brand
model
year
```

Example

Loop through both *keys* and *values*, by using the `items()` method:

```
for x, y in thisdict.items():
  print(x, y)
```

```
brand Ford
model Mustang
year 1964
```

Copy a Dictionary

You cannot copy a dictionary simply by typing `dict2 = dict1`, because: `dict2` will only be a *reference* to `dict1`, and changes made in `dict1` will automatically also be made in `dict2`.

There are ways to make a copy, one way is to use the built-in Dictionary method `copy()`.

Example

Make a copy of a dictionary with the `copy()` method:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
mydict = thisdict.copy()
print(mydict)
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
```

Another way to make a copy is to use the built-in function `dict()`.

Example

Make a copy of a dictionary with the `dict()` function:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
mydict = dict(thisdict)
print(mydict)
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
```

Nested Dictionaries

A dictionary can contain dictionaries, this is called nested dictionaries.

Example

Create a dictionary that contain three dictionaries:

```
myfamily = {
  "child1" : {
    "name" : "Emil",
    "year" : 2004
  },
  "child2" : {
    "name" : "Tobias",
    "year" : 2007
  },
  "child3" : {
    "name" : "Linus",
    "year" : 2011
  }
}
```

Or, if you want to add three dictionaries into a new dictionary:

Example

Create three dictionaries, then create one dictionary that will contain the other three dictionaries:

```
child1 = {
  "name" : "Emil",
```

```
  "year" : 2004
}
child2 = {
  "name" : "Tobias",
  "year" : 2007
}
child3 = {
  "name" : "Linus",
  "year" : 2011
}

myfamily = {
  "child1" : child1,
  "child2" : child2,
  "child3" : child3
}
```

RECAP

Dictionary Methods

Python has a set of built-in methods that you can use on dictionaries.

Club Informatique
< ARTS & MÉTIERS />

Method	Description
clear()	Removes all the elements from the dictionary
copy()	Returns a copy of the dictionary
fromkeys()	Returns a dictionary with the specified keys and value
get()	Returns the value of the specified key
items()	Returns a list containing a tuple for each key value pair
keys()	Returns a list containing the dictionary's keys
pop()	Removes the element with the specified key
popitem()	Removes the last inserted key-value pair
setdefault()	Returns the value of the specified key. If the key does not exist: insert the key, wi
update()	Updates the dictionary with the specified key-value pairs
values()	Returns a list of all the values in the dictionary

Pandas

What is Pandas?

Pandas is a Python library used for working with data sets.

It has functions for analyzing, cleaning, exploring, and manipulating data.

The name "Pandas" has a reference to both "Panel Data", and "Python Data Analysis" and was created by Wes McKinney in 2008.

Why Use Pandas?

Pandas allows us to analyze big data and make conclusions based on statistical theories.

Pandas can clean messy data sets, and make them readable and relevant.

Relevant data is very important in data science.

Data Science: is a branch of computer science where we study how to store, use and analyze data for deriving information from it.

What Can Pandas Do?

Pandas gives you answers about the data. Like:

- Is there a correlation between two or more columns?
- What is average value?
- Max value?
- Min value?

Pandas are also able to delete rows that are not relevant, or contains wrong values, like empty or NULL values. This is called *cleaning* the data.

Where is the Pandas Codebase?

The source code for Pandas is located at this github repository https://github.com/pandas-dev/pandas

Installation of Pandas

If you have Python and PIP already installed on a system, then installation of Pandas is very easy.

Install it using this command:

```
C:\Users\AmineGatou >pip install pandas
```

If this command fails, then use a python distribution that already has Pandas installed like, Anaconda, Spyder etc.

Import Pandas

Once Pandas is installed, import it in your applications by adding the `import` keyword:

```
import pandas
```

Now Pandas is imported and ready to use.

What is a Series?

A Pandas Series is like a column in a table.

It is a one-dimensional array holding data of any type.

Example

Create a simple Pandas Series from a list:

```
import pandas as pd

ages = pd.Series([12,32,7,8,14,21,18])

ages
```

```
0    12
1    32
2     7
3     8
4    14
5    21
6    18
dtype: int64
```

Labels

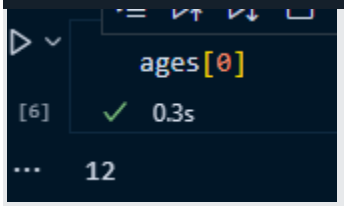
If nothing else is specified, the values are labeled with their index number. First value has index 0, second value has index 1 etc.

This label can be used to access a specified value.

Example

Return the first value of the Series:

```
print(ages[0])
```



Create Labels

With the `index` argument, you can name your own labels.

Example

Create your own labels:

```
import pandas as pd

names = ['person1', 'person2', 'person3', 'person4', 'person5', 'person6',
'person7']

ages = pd.Series([12,32,7,8,14,21,18], index = names)
```

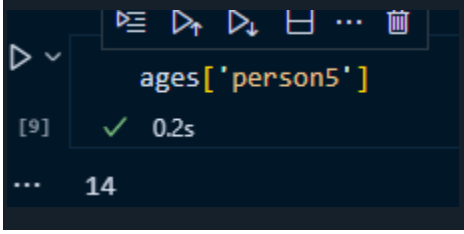
```
person1    12
person2    32
person3     7
person4     8
person5    14
person6    21
person7    18
dtype: int64
```

When you have created labels, you can access an item by referring to the label.

Example

Return the value of "y":

```
print(ages["person5"])
```



Key/Value Objects as Series

You can also use a key/value object, like a dictionary, when creating a Series.

Example

Create a simple Pandas Series from a dictionary:

```
import pandas as pd

Dist = {'person1': 12,
 'person2': 32,
 'person3': 7,
 'person4': 8,
 'person5': 14,
 'person6': 21,
 'person7': 18}


ages = pd.Series(Dist)
```

```
print(ages)
```

```
person1    12
person2    32
person3     7
person4     8
person5    14
person6    21
person7    18
dtype: int64
```

What is a DataFrame?

A Pandas DataFrame is a 2 dimensional data structure, like a 2 dimensional array, or a table with rows and columns.

Example

Create a simple Pandas DataFrame:

```
import pandas as pd

dist = {"names":['omar','abdelah','hossam','amine','mohamed','othman'],
"ages":[13,40,32,16,21,30]
}


#load data into a DataFrame object:
df = pd.DataFrame(dist)

print(df)
```

Result

	names	ages
0	omar	13
1	abdelah	40
2	hossam	32
3	amine	16
4	mohamed	21
5	othman	30

Read CSV Files

A simple way to store big data sets is to use CSV files (comma separated files).

CSV files contains plain text and is a well know format that can be read by everyone including Pandas.

Club Informatique
< ARTS & MÉTIERS />

Example

```
import pandas as pd

df = pd.read_csv("data.csv")

print(df)
```

	Person	ages
0	abdelah	21
1	zouhair	13
2	houssam	34
3	amine	18
4	souhail	23

The head() function is used to get the first n rows.

This function returns the first n rows for the object based on position. It is useful for quickly testing if your object has the right type of data in it.

Syntax:

```
DataFrame.head(self, n=5)
```

The tail() function is used to get the last n rows.

This function returns last n rows from the object based on position. It is useful for quickly verifying data, for example, after sorting or appending rows.

Syntax:

```
DataFrame.tail(self, n=5)
```