7. Pandas objects

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

Python has a series of data containers (list, dicts etc.) and Numpy offers multi-dimensional arrays, however none of these structures offers a simple way neither to handle tabular data, nor to easily do standard database operations. This is why Pandas exists: it offers a complete ecosystem of structures and functions dedicated to handle large tables with inhomogeneous contents.

In this first chapter, we are going to learn about the two main structures of Pandas: Series and Dataframes.

7.1 Series

7.1.1 Simple series

Series are a the Pandas version of 1-D Numpy arrays. We are rarely going to use them directly, but they often appear implicitly when handling data from the more general Dataframe structure. We therefore only give here basics.

To understand Series' specificities, let's create one. Usually Pandas structures (Series and Dataframes) are created from other simpler structures like Numpy arrays or dictionaries:

In [2]:
```
numpy_array = np.array([4,8,38,1,6])
```

The function `pd.Series()` allows us to convert objects into Series:

In [3]:
```
pd_series = pd.Series(numpy_array)
pd_series
```

Out[3]:
```
0     4
1     8
2    38
3     1
4     6
dtype: int64
```

The underlying structure can be recovered with the `.values` attribute:

In [4]:
```
pd_series.values
```

Out[4]:
```
array([ 4,  8, 38,  1,  6])
```

Otherwise, indexing works as for regular arrays:

In [5]:
```
pd_series[1]
```

Out[5]:
```
8
```

7.1.2 Indexing

On top of accessing values in a series by regular indexing, one can create custom indices for each element in the series:

```
In [6]: pd_series2 = pd.Series(numpy_array, index=['a', 'b', 'c', 'd','e'])
```

```
In [7]: pd_series2
```

```
Out[7]: a     4
        b     8
        c    38
        d     1
        e     6
        dtype: int64
```

Now a given element can be accessed either by using its regular index:

```
In [8]: pd_series2[1]
```

```
Out[8]: 8
```

or its chosen index:

```
In [9]: pd_series2['b']
```

```
Out[9]: 8
```

A more direct way to create specific indexes is to transform as dictionary into a Series:

```
In [10]: composer_birth = {'Mahler': 1860, 'Beethoven': 1770, 'Puccini': 1858, 'Shost
         akovich': 1906}
```

```
In [11]: pd_composer_birth = pd.Series(composer_birth)
         pd_composer_birth
```

```
Out[11]: Mahler          1860
         Beethoven       1770
         Puccini         1858
         Shostakovich    1906
         dtype: int64
```

```
In [12]: pd_composer_birth['Puccini']
```

```
Out[12]: 1858
```

7.2 Dataframes

In most cases, one has to deal with more than just one variable, e.g. one has the birth year and the death year of a list of composers. Also one might have different types of information, e.g. in addition to numerical variables (year) one might have string variables like the city of birth. The Pandas structure that allow one to deal with such complex data is called a Dataframe, which can somehow be seen as an aggregation of Series with a common index.

7.2.1 Creating a Dataframe

To see how to construct such a Dataframe, let's create some more information about composers:

In [13]:
```
composer_death = pd.Series({'Mahler': 1911, 'Beethoven': 1827, 'Puccini': 19
24, 'Shostakovich': 1975})
composer_city_birth = pd.Series({'Mahler': 'Kaliste', 'Beethoven': 'Bonn', '
Puccini': 'Lucques', 'Shostakovich': 'Saint-Petersburg'})
```

Now we can combine multiple series into a Dataframe by precising a variable name for each series. Note that all our series need to have the same indices (here the composers' name):

In [14]:
```
composers_df = pd.DataFrame({'birth': pd_composer_birth, 'death': composer_d
eath, 'city': composer_city_birth})
composers_df
```

Out[14]:

	birth	death	city
Mahler	1860	1911	Kaliste
Beethoven	1770	1827	Bonn
Puccini	1858	1924	Lucques
Shostakovich	1906	1975	Saint-Petersburg

A more common way of creating a Dataframe is to construct it directly from a dictionary of lists where each element of the dictionary turns into a column:

In [15]:
```
dict_of_list = {'birth': [1860, 1770, 1858, 1906], 'death':[1911, 1827, 192
4, 1975],
 'city':['Kaliste', 'Bonn', 'Lucques', 'Saint-Petersburg']}
```

In [16]:
```
pd.DataFrame(dict_of_list)
```

Out[16]:

	birth	death	city
0	1860	1911	Kaliste
1	1770	1827	Bonn
2	1858	1924	Lucques
3	1906	1975	Saint-Petersburg

However we now lost the composers name. We can enforce it by providing, as we did before for the Series, a list of indices:

In [17]:
```
pd.DataFrame(dict_of_list, index=['Mahler', 'Beethoven', 'Puccini', 'Shostakovich'])
```

Out[17]:

	birth	death	city
Mahler	1860	1911	Kaliste
Beethoven	1770	1827	Bonn
Puccini	1858	1924	Lucques
Shostakovich	1906	1975	Saint-Petersburg

7.2.2 Accessing values

There are multiple ways of accessing values or series of values in a Dataframe. Unlike in Series, a simple bracket gives access to a column and not an index, for example:

In [18]:
```
composers_df['city']
```

Out[18]:
```
Mahler                  Kaliste
Beethoven                  Bonn
Puccini                 Lucques
Shostakovich    Saint-Petersburg
Name: city, dtype: object
```

returns a Series. Alternatively one can also use the *attributes* synthax and access columns by using:

In [19]:
```
composers_df.city
```

Out[19]:
```
Mahler                  Kaliste
Beethoven                  Bonn
Puccini                 Lucques
Shostakovich    Saint-Petersburg
Name: city, dtype: object
```

The attributes synthax has some limitations, so in case something does not work as expected, revert to the brackets notation.

When specifiying multiple columns, a DataFrame is returned:

In [20]:
```
composers_df[['city', 'birth']]
```

Out[20]:

	city	birth
Mahler	Kaliste	1860
Beethoven	Bonn	1770
Puccini	Lucques	1858
Shostakovich	Saint-Petersburg	1906

One of the important differences with a regular Numpy array is that here, regular indexing doesn't work:

In [21]:
```
#composers_df[0,0]
```

Instead one has to use either the `.iloc[]` or the `.loc[]` method. `.ìloc[]` can be used to recover the regular indexing:

```
In [22]: composers_df.iloc[0,1]
```

```
Out[22]: 1911
```

While `.loc[]` allows one to recover elements by using the **explicit** index, on our case the composers name:

```
In [23]: composers_df.loc['Mahler','death']
```

```
Out[23]: 1911
```

Remember that `loc` and ``ìloc``` use brackets [] and not parenthesis ().

Numpy style indexing works here too

```
In [24]: composers_df.iloc[1:3,:]
```

Out[24]:

	birth	death	city
Beethoven	1770	1827	Bonn
Puccini	1858	1924	Lucques

If you are working with a large table, it might be useful to sometimes have a list of all the columns. This is given by the `.keys()` attribute:

```
In [25]: composers_df.keys()
```

```
Out[25]: Index(['birth', 'death', 'city'], dtype='object')
```

7.2.3 Adding columns

It is very simple to add a column to a Dataframe. One can e.g. just create a column a give it a default value that we can change later:

```
In [26]: composers_df['country'] = 'default'
```

```
In [27]: composers_df
```

Out[27]:

	birth	death	city	country
Mahler	1860	1911	Kaliste	default
Beethoven	1770	1827	Bonn	default
Puccini	1858	1924	Lucques	default
Shostakovich	1906	1975	Saint-Petersburg	default

Or one can use an existing list:

In [28]:
```
country = ['Austria','Germany','Italy','Russia']
```

In [29]:
```
composers_df['country2'] = country
```

In [30]:
```
composers_df
```

Out[30]:

	birth	death	city	country	country2
Mahler	1860	1911	Kaliste	default	Austria
Beethoven	1770	1827	Bonn	default	Germany
Puccini	1858	1924	Lucques	default	Italy
Shostakovich	1906	1975	Saint-Petersburg	default	Russia