3. Operations with Pandas objects

In [1]:
```
import pandas as pd
import numpy as np
```

One of the great advantages of using Pandas to handle tabular data is how simple it is to extract valuable information from them. Here we are going to see various types of operations that are available for this.

3.1 Matrix types of operations

The strength of Numpy is its natural way of handling matrix operations, and Pandas reuses a lot of these features. For example one can use simple mathematical operations to opereate at the cell level:

In [2]:
```
compo_pd = pd.read_excel('Datasets/composers.xlsx')
compo_pd
```

Out[2]:

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

In [3]:
```
compo_pd['birth']*2
```

```
/usr/local/lib/python3.5/dist-packages/pandas/core/computation/check.py:1
9: UserWarning: The installed version of numexpr 2.4.3 is not supported i
n pandas and will be not be used
The minimum supported version is 2.6.1

  ver=ver, min_ver=_MIN_NUMEXPR_VERSION), UserWarning)
```

Out[3]:
```
0    3720
1    3540
2    3716
3    3812
Name: birth, dtype: int64
```

In [4]:
```
np.log(compo_pd['birth'])
```

Out[4]:
```
0    7.528332
1    7.478735
2    7.527256
3    7.552762
Name: birth, dtype: float64
```

Here we applied functions only to series. Indeed, since our Dataframe contains e.g. strings, no operation can be done on it:

In [5]:
```
#compo_pd+1
```

If however we have a homogenous Dataframe, this is possible:

```
In [6]: compo_pd[['birth','death']]*2
```

Out[6]:

	birth	death
0	3720	3822
1	3540	3654
2	3716	3848
3	3812	3950

3.2 Column operations

There are other types of functions whose purpose is to summarize the data. For example the mean or standard deviation. Pandas by default applies such functions column-wise and returns a series containing e.g. the mean of each column:

```
In [7]: np.mean(compo_pd)
```

```
Out[7]: birth    1848.50
        death    1909.25
        dtype: float64
```

Note that columns for which a mean does not make sense, like the city are discarded. A series of commun functions like mean or standard deviation are directly implemented as methods and can be accessed in the alternative form:

```
In [8]: compo_pd.mean()
```

```
Out[8]: birth    1848.50
        death    1909.25
        dtype: float64
```

If you need the mean of only a single column you can of course chains operations:

```
In [9]: compo_pd.birth.mean()
```

```
Out[9]: 1848.5
```

3.3 Operations between Series

We can also do computations with multiple series as we would do with Numpy arrays:

```
In [10]: compo_pd['death']-compo_pd['birth']
         # operations between columns
```

```
Out[10]: 0    51
         1    57
         2    66
         3    69
         dtype: int64
```

We can even use the result of this computation to create a new column in our Dataframe:

In [11]:
```
compo_pd['age'] = compo_pd['death']-compo_pd['birth']
```

In [12]:
```
compo_pd
```

Out[12]:

	composer	birth	death	city	age
0	Mahler	1860	1911	Kaliste	51
1	Beethoven	1770	1827	Bonn	57
2	Puccini	1858	1924	Lucques	66
3	Shostakovich	1906	1975	Saint-Petersburg	69

3.4 Other functions

Sometimes one needs to apply to a column a very specific function that is not provided by default. In that case we can use one of the different `apply` methods of Pandas.

The simplest case is to apply a function to a column, or Series of a DataFrame. Let's say for example that we want to define the the age >60 as 'old' and <60 as 'young'. We can define the following general function:

In [13]:
```
def define_age(x):
    if x>60:
        return 'old'
    else:
        return 'young'
```

In [14]:
```
define_age(30)
```

Out[14]:
```
'young'
```

In [15]:
```
define_age(70)
```

Out[15]:
```
'old'
```

We can now apply this function on an entire Series:

In [16]:
```
compo_pd.age.apply(define_age)
# MZ: apply take variable inputs and return variable outputs
# to apply, you can pass Series, DataFrame; can return DataFrame or sing
le numbers or list of numbers, etc.
```

Out[16]:
```
0    young
1    young
2      old
3      old
Name: age, dtype: object
```

In [17]:
```
# MZ: or using lambda function
compo_pd.age.apply(lambda x: x**2)
```

Out[17]:
```
0    2601
1    3249
2    4356
3    4761
Name: age, dtype: int64
```

And again, if we want, we can directly use this output to create a new column:

In [18]:
```
compo_pd['age_def'] = compo_pd.age.apply(define_age)
compo_pd
# MZ: NB: you can also apply functions to rows of the dataframe
# can be useful to create categorical variables
```

Out[18]:

	composer	birth	death	city	age	age_def
0	Mahler	1860	1911	Kaliste	51	young
1	Beethoven	1770	1827	Bonn	57	young
2	Puccini	1858	1924	Lucques	66	old
3	Shostakovich	1906	1975	Saint-Petersburg	69	old

We can also apply a function to an entire DataFrame. For example we can ask how many composers have birth and death dates within the XIXth century:

In [19]:
```
def nineteen_century_count(x):
    return np.sum((x>=1800)&(x<1900))
#def nineteen_century_count2(x):
#    return np.sum((x>=1800)and(x<1900)) # does not work !!
```

In [20]:
```
5 < 10 and 5 < 6
```

Out[20]:
```
True
```

In [21]:
```
compo_pd[['birth','death']].apply(nineteen_century_count)
#compo_pd[['birth','death']].apply(nineteen_century_count2)
```

Out[21]:
```
birth    2
death    1
dtype: int64
```

The function is applied column-wise and returns a single number for each in the form of a series.

In [22]:
```
def nineteen_century_true(x):
    return (x>=1800)&(x<1900)
```

In [23]:
```
compo_pd[['birth','death']].apply(nineteen_century_true)
```

Out[23]:

	birth	death
0	True	False
1	False	True
2	True	False
3	False	False

Here the operation is again applied column-wise but the output is a Series.

There are more combinations of what can be the in- and output of the apply function and in what order (column- or row-wise) they are applied that cannot be covered here.

3.5 Logical indexing

Just like with Numpy, it is possible to subselect parts of a Dataframe using logical indexing. Let's have a look again at an example:

In [24]:
```
compo_pd
```

Out[24]:

	composer	birth	death	city	age	age_def
0	Mahler	1860	1911	Kaliste	51	young
1	Beethoven	1770	1827	Bonn	57	young
2	Puccini	1858	1924	Lucques	66	old
3	Shostakovich	1906	1975	Saint-Petersburg	69	old

If we use a logical comprison on a series, this yields a **logical Series**:

In [25]:
```
compo_pd['birth'] > 1859
```

Out[25]:
```
0     True
1    False
2    False
3     True
Name: birth, dtype: bool
```

In [26]:
```
log_indexer = compo_pd['birth'] > 1859
log_indexer
compo_pd[log_indexer]
# MZ: select the rows based on logical indexing
# MZ: to negate the logicals
compo_pd[~log_indexer]
# ! again here not is not working !
# compo_pd[not log_indexer] # ERROR !
```

Out[26]:

	composer	birth	death	city	age	age_def
1	Beethoven	1770	1827	Bonn	57	young
2	Puccini	1858	1924	Lucques	66	old

Just like in Numpy we can use this logical Series as an index to select elements in the Dataframe:

In [27]:
```
compo_pd[compo_pd['birth'] > 1859]
```

Out[27]:

	composer	birth	death	city	age	age_def
0	Mahler	1860	1911	Kaliste	51	young
3	Shostakovich	1906	1975	Saint-Petersburg	69	old

We can also create more complex logical indexings:

In [28]:
```
compo_pd[(compo_pd['birth'] > 1859)&(compo_pd['age']>60)]
```

Out[28]:

	composer	birth	death	city	age	age_def
3	Shostakovich	1906	1975	Saint-Petersburg	69	old

And we can create new arrays containing only these subselections:

In [29]:
```
compos_sub = compo_pd[compo_pd['birth'] > 1859]
# MZ how the 2 are connected ??? tricky to know if Pandas create a copy
or not
# best to explicitly create a copy !
```

We can then modify the new array:

In [30]:
```
compos_sub.loc[0,'birth'] = 3000
# warning to tell that something might go wrong (because not used copy)
```

```
/usr/local/lib/python3.5/dist-packages/pandas/core/indexing.py:543: Setti
ngWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pandas-doc
s/stable/indexing.html#indexing-view-versus-copy
  self.obj[item] = s
```

Note that we get this SettingWithCopyWarning warning. This is a very common problem hand has to do with how new arrays are created when making subselections. Simply stated, did we create an entirely new array or a "view" of the old one? This will be very case-dependent and to avoid this, if we want to create a new array we can just enforce it using the `copy()` method (for more information on the topic see for example this explanation (https://www.dataquest.io/blog/settingwithcopywarning/):

In [31]:
```
# MZ: better to explicitly create a copy
compos_sub2 = compo_pd[compo_pd['birth'] > 1859].copy()
compos_sub2.loc[0,'birth'] = 3000
```