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.

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 operate at the cell level:

In [2]:
```
compo_pd = pd.read_excel('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
```

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']]
```

Out[6]:

	birth	death
0	1860	1911
1	1770	1827
2	1858	1924
3	1906	1975

In [7]:
```
compo_pd[['birth','death']]*2
```

Out[7]:

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

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

```
C:\Users\user\anaconda3\lib\site-packages\numpy\core\fromnumeric.py:3438: Fut
ureWarning: In a future version, DataFrame.mean(axis=None) will return a scal
ar mean over the entire DataFrame. To retain the old behavior, use 'frame.mea
n(axis=0)' or just 'frame.mean()'
  return mean(axis=axis, dtype=dtype, out=out, **kwargs)
C:\Users\user\anaconda3\lib\site-packages\numpy\core\fromnumeric.py:3438: Fut
ureWarning: Dropping of nuisance columns in DataFrame reductions (with 'numer
ic_only=None') is deprecated; in a future version this will raise TypeError.
Select only valid columns before calling the reduction.
  return mean(axis=axis, dtype=dtype, out=out, **kwargs)
```

Out[8]:
```
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 common functions like mean or standard deviation are directly implemented as methods and can be accessed in the alternative form:

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

Out[8]:

	birth	death
count	4.000000	4.000000
mean	1848.500000	1909.250000
std	56.836021	61.396933
min	1770.000000	1827.000000
25%	1836.000000	1890.000000
50%	1859.000000	1917.500000
75%	1871.500000	1936.750000
max	1906.000000	1975.000000

In [10]:
```
compo_pd.std()
```

```
C:\Users\user\AppData\Local\Temp\ipykernel_14516\237597691.py:1: FutureWarnin
g: Dropping of nuisance columns in DataFrame reductions (with 'numeric_only=N
one') is deprecated; in a future version this will raise TypeError.  Select o
nly valid columns before calling the reduction.
  compo_pd.std()
```

Out[10]:
```
birth    56.836021
death    61.396933
dtype: float64
```

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

In [11]:
```
compo_pd.birth.mean()
```

Out[11]:
```
1848.5
```

Operations between Series

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

In [9]:
```
compo_pd['death']-compo_pd['birth']
```

Out[9]:
```
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 [10]:
```
compo_pd
```

Out[10]:

	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 [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

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)
```

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

In [17]:
```
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
```

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))
```

In [20]:
```
compo_pd[['birth','death']].apply(nineteen_century_count)
```

Out[20]:
```
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 [21]:
```
def nineteen_century_true(x):
    return (x>=1800)&(x<1900)
```

In [22]:
```
compo_pd[['birth','death']].apply(nineteen_century_true)
```

Out[22]:

	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.

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 [23]:
```
compo_pd
```

Out[23]:

	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 comparison on a series, this yields a **logical Series**:

In [24]:
```
compo_pd['birth']
```

Out[24]:
```
0    1860
1    1770
2    1858
3    1906
Name: birth, dtype: int64
```

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

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

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

In [26]:
```
log_indexer = compo_pd['birth'] > 1859
log_indexer
```

Out[26]:
```
0     True
1    False
2    False
3     True
Name: birth, dtype: bool
```

In [27]:
```
compo_pd
```

Out[27]:

	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

In [28]:
```
#complement of log_indexer
~log_indexer
```

Out[28]:
```
0    False
1     True
2     True
3    False
Name: birth, dtype: bool
```

In [29]:
```
compo_pd[~log_indexer]
```

Out[29]:

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

We can also create more complex logical indexings:

In [30]:
```
(compo_pd['birth'] > 1859)&(compo_pd['age']>60)
```

Out[30]:
```
0    False
1    False
2    False
3     True
dtype: bool
```

In [31]:
```
compo_pd[(compo_pd['birth'] > 1859)&(compo_pd['age']>60)]
```

Out[31]:

	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 [32]:
```
compos_sub = compo_pd[compo_pd['birth'] > 1859]
```

In [33]:
```
compos_sub
```

Out[33]:

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

We can then modify the new array:

In [34]:
```
compos_sub.loc[0,'birth'] = 3000
```

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/)](https://www.dataquest.io/blog/settingwithcopywarning/):

In [35]:
```
compos_sub2 = compo_pd[compo_pd['birth'] > 1859].copy()
compos_sub2.loc[0,'birth'] = 3000
```

In [36]:
```
compos_sub2
```

Out[36]:

	composer	birth	death	city	age	age_def
0	Mahler	3000	1911	Kaliste	51	young
3	Shostakovich	1906	1975	Saint-Petersburg	69	old