1 Curs 4: Pandas - elemente avansate

1.1 Lucrul cu valori lipsa in Pandas

1.1.1 Reprezentarea valorilor lipsa in Pandas

Pandas foloseste doua variante pentru reprezentarea de valori lipsa: None si NaN. NaN este utilizat pentru tipuri numerice in virgula mobila. None este convertit la NaN daca seria este numerica; daca seria este ne-numerica, se considera de tip `object` :

In [1]:

```
import pandas as pd
import numpy as np
```

executed in 596ms, finished 10:14:51 2021-03-15

In [2]:

```
print(f'pandas version: {pd.__version__}')
print(f'numpy version: {np.__version__}')

# pandas version: 1.2.3
# numpy version: 1.19.2
```

executed in 8ms, finished 10:14:51 2021-03-15

```
pandas version: 1.2.3
numpy version: 1.19.2
```

NaN si None sunt echivalene in context numeric, in Pandas:

In [3]:

```
pd.Series([1, np.nan, 2, None])
```

executed in 23ms, finished 10:14:51 2021-03-15

Out[3]:

```
0    1.0
1    NaN
2    2.0
3    NaN
dtype: float64
```

In [4]:

```
pd.Series(['John', 'Danny', None])
```

executed in 9ms, finished 10:14:51 2021-03-15

Out[4]:

```
0     John
1    Danny
2     None
dtype: object
```

Intrucat doar tipurile numerice floating point suporta valoare de NaN, conform standardulului IEEE 754, se va face transformarea unei serii de tip intreg intr-una de tip floating point daca se insereaza sau adauga un NaN:

In [5]:

```
# creare de serie cu valori intregi
x = pd.Series([10, 20], dtype=int)
x
```

executed in 10ms, finished 10:14:51 2021-03-15

Out[5]:

```
0    10
1    20
dtype: int32
```

In [6]:

```
x[1] = np.nan
x
```

executed in 10ms, finished 10:14:51 2021-03-15

Out[6]:

```
0    10.0
1     NaN
dtype: float64
```

In [7]:

```
# adaugare cu append
x = pd.Series([10, 20], dtype=int)
print(f'Serie de intregi:\n{x}')
x = x.append(pd.Series([100, np.nan]))
print(f'Dupa adaugare:\n{x}')
```

executed in 10ms, finished 10:14:51 2021-03-15

```
Serie de intregi:
0    10
1    20
dtype: int32
Dupa adaugare:
0     10.0
1     20.0
0    100.0
1      NaN
dtype: float64
```

1.1.2 Operatii cu valori lipsa in Pandas

Metodele ce se pot folosi pentru operarea cu valori lipsa sunt:

- `isnull()` - genereaza o matrice de valori logice, ce specifica daca pe pozitiile corespunzatoare sunt valori lipsa
- `notnull()` - complementara lui `isnull()`
- `dropna()` - returneaza o versiune filtrata a datelor, doar acele linii care nu au null
- `fillna()` - returneaza o copie a obiectului initial, in care valorile lipsa sunt umplute cu ceva specificat

1.1.2.1 `isnull()` si `notnull()`

In [8]:

```
data = pd.Series([1, np.nan, 'hello', None])
data
```

executed in 8ms, finished 10:14:51 2021-03-15

Out[8]:

```
0        1
1      NaN
2    hello
3     None
dtype: object
```

In [9]:

```
data.isnull()
```

executed in 7ms, finished 10:14:52 2021-03-15

Out[9]:

```
0    False
1     True
2    False
3     True
dtype: bool
```

In [10]:

```
data.notnull()
```

executed in 9ms, finished 10:14:52 2021-03-15

Out[10]:

```
0     True
1    False
2     True
3    False
dtype: bool
```

Selectarea doar acelor valori din obiectul Series care sunt ne-nule se face cu:

In [11]:

```
# filtrare
data[data.notnull()]
```

executed in 9ms, finished 10:14:52 2021-03-15

Out[11]:

```
0        1
2    hello
dtype: object
```

Functiile `isnull()` si `notnull()` functioneaza la fel si pentru obiecte DataFrame:

In [12]:

```
df = pd.DataFrame({'Name': ['Will', 'Mary', 'Joan'], 'Age': [20, 25, 30]})
df
```

executed in 15ms, finished 10:14:52 2021-03-15

Out[12]:

	Name	Age
0	Will	20
1	Mary	25
2	Joan	30

In [13]:

```
df.loc[2, 'Age'] = np.NaN
df
```

executed in 12ms, finished 10:14:52 2021-03-15

Out[13]:

	Name	Age
0	Will	20.0
1	Mary	25.0
2	Joan	NaN

In [14]:

```
df.isnull()
```

executed in 11ms, finished 10:14:52 2021-03-15

Out[14]:

	Name	Age
0	False	False
1	False	False
2	False	True

In [15]:

```
df.notnull()
```

executed in 12ms, finished 10:14:52 2021-03-15

Out[15]:

	Name	Age
0	True	True
1	True	True
2	True	False

In cazul obiectelor DataFrame, aplicarea lui `notnull()` nu lasa afara elemente din dataframe:

In [16]:

```
df[df.notnull()]
```

executed in 26ms, finished 10:14:52 2021-03-15

Out[16]:

	Name	Age
0	Will	20.0
1	Mary	25.0
2	Joan	NaN

1.1.2.2 Stergerea de elemente cu `dropna()`

Pentru un obiect Series, metoda `dropna()` produce un alt obiect in care liniile cu valori de null sunt sterse:

In [17]:

```
data
```

executed in 7ms, finished 10:14:52 2021-03-15

Out[17]:

```
0        1
1      NaN
2    hello
3     None
dtype: object
```

In [18]:

```
data2 = data.dropna()
data2
```

executed in 7ms, finished 10:14:52 2021-03-15

Out[18]:

```
0        1
2    hello
dtype: object
```

Functia dropna nu modifica obiectul sursa, decat daca se sepecifica `inplace=True` :

In [19]:

```
data
```

executed in 6ms, finished 10:14:52 2021-03-15

Out[19]:

```
0        1
1      NaN
2    hello
3     None
dtype: object
```

Pentru un obiect DataFrame se pot sterge doar linii sau coloane in intregime - obiectul care ramane trebuie sa fie tot un DataFrame:

In [20]:

```
df = pd.DataFrame([[1, np.nan, 2],[2, 3, 5],[np.nan, 4, 6]])
df
```

executed in 14ms, finished 10:14:52 2021-03-15

Out[20]:

	0	1	2
0	1.0	NaN	2
1	2.0	3.0	5
2	NaN	4.0	6

In [21]:

```
# Implicit: eliminare de linii care contin null
df2 = df.dropna()
df2
```

executed in 15ms, finished 10:14:52 2021-03-15

Out[21]:

	0	1	2
1	2.0	3.0	5

Mai sus s-a ales implicit stergerea de linii, datorita faptului ca parametrul `axis` are implicit valoarea 0:

In [22]:

```
help(df.dropna)
```

executed in 6ms, finished 10:14:52 2021-03-15

```
Help on method dropna in module pandas.core.frame:

dropna(axis=0, how='any', thresh=None, subset=None, inplace=False) method of
pandas.core.frame.DataFrame instance
    Remove missing values.

    See the :ref:`User Guide <missing_data>` for more on which values are
    considered missing, and how to work with missing data.

    Parameters
    ----------
    axis : {0 or 'index', 1 or 'columns'}, default 0
        Determine if rows or columns which contain missing values are
        removed.

        * 0, or 'index' : Drop rows which contain missing values.
        * 1, or 'columns' : Drop columns which contain missing value.

        .. versionchanged:: 1.0.0

           Pass tuple or list to drop on multiple axes.
           Only a single axis is allowed.

    how : {'any', 'all'}, default 'any'
        Determine if row or column is removed from DataFrame, when we have
        at least one NA or all NA.

        * 'any' : If any NA values are present, drop that row or column.
        * 'all' : If all values are NA, drop that row or column.

    thresh : int, optional
        Require that many non-NA values.
    subset : array-like, optional
        Labels along other axis to consider, e.g. if you are dropping rows
        these would be a list of columns to include.
    inplace : bool, default False
        If True, do operation inplace and return None.

    Returns
    -------
    DataFrame or None
        DataFrame with NA entries dropped from it or None if ``inplace=True`
`.

    See Also
    --------
    DataFrame.isna: Indicate missing values.
    DataFrame.notna : Indicate existing (non-missing) values.
    DataFrame.fillna : Replace missing values.
    Series.dropna : Drop missing values.
    Index.dropna : Drop missing indices.

    Examples
    --------
    >>> df = pd.DataFrame({"name": ['Alfred', 'Batman', 'Catwoman'],
    ...                    "toy": [np.nan, 'Batmobile', 'Bullwhip'],
```

```
...                     "born": [pd.NaT, pd.Timestamp("1940-04-25"),
...                              pd.NaT]})
>>> df
       name        toy       born
0    Alfred        NaN        NaT
1    Batman  Batmobile 1940-04-25
2  Catwoman   Bullwhip        NaT

Drop the rows where at least one element is missing.

>>> df.dropna()
     name        toy       born
1  Batman  Batmobile 1940-04-25

Drop the columns where at least one element is missing.

>>> df.dropna(axis='columns')
       name
0    Alfred
1    Batman
2  Catwoman

Drop the rows where all elements are missing.

>>> df.dropna(how='all')
       name        toy       born
0    Alfred        NaN        NaT
1    Batman  Batmobile 1940-04-25
2  Catwoman   Bullwhip        NaT

Keep only the rows with at least 2 non-NA values.

>>> df.dropna(thresh=2)
       name        toy       born
1    Batman  Batmobile 1940-04-25
2  Catwoman   Bullwhip        NaT

Define in which columns to look for missing values.

>>> df.dropna(subset=['name', 'toy'])
       name        toy       born
1    Batman  Batmobile 1940-04-25
2  Catwoman   Bullwhip        NaT

Keep the DataFrame with valid entries in the same variable.

>>> df.dropna(inplace=True)
>>> df
     name        toy       born
1  Batman  Batmobile 1940-04-25
```

Se poate opta pentru stergerea de coloane care contin null:

In [23]:

```
df
```

executed in 10ms, finished 10:14:52 2021-03-15

Out[23]:

	0	1	2
0	1.0	NaN	2
1	2.0	3.0	5
2	NaN	4.0	6

In [24]:

```
# stergere de coloane cu null
# df3 = df.dropna(axis=1) # functioneaza
df3 = df.dropna(axis='columns')
df3
```

executed in 14ms, finished 10:14:52 2021-03-15

Out[24]:

	2
0	2
1	5
2	6

Operatiile de mai sus sterg o linie sau o coloana daca ea contine cel putin o valoare de null. Se poate cere stergerea doar in cazul in care intreaga linie sau coloana e plina cu null, folosind parametrul `how` :

In [25]:

```
df
```

executed in 11ms, finished 10:14:52 2021-03-15

Out[25]:

	0	1	2
0	1.0	NaN	2
1	2.0	3.0	5
2	NaN	4.0	6

In [26]:

```
df2 = df.dropna(how='all')
df2
```

executed in 14ms, finished 10:14:52 2021-03-15

Out[26]:

	0	1	2
0	1.0	NaN	2
1	2.0	3.0	5
2	NaN	4.0	6

De remarcat ca `dropna()` nu modifica obiectul originar, decat daca se specifica parametrul `inplace=True` .

1.1.2.3 Umplerea de valori nule cu `fillna()`

In [27]:

```
data = pd.Series([1, np.nan, 2, None, 3], index=list('abcde'))
```

executed in 5ms, finished 10:14:52 2021-03-15

In [28]:

```
# umplere cu valoare constanta
data2 = data.fillna(0)
data2
```

executed in 8ms, finished 10:14:52 2021-03-15

Out[28]:

```
a    1.0
b    0.0
c    2.0
d    0.0
e    3.0
dtype: float64
```

In [29]:

```
# Umplere cu copierea ultimei valori cunoscute:
data2 = data.fillna(method='ffill')
data2
```

executed in 8ms, finished 10:14:52 2021-03-15

Out[29]:

```
a    1.0
b    1.0
c    2.0
d    2.0
e    3.0
dtype: float64
```

In [30]:

```
# Umplere 'inapoi':
data2 = data.fillna(method='bfill')
data2
```

executed in 9ms, finished 10:14:52 2021-03-15

Out[30]:

```
a    1.0
b    2.0
c    2.0
d    3.0
e    3.0
dtype: float64
```

In [31]:

```
# umplerea cu valoare calculata:
print(f'Media valorilor non-nan este: {data.mean()}')
data2 = data.fillna(data.mean())
data2
```

executed in 11ms, finished 10:14:52 2021-03-15

```
Media valorilor non-nan este: 2.0
```

Out[31]:

```
a    1.0
b    2.0
c    2.0
d    2.0
e    3.0
dtype: float64
```

1.2 Agregare si grupare

1.2.1 Agregari simple

In [32]:

```
np.random.seed(100)
ser = pd.Series(np.random.rand(10))
ser
```

executed in 9ms, finished 10:14:52 2021-03-15

Out[32]:

```
0    0.543405
1    0.278369
2    0.424518
3    0.844776
4    0.004719
5    0.121569
6    0.670749
7    0.825853
8    0.136707
9    0.575093
dtype: float64
```

In [33]:

```
ser.sum(), ser.max(), ser.min()
```

executed in 7ms, finished 10:14:52 2021-03-15

Out[33]:

```
(4.425757785871915, 0.8447761323199037, 0.004718856190972565)
```

Pentru obiecte DataFrame, operatiile de agregare opereaza pe coloane:

In [34]:

```
df = pd.DataFrame({'A': np.random.rand(10), 'B': -np.random.rand(10) }, index=['line ' +
df
```

executed in 15ms, finished 10:14:52 2021-03-15

Out[34]:

	A	B
line 1	0.891322	-0.431704
line 2	0.209202	-0.940030
line 3	0.185328	-0.817649
line 4	0.108377	-0.336112
line 5	0.219697	-0.175410
line 6	0.978624	-0.372832
line 7	0.811683	-0.005689
line 8	0.171941	-0.252426
line 9	0.816225	-0.795663
line 10	0.274074	-0.015255

In [35]:

```
df.mean()
```

executed in 11ms, finished 10:14:52 2021-03-15

Out[35]:

```
A    0.466647
B   -0.414277
dtype: float64
```

.. si daca se doreste calculul pe linii, se poate indica via parametrul `axis` :

In [36]:

```
# df.mean(axis=1)
df.mean(axis='columns')
```

executed in 8ms, finished 10:14:52 2021-03-15

Out[36]:

```
line 1     0.229809
line 2    -0.365414
line 3    -0.316161
line 4    -0.113868
line 5     0.022144
line 6     0.302896
line 7     0.402997
line 8    -0.040243
line 9     0.010281
line 10    0.129409
dtype: float64
```

Exista o metoda utila, care pentru un obiect DataFrame calculeaza statisticile:

In [37]:

```
df.describe(include='all')
```

executed in 21ms, finished 10:14:52 2021-03-15

Out[37]:

	A	B
count	10.000000	10.000000
mean	0.466647	-0.414277
std	0.356280	0.333688
min	0.108377	-0.940030
25%	0.191297	-0.704673
50%	0.246886	-0.354472
75%	0.815089	-0.194664
max	0.978624	-0.005689

Operatiile nu iau in considerare valorile lipsa:

In [38]:

```
df.iloc[0, 0] = df.iloc[0,1] = np.nan
df.iloc[5, 0] = df.iloc[7, 1] = df.iloc[9, 1] = np.nan
df
```

executed in 14ms, finished 10:14:52 2021-03-15

Out[38]:

	A	B
line 1	NaN	NaN
line 2	0.209202	-0.940030
line 3	0.185328	-0.817649
line 4	0.108377	-0.336112
line 5	0.219697	-0.175410
line 6	NaN	-0.372832
line 7	0.811683	-0.005689
line 8	0.171941	NaN
line 9	0.816225	-0.795663
line 10	0.274074	NaN

In [39]:

```
df.describe(include='all')
```

executed in 20ms, finished 10:14:52 2021-03-15

Out[39]:

	A	B
count	8.000000	7.000000
mean	0.349566	-0.491912
std	0.290390	0.359217
min	0.108377	-0.940030
25%	0.181981	-0.806656
50%	0.214450	-0.372832
75%	0.408476	-0.255761
max	0.816225	-0.005689

In [40]:

```
df.count()
```

executed in 10ms, finished 10:14:52 2021-03-15

Out[40]:

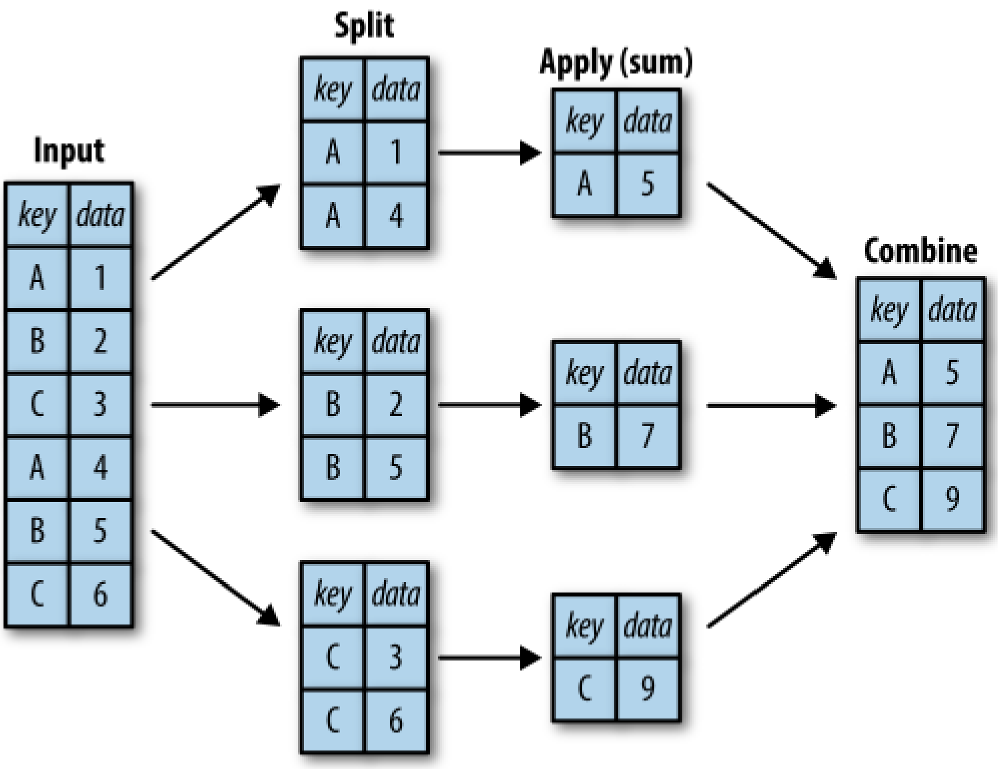
```
A    8
B    7
dtype: int64
```

Metoda de agregare	Descriere
count()	Numarul total de elemente
first(), last()	primul si ultimul element
mean(), median()	Media si mediana
min(), max()	Minimul si maximul
std(), var()	Deviatia standard si varianta
mad()	Deviatia absoluta medie
prod(), sum()	Produsul si suma elementelor

1.2.2 Gruparea datelor: `split()`, `apply()`, `combine()`

Pasii care se fac pentru agregarea datelor urmeaza secventa: imparte, aplica operatie, combina:

1. imparte - via metoda `split()` : separa datele initiale in grupuri, pe baza unei chei
2. aplica, via metoda `apply()` : calculeaza o functie pentru fiecare grup: agregare, transformare, filtrare
3. combina, via metoda `combine()` : concateneaza rezultatele si produ raspunsul final



In [41]:

```
df = pd.DataFrame({'key': ['A', 'B', 'C', 'A', 'B', 'C'], 'data': range(6)}, columns=['ke
df
```

executed in 14ms, finished 10:14:52 2021-03-15

Out[41]:

	key	data
0	A	0
1	B	1
2	C	2
3	A	3
4	B	4
5	C	5

In [42]:

```
groups = df.groupby('key')
type(groups)
```

executed in 7ms, finished 10:14:52 2021-03-15

Out[42]:

```
pandas.core.groupby.generic.DataFrameGroupBy
```

In [43]:

```
print(groups)
```

executed in 5ms, finished 10:14:52 2021-03-15

```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x000002788FC42280>
```

In [44]:

```
groups.sum()
```

executed in 14ms, finished 10:14:52 2021-03-15

Out[44]:

	data
key	
A	3
B	5
C	7

Ca functie de agregare se poate folosi orice functie Pandas sau NumPy.

In [45]:

```
import seaborn as sns
planets = sns.load_dataset('planets')
```

executed in 914ms, finished 10:14:53 2021-03-15

In [46]:

```
planets.head()
```

executed in 15ms, finished 10:14:53 2021-03-15

Out[46]:

	method	number	orbital_period	mass	distance	year
0	Radial Velocity	1	269.300	7.10	77.40	2006
1	Radial Velocity	1	874.774	2.21	56.95	2008
2	Radial Velocity	1	763.000	2.60	19.84	2011
3	Radial Velocity	1	326.030	19.40	110.62	2007
4	Radial Velocity	1	516.220	10.50	119.47	2009

In [47]:

```
planets.describe(include='all')
```

executed in 33ms, finished 10:14:53 2021-03-15

Out[47]:

	method	number	orbital_period	mass	distance	year
count	1035	1035.000000	992.000000	513.000000	808.000000	1035.000000
unique	10	NaN	NaN	NaN	NaN	NaN
top	Radial Velocity	NaN	NaN	NaN	NaN	NaN
freq	553	NaN	NaN	NaN	NaN	NaN
mean	NaN	1.785507	2002.917596	2.638161	264.069282	2009.070531
std	NaN	1.240976	26014.728304	3.818617	733.116493	3.972567
min	NaN	1.000000	0.090706	0.003600	1.350000	1989.000000
25%	NaN	1.000000	5.442540	0.229000	32.560000	2007.000000
50%	NaN	1.000000	39.979500	1.260000	55.250000	2010.000000
75%	NaN	2.000000	526.005000	3.040000	178.500000	2012.000000
max	NaN	7.000000	730000.000000	25.000000	8500.000000	2014.000000

In [48]:

```
planets.info()
```

executed in 14ms, finished 10:14:53 2021-03-15

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1035 entries, 0 to 1034
Data columns (total 6 columns):
 #   Column          Non-Null Count  Dtype  
---  ------          --------------  -----  
 0   method          1035 non-null   object 
 1   number          1035 non-null   int64  
 2   orbital_period  992 non-null    float64
 3   mass            513 non-null    float64
 4   distance        808 non-null    float64
 5   year            1035 non-null   int64  
dtypes: float64(3), int64(2), object(1)
memory usage: 48.6+ KB
```

In [49]:

```
planets.describe()
```

executed in 32ms, finished 10:14:53 2021-03-15

Out[49]:

	number	orbital_period	mass	distance	year
count	1035.000000	992.000000	513.000000	808.000000	1035.000000
mean	1.785507	2002.917596	2.638161	264.069282	2009.070531
std	1.240976	26014.728304	3.818617	733.116493	3.972567
min	1.000000	0.090706	0.003600	1.350000	1989.000000
25%	1.000000	5.442540	0.229000	32.560000	2007.000000
50%	1.000000	39.979500	1.260000	55.250000	2010.000000
75%	2.000000	526.005000	3.040000	178.500000	2012.000000
max	7.000000	730000.000000	25.000000	8500.000000	2014.000000

In [50]:

```
planets['method'].unique()
```

executed in 10ms, finished 10:14:53 2021-03-15

Out[50]:

```
array(['Radial Velocity', 'Imaging', 'Eclipse Timing Variations',
       'Transit', 'Astrometry', 'Transit Timing Variations',
       'Orbital Brightness Modulation', 'Microlensing', 'Pulsar Timing',
       'Pulsation Timing Variations'], dtype=object)
```

Pentru grupurile rezultate se poate alege o coloana, pentru care sa se calculeze valori agregate:

In [51]:

```
planets.groupby('method')['orbital_period'].median()
```

executed in 10ms, finished 10:14:53 2021-03-15

Out[51]:

```
method
Astrometry                          631.180000
Eclipse Timing Variations          4343.500000
Imaging                           27500.000000
Microlensing                       3300.000000
Orbital Brightness Modulation         0.342887
Pulsar Timing                        66.541900
Pulsation Timing Variations        1170.000000
Radial Velocity                     360.200000
Transit                               5.714932
Transit Timing Variations            57.011000
Name: orbital_period, dtype: float64
```

Grupurile pot fi iterate, returnand pentru fiecare grup un obiect de tip Series sau DataFrame:

In [52]:

```
print(f'Number of columns: {len(planets.columns)}')

for (method, group) in planets.groupby('method'):
    print("{0:30s} shape={1}".format(method, group.shape))
```

executed in 9ms, finished 10:14:53 2021-03-15

```
Number of columns: 6
Astrometry                     shape=(2, 6)
Eclipse Timing Variations      shape=(9, 6)
Imaging                        shape=(38, 6)
Microlensing                   shape=(23, 6)
Orbital Brightness Modulation  shape=(3, 6)
Pulsar Timing                  shape=(5, 6)
Pulsation Timing Variations    shape=(1, 6)
Radial Velocity                shape=(553, 6)
Transit                        shape=(397, 6)
Transit Timing Variations      shape=(4, 6)
```

Fiecare grup rezultat, fiind vazut ca un Series sau DataFrame, suporta apel de metode aferete acestor obiecte:

In [53]:

```
planets.groupby('method')['year'].describe()
```

executed in 48ms, finished 10:14:53 2021-03-15

Out[53]:

method	count	mean	std	min	25%	50%	75%	max
Astrometry	2.0	2011.500000	2.121320	2010.0	2010.75	2011.5	2012.25	2013.0
Eclipse Timing Variations	9.0	2010.000000	1.414214	2008.0	2009.00	2010.0	2011.00	2012.0
Imaging	38.0	2009.131579	2.781901	2004.0	2008.00	2009.0	2011.00	2013.0
Microlensing	23.0	2009.782609	2.859697	2004.0	2008.00	2010.0	2012.00	2013.0
Orbital Brightness Modulation	3.0	2011.666667	1.154701	2011.0	2011.00	2011.0	2012.00	2013.0
Pulsar Timing	5.0	1998.400000	8.384510	1992.0	1992.00	1994.0	2003.00	2011.0
Pulsation Timing Variations	1.0	2007.000000	NaN	2007.0	2007.00	2007.0	2007.00	2007.0
Radial Velocity	553.0	2007.518987	4.249052	1989.0	2005.00	2009.0	2011.00	2014.0
Transit	397.0	2011.236776	2.077867	2002.0	2010.00	2012.0	2013.00	2014.0
Transit Timing Variations	4.0	2012.500000	1.290994	2011.0	2011.75	2012.5	2013.25	2014.0

1.2.3 Metodele `aggregate()` , `filter()` , `transform()` , `apply()`

Inainte de pasul de combinare a datelor se pot folosi metode care implementeaza operatii pe grupuri inainte de a face in final gruparea rezultatelor din grupuri.

In [54]:

```
df = pd.DataFrame({'key': ['A', 'B', 'C', 'A', 'B', 'C'],
'data1': range(6),
'data2': np.random.randint(0, 10, 6)},
columns = ['key', 'data1', 'data2'])
df
```

executed in 13ms, finished 10:14:53 2021-03-15

Out[54]:

	key	data1	data2
0	A	0	5
1	B	1	8
2	C	2	1
3	A	3	0
4	B	4	7
5	C	5	6

Metoda `aggregate()` permite specificare de functii prin numele lor (string sau referinta la functie):

In [55]:

```
df.groupby('key').aggregate(['min', np.median, max])
```

executed in 23ms, finished 10:14:53 2021-03-15

Out[55]:

	data1			data2		
	min	**median**	**max**	**min**	**median**	**max**
key						
A	0	1.5	3	0	2.5	5
B	1	2.5	4	7	7.5	8
C	2	3.5	5	1	3.5	6

Filtrarea cu `filter()` permite selectarea doar acelor grupuri care satisfac o anumita conditie:

In [56]:

```
def filter_func(x): # x este o linie, corespunzand fiecarui grup
    return x['data2'].std() > 3
```

executed in 5ms, finished 10:14:53 2021-03-15

In [57]:

```
df.groupby('key').std()
```

executed in 19ms, finished 10:14:53 2021-03-15

Out[57]:

	data1	data2
key		
A	2.12132	3.535534
B	2.12132	0.707107
C	2.12132	3.535534

In [58]:

```
df.groupby('key').filter(filter_func)
```

executed in 14ms, finished 10:14:53 2021-03-15

Out[58]:

	key	data1	data2
0	A	0	5
2	C	2	1
3	A	3	0
5	C	5	6

Acelasi efect se obtine cu lambda functii:

In [59]:

```
df.groupby('key').filter(lambda row: row['data2'].std() > 3)
```

executed in 15ms, finished 10:14:53 2021-03-15

Out[59]:

	key	data1	data2
0	A	0	5
2	C	2	1
3	A	3	0
5	C	5	6

Transformarea cu `transform()` produce un dataframe cu acelasi numar de linii ca si cel initial, dar cu valorile calculate prin aplicarea unei operatii la nivelul fiecarui grup:

In [60]:

```
df
```

executed in 13ms, finished 10:14:53 2021-03-15

Out[60]:

	key	data1	data2
0	A	0	5
1	B	1	8
2	C	2	1
3	A	3	0
4	B	4	7
5	C	5	6

Media pe fieare grup este:

In [61]:

```
df.groupby('key').mean()
```

executed in 19ms, finished 10:14:53 2021-03-15

Out[61]:

key	data1	data2
A	1.5	2.5
B	2.5	7.5
C	3.5	3.5

Centrarea valorilor pentru fiecare grup - adica: in fiecare grup sa fie media 0 - se face cu:

In [62]:

```
df.groupby('key').transform(lambda x: x - x.mean())
```

executed in 28ms, finished 10:14:53 2021-03-15

Out[62]:

	data1	data2
0	-1.5	2.5
1	-1.5	0.5
2	-1.5	-2.5
3	1.5	-2.5
4	1.5	-0.5
5	1.5	2.5

In [63]:

```
df.groupby('key').transform(lambda x: x - x.mean()).mean()
```

executed in 25ms, finished 10:14:53 2021-03-15

Out[63]:

```
data1    0.0
data2    0.0
dtype: float64
```

In [64]:

```
df.groupby('key').transform(lambda x: x - x.mean())
```

executed in 26ms, finished 10:14:53 2021-03-15

Out[64]:

	data1	data2
0	-1.5	2.5
1	-1.5	0.5
2	-1.5	-2.5
3	1.5	-2.5
4	1.5	-0.5
5	1.5	2.5

Functia `apply()` permite calculul unei functii peste fiecare grup. Exemplul de mai jos calculeaza prima coloana impartita la suma elementelor din coloana data2, in cadrul fiecarui grup:

In [65]:

```
def norm_by_data2(x):
    # x is a DataFrame of group values
    x['data1'] /= x['data2'].sum()
    return x

df.groupby('key').apply(norm_by_data2)
```

executed in 26ms, finished 10:14:53 2021-03-15

Out[65]:

	key	data1	data2
0	A	0.000000	5
1	B	0.066667	8
2	C	0.285714	1
3	A	0.600000	0
4	B	0.266667	7
5	C	0.714286	6

Functia `apply()` se poate folosi si in afara lui `groupby` , permitand calcul vectorizat de mare viteza:

In [66]:

```
data_len = 10000

df_big = pd.DataFrame({'Noise_' + str(i) : np.random.rand(data_len) for i in range(1, 50)

df_big.head(n=10)
```

executed in 55ms, finished 10:14:53 2021-03-15

Out[66]:

	Noise_1	Noise_2	Noise_3	Noise_4	Noise_5	Noise_6	Noise_7	Noise_8	Noise_9	N
0	0.030123	0.203968	0.706581	0.298033	0.534726	0.515900	0.258939	0.413919	0.026733	(
1	0.776005	0.688731	0.204790	0.082986	0.053910	0.295277	0.478298	0.878959	0.426999	(
2	0.550958	0.953967	0.185411	0.603051	0.411614	0.204954	0.782968	0.377960	0.100514	(
3	0.381073	0.756840	0.121745	0.999780	0.766192	0.881829	0.667565	0.271940	0.286227	(
4	0.529266	0.347373	0.184114	0.983282	0.353940	0.246467	0.866640	0.575963	0.430655	(
5	0.956877	0.593913	0.343684	0.049248	0.186315	0.339804	0.413873	0.583093	0.777096	(
6	0.175821	0.465888	0.132411	0.655425	0.076958	0.183606	0.648024	0.516551	0.944216	(
7	0.118303	0.389591	0.358417	0.313715	0.577916	0.055776	0.907747	0.979670	0.200293	(
8	0.862946	0.269849	0.186387	0.951164	0.929856	0.040157	0.035268	0.834389	0.629915	(
9	0.074867	0.811474	0.787127	0.162639	0.936675	0.637662	0.739710	0.711666	0.203566	(

10 rows × 49 columns

In [67]:

```
all_noise_columns = [column for column in df_big.columns if column.startswith('Noise_')]

row = df_big.iloc[0]
row[all_noise_columns]
```

executed in 11ms, finished 10:14:53 2021-03-15

```
Noise_28    0.577306
Noise_29    0.590815
Noise_30    0.147199
Noise_31    0.009771
Noise_32    0.625495
Noise_33    0.043671
Noise_34    0.914573
Noise_35    0.822432
Noise_36    0.405514
Noise_37    0.393812
Noise_38    0.769161
Noise_39    0.858692
Noise_40    0.461877
Noise_41    0.076768
Noise_42    0.700336
Noise_43    0.301304
Noise_44    0.381791
Noise_45    0.114720
Noise_46    0.870638
Noise_47    0.363271
```

In [68]:

```
# %%timeit

df_big['All_noises'] = df_big.apply(lambda row: np.mean(row[all_noise_columns]) > 0.1, ax

# 5.41 s ± 473 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

executed in 5.49s, finished 10:14:59 2021-03-15

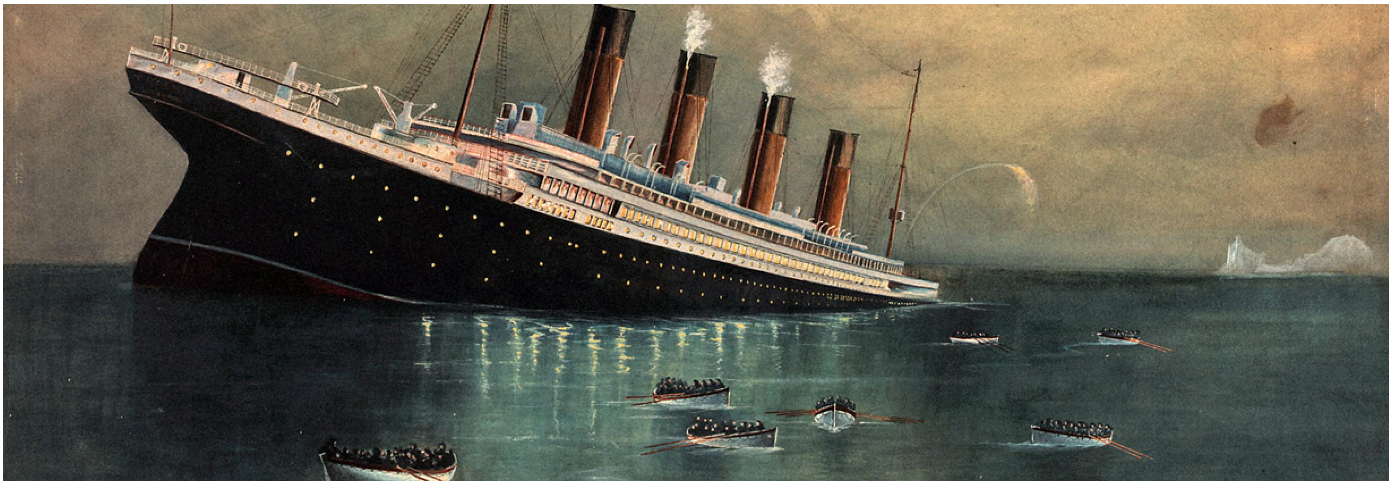
In [69]:

```
# %%timeit

for index in df_big.index:
    df_big.loc[index, 'All_noises'] = np.mean(df_big.loc[index, all_noise_columns]) > 0.1
#  9.37 s ± 317 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

executed in 12.4s, finished 10:15:11 2021-03-15

1.3 Tabele pivot



In [70]:

```
# Incarcarea datelor:

titanic = sns.load_dataset('titanic')
titanic.head()
```

executed in 45ms, finished 10:15:11 2021-03-15

Out[70]:

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male
0	0	3	male	22.0	1	0	7.2500	S	Third	man	True
1	1	1	female	38.0	1	0	71.2833	C	First	woman	False
2	1	3	female	26.0	0	0	7.9250	S	Third	woman	False
3	1	1	female	35.0	1	0	53.1000	S	First	woman	False
4	0	3	male	35.0	0	0	8.0500	S	Third	man	True

Pornim de la urmatoarea problema: care este procentul de femei si barbati supravietuitori? Diferentierea de gen se face dupa coloana 'sex', iar supravietuirea este in coloana 'survived':

In [71]:

```
titanic.groupby('sex')['survived'].mean()
```

executed in 13ms, finished 10:15:11 2021-03-15

Out[71]:

```
sex
female    0.742038
male      0.188908
Name: survived, dtype: float64
```

Mai departe, se cere determinarea distributiei pe gen si clasa imbarcare, folosind `groupby()` :

In [72]:

```
titanic.groupby(['sex', 'class'])['survived'].aggregate('mean').unstack()
```

executed in 27ms, finished 10:15:11 2021-03-15

Out[72]:

class	First	Second	Third
sex			
female	0.968085	0.921053	0.500000
male	0.368852	0.157407	0.135447

Acest tip de operatii (grupare dupa doua atribute, calcul de valori agregate) este des intalnit si se numeste pivotare. Pandas introduce suport nativ pentru pivotare, simplificand codul:

In [73]:

```
titanic.pivot_table('survived', index='sex', columns='class' )
```

executed in 35ms, finished 10:15:11 2021-03-15

Out[73]:

class	First	Second	Third
sex			
female	0.968085	0.921053	0.500000
male	0.368852	0.157407	0.135447

Se poate face pivotare pe mai mult de doua niveluri (mai sus au fost folosite: sex si class). De exemplu, varsta poate fi adaugata pentru analiza, persoane sub 18 ani (copii) si cei peste 18 (adulti). In primul pas se poate face impartirea persoanelor pe cele doua subintervale de varsta (<=18, >18) folosind `cut` :

In [74]:

```
age = pd.cut(titanic['age'], [0, 18, 1000], labels=['child', 'adult'])
age.head(15)
```

executed in 15ms, finished 10:15:11 2021-03-15

Out[74]:

```
0     adult
1     adult
2     adult
3     adult
4     adult
5       NaN
6     adult
7     child
8     adult
9     child
10    child
11    adult
12    adult
13    adult
14    child
Name: age, dtype: category
Categories (2, object): ['child' < 'adult']
```

In [75]:

```
titanic.pivot_table('survived', ['sex', age], 'class')
```

executed in 37ms, finished 10:15:11 2021-03-15

Out[75]:

	class	First	Second	Third
sex	**age**			
female	**child**	0.909091	1.000000	0.511628
	adult	0.972973	0.900000	0.423729
male	**child**	0.800000	0.600000	0.215686
	adult	0.375000	0.071429	0.133663

In [76]:

```
fare_split = pd.cut(titanic.fare, 2, labels=['cheap fare', 'expensive fare'])
```

executed in 7ms, finished 10:15:11 2021-03-15

In [77]:

```
fare_split
```

executed in 9ms, finished 10:15:11 2021-03-15

Out[77]:

```
0      cheap fare
1      cheap fare
2      cheap fare
3      cheap fare
4      cheap fare
          ...
886    cheap fare
887    cheap fare
888    cheap fare
889    cheap fare
890    cheap fare
Name: fare, Length: 891, dtype: category
Categories (2, object): ['cheap fare' < 'expensive fare']
```

In [78]:

```
titanic.pivot_table('survived', ['sex', age, fare_split], 'class')
```

executed in 54ms, finished 10:15:12 2021-03-15

Out[78]:

		class	First	Second	Third
sex	age	fare			
female	child	cheap fare	0.900000	1.000000	0.511628
		expensive fare	1.000000	NaN	NaN
	adult	cheap fare	0.971429	0.900000	0.423729
		expensive fare	1.000000	NaN	NaN
male	child	cheap fare	0.800000	0.600000	0.215686
	adult	cheap fare	0.369565	0.071429	0.133663
		expensive fare	0.500000	NaN	NaN

In [ ]: