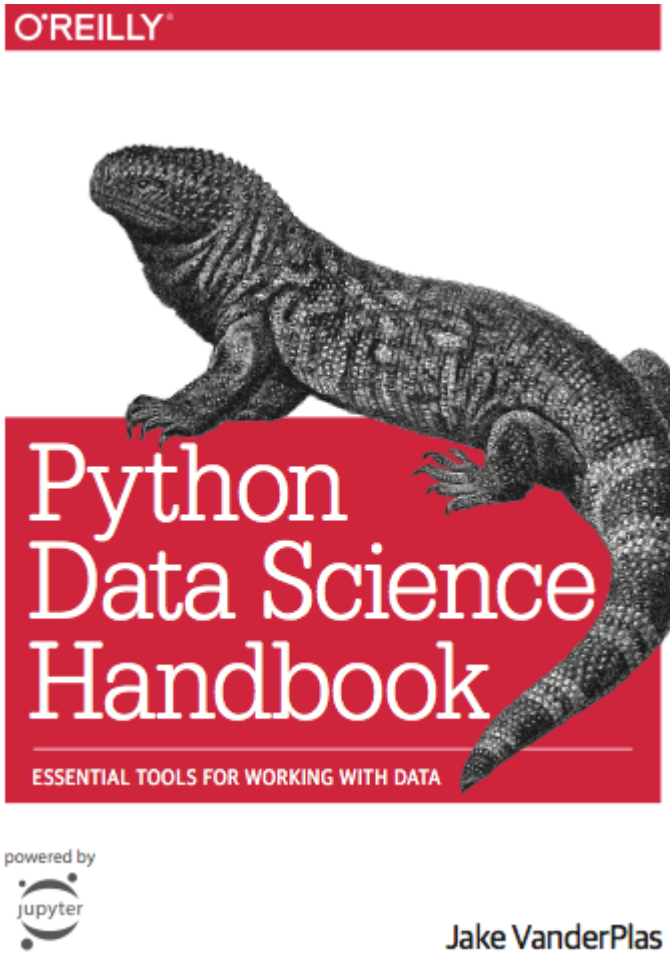
Curs 3: Pandas

Bibiligrafie: Python Data Science Handbook, Jake VanderPlas, disponibila pe pagina autorului (https://jakevdp.github.io/PythonDataScienceHandbook/).



Incarcarea datelor

In NumPy se pot manipula colectii matriceale de date, dar se presupune ca toate datele au acelasi tip:

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

tablou = np.array([[1, 2, 3], [3.5, 2, '10']])
tablou
```

Out[1]:
```
array([['1', '2', '3'],
       ['3.5', '2', '10']], dtype='<U32')
```

Pandas permite lucrul cu date in care coloanele pot avea tipuri diferite; prima coloana sa fie de tip intreg, al doilea - datetime etc.

In [2]:
```
import pandas as pd
pd.__version__
```

Out[2]:
```
'0.24.1'
```

Pandas Series

O serie Pandas este un vector unidimensional de date indexate.

In [3]:
```
data = pd.Series([0.25, 0.5, 0.75, 1.0])
data
```

Out[3]:
```
0    0.25
1    0.50
2    0.75
3    1.00
dtype: float64
```

Valorile se obtin folosind atributul values, returnand un NumPy array:

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

Out[4]:
```
array([0.25, 0.5 , 0.75, 1.  ])
```

Indexul unei serii se obtine prin atributul `index`. In cadrul unui obiect `Series` sau al unui `DataFrame` este util pentru adresarea datelor.

In [5]:
```
type(data.index)
```

Out[5]:
```
pandas.core.indexes.range.RangeIndex
```

Specificarea unui index pentru o serie se poate face la instantiere:

In [6]:
```
data = pd.Series([0.25, 0.5, 0.75, 1.0], index=['a', 'b', 'c', 'd'])
```

In [7]:
```
data.values
```

Out[7]:
```
array([0.25, 0.5 , 0.75, 1.  ])
```

In [8]:
```
data.index
```

Out[8]:
```
Index(['a', 'b', 'c', 'd'], dtype='object')
```

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

Out[9]:
```
0.5
```

Analogia dintre un obiect `Series` si un dictionar clasic Python poate fi speculata in crearea unui obiect Series plecand de la un dictionar:

In [10]:
```
geografie_populatie = {'Romania': 19638000, 'Franta': 67201000, 'Grecia': 1118
3957}
populatie = pd.Series(geografie_populatie)
populatie
```

Out[10]:
```
Romania    19638000
Franta     67201000
Grecia     11183957
dtype: int64
```

In [11]:
```
populatie.index
```

Out[11]:
```
Index(['Romania', 'Franta', 'Grecia'], dtype='object')
```

In [12]:
```
populatie['Grecia']
```

Out[12]:
```
11183957
```

In [13]:
```
# populatie['Germania']
# eroare: KeyError: 'Germania'
```

Daca nu se specifica un index la crearea unui obiect `Series`, atunci implicit acesta va fi format pe baza secventei de intregi 0, 1, 2, ...

Nu e obligatoriu ca o serie sa contina doar valori numerice:

In [14]:
```
s1 = pd.Series(['rosu', 'verde', 'galben', 'albastru'])
print(s1)
print('s1[2]=', s1[2])
```

```
0        rosu
1       verde
2      galben
3    albastru
dtype: object
s1[2]= galben
```

Datele unei serii se vad ca avand toate acelasi tip:

In [15]:
```
s_tip = pd.Series(['rosu', 1, 1.5])
s_tip
```

Out[15]:
```
0    rosu
1       1
2     1.5
dtype: object
```

Selectarea datelor in serii

Datele dintr-o serie pot fi referite prin intermediul indexului:

In [16]:
```
data = pd.Series(np.linspace(0, 75, 4), index=['a', 'b', 'c', 'd'])
print(data)
data['b']
```

```
a     0.0
b    25.0
c    50.0
d    75.0
dtype: float64
```

Out[16]:
```
25.0
```

Se poate face modificarea datelor dintr-o serie folosind indexul:

In [17]:
```
data['b'] = 300
print(data)
```

```
a      0.0
b    300.0
c     50.0
d     75.0
dtype: float64
```

Se poate folosi slicing:

In [18]:
```
data['a':'c']
```

Out[18]:
```
a      0.0
b    300.0
c     50.0
dtype: float64
```

sau se pot folosi liste de selectie:

In [19]:
```
data[['a', 'c', 'b', 'c']]
```

Out[19]:
```
a      0.0
c     50.0
b    300.0
c     50.0
dtype: float64
```

sau expresii logice:

In [20]:
```
data[(data > 30) & (data < 80)] #se remarca returnarea in rezultat a indicilor
 care satisfac proprietatea ceruta
```

Out[20]:
```
c    50.0
d    75.0
dtype: float64
```

Se prefera folosirea urmatoarelor atribute de indexare: `loc`, `iloc`. Indexarea prin `ix`, daca se regaseste prin tutoriale mai vechi, se considera a fi sursa de confuzie si se recomanda evitarea ei.

Atributul `loc` permite indicierea folosind valoarea de index.

In [21]:
```
data = pd.Series([1, 2, 3], index=['a', 'b', 'c'])

data
```

Out[21]:
```
a    1
b    2
c    3
dtype: int64
```

In [22]:
```
#cautare dupa index cu o singura valoare
data.loc['b']
```

Out[22]:
```
2
```

In [23]:
```
#cautare dupa index cu o doua valori. Lista interioara este folosita pentru a
 stoca o colectie de valori de indecsi.
data.loc[['a', 'c']]
```

Out[23]:
```
a    1
c    3
dtype: int64
```

Atributul `iloc` este folosit pentru a face referire la linii dupa pozitia (numarul) lor. Numerotarea incepe de la 0.

In [24]:
```
data.iloc[0]
```

Out[24]:
```
1
```

In [25]:
```
data.iloc[[0, 2]]
```

Out[25]:
```
a    1
c    3
dtype: int64
```

DataFrame

Un obiect `DataFrame` este o colectie de coloane de tip `Series`. Numarul de elemente din fiecare serie este acelasi.

In [26]:
```
df = pd.DataFrame([[1, 2, 3], [4, 5, 6]])
df
```

Out[26]:

	0	1	2
0	1	2	3
1	4	5	6

Se poate folosi un dictionar cu cheia avand nume de coloane, iar valorile de pe coloane ca liste:

In [27]:
```
df = pd.DataFrame({'Nume' : ['Ana', 'Dan', 'Maria'], 'Varsta': [20,30, 40]})
df
```

Out[27]:

	Nume	Varsta
0	Ana	20
1	Dan	30
2	Maria	40

In [28]:
```
geografie_suprafata = {'Romania': 238397, 'Franta': 640679, 'Grecia': 131957}

geografie_moneda = {'Romania': 'RON', 'Franta': 'EUR', 'Grecia': 'EUR'}

geografie = pd.DataFrame({'Populatie' : geografie_populatie, 'Suprafata' : geo
grafie_suprafata, 'Moneda' : geografie_moneda})

print(geografie)
```

```
         Populatie  Suprafata Moneda
Franta    67201000     640679    EUR
Grecia    11183957     131957    EUR
Romania   19638000     238397    RON
```

In [29]:
```
print(geografie.index)
```

```
Index(['Franta', 'Grecia', 'Romania'], dtype='object')
```

Atributul `columns` da lista de coloane din obiectul `DataFrame`:

In [30]:
```
geografie.columns
```

Out[30]:
```
Index(['Populatie', 'Suprafata', 'Moneda'], dtype='object')
```

Referirea la o serie care compune o coloana din DataFrame se face astfel

In [31]:
```
print(geografie['Populatie'])
print('**********')
print(type(geografie['Populatie']))
```

```
Franta     67201000
Grecia     11183957
Romania    19638000
Name: Populatie, dtype: int64
**********
<class 'pandas.core.series.Series'>
```

Crearea unui obiect DataFrame se poate face pornind si de la o singura serie:

In [32]:
```
mydf = pd.DataFrame([1, 2, 3], columns=['values'])
mydf
```

Out[32]:

	values
0	1
1	2
2	3

... sau se poate crea pornind de la o lista de dictionare:

In [33]:
```
data
```

Out[33]:
```
a    1
b    2
c    3
dtype: int64
```

In [34]:
```
data = [{'a': i, 'b': 2 * i} for i in range(3)]
pd.DataFrame(data)
```

Out[34]:

	a	b
0	0	0
1	1	2
2	2	4

Daca lipsesc chei din vreunul din dictionare, respectiva valoare se va umple cu `NaN`.

In [35]:
```
pd.DataFrame([{'a': 1, 'b': 2}, {'b': 3, 'c': 4}])
```

Out[35]:

	a	b	c
0	1.0	2	NaN
1	NaN	3	4.0

In [36]:
```
pd.DataFrame([{'a': 'aaa', 'b': 'bbb'}, {'b': 'bbb2', 'c': 'cccc'}])
```

Out[36]:

	a	b	c
0	aaa	bbb	NaN
1	NaN	bbb2	cccc

Instantierea unui DataFrame se poate face si de la un NumPy array:

In [37]:
```
pd.DataFrame(np.random.rand(3, 2), columns=['Col1', 'Col2'], index=['a', 'b',
'c'])
```

Out[37]:

	Col1	Col2
a	0.289623	0.809251
b	0.874043	0.579393
c	0.142901	0.199167

Se poate adauga o coloana noua la un DataFrame, similar cu adaugarea unui element (cheie, valoare) la un dictionar:

In [38]:
```
geografie['Densitatea populatiei'] = geografie['Populatie'] / geografie['Suprafata']

geografie
```

Out[38]:

	Populatie	Suprafata	Moneda	Densitatea populatiei
Franta	67201000	640679	EUR	104.890280
Grecia	11183957	131957	EUR	84.754556
Romania	19638000	238397	RON	82.375198

Un obiect DataFrame poate fi transpus cu atributul `T`:

In [39]:
```
geografie.T
```

Out[39]:

	Franta	Grecia	Romania
Populatie	67201000	11183957	19638000
Suprafata	640679	131957	238397
Moneda	EUR	EUR	RON
Densitatea populatiei	104.89	84.7546	82.3752

Selectarea datelor intr-un `DataFrame`

S-a demonstrat posibilitatea de referire dupa numele de coloana:

In [40]:
```
print(geografie)
```

```
          Populatie  Suprafata Moneda  Densitatea populatiei
Franta     67201000     640679    EUR             104.890280
Grecia     11183957     131957    EUR              84.754556
Romania    19638000     238397    RON              82.375198
```

In [41]:
```
print(geografie['Moneda'])
```

```
Franta     EUR
Grecia     EUR
Romania    RON
Name: Moneda, dtype: object
```

Daca numele unei coloane este un string fara spatii, se poate folosi acesta ca un atribut:

In [42]:
```
geografie.Moneda
```

Out[42]:
```
Franta     EUR
Grecia     EUR
Romania    RON
Name: Moneda, dtype: object
```

Se poate face referire la o coloana dupa indicele ei, indirect:

In [43]:
```
geografie[geografie.columns[0]]
```

Out[43]:
```
Franta     67201000
Grecia     11183957
Romania    19638000
Name: Populatie, dtype: int64
```

Pentru cazul in care un DataFrame nu are nume de coloana, else sunt implicit intregii 0, 1, ... si se pot folosi pentru selectarea de coloana folosind paranteze drepte:

In [44]:
```
my_data = pd.DataFrame(np.random.rand(3, 4))

my_data
```

Out[44]:

	0	1	2	3
0	0.534023	0.522727	0.676544	0.656179
1	0.882820	0.927845	0.135408	0.803261
2	0.196228	0.864861	0.449244	0.597577

In [45]:
```
my_data[0]
```

Out[45]:
```
0    0.534023
1    0.882820
2    0.196228
Name: 0, dtype: float64
```

Atributul `values` returneaza un obiect ndarray continand valori. Tipul unui ndarray este cel mai specializat tip de date care poate sa contina valorile din DataFrame:

In [46]:
```
#afisare ndarray si tip pentru my_data.values
print(my_data.values)
print(my_data.values.dtype)
```

```
[[0.53402313 0.52272693 0.67654353 0.65617851]
 [0.88281962 0.92784455 0.13540805 0.80326064]
 [0.19622782 0.86486064 0.44924447 0.59757736]]
float64
```

In [47]:
```
#afisare ndarray si tip pentru geografie.values
print(geografie.values)
print(geografie.values.dtype)
```

```
[[67201000 640679 'EUR' 104.89028046806591]
 [11183957 131957 'EUR' 84.75455640852702]
 [19638000 238397 'RON' 82.37519767446739]]
object
```

Indexarea cu `iloc` in cazul unui obiect `DataFrame` permite precizarea a doua valori: prima reprezinta linia si al doilea coloana, numerotate de la 0. Pentru linie si coloana se poate folosi si slicing:

In [48]:
```
print(geografie)

geografie.iloc[0:2, 2:4]
```

```
         Populatie  Suprafata Moneda  Densitatea populatiei
Franta    67201000     640679    EUR             104.890280
Grecia    11183957     131957    EUR              84.754556
Romania   19638000     238397    RON              82.375198
```

Out[48]:

	Moneda	Densitatea populatiei
Franta	EUR	104.890280
Grecia	EUR	84.754556

Indexarea cu `loc` permite precizarea valorilor de indice si respectiv nume de coloana:

In [49]:
```
print(geografie)

geografie.loc[['Franta', 'Romania'], 'Populatie':'Densitatea populatiei']
```

```
         Populatie  Suprafata Moneda  Densitatea populatiei
Franta    67201000     640679    EUR             104.890280
Grecia    11183957     131957    EUR              84.754556
Romania   19638000     238397    RON              82.375198
```

Out[49]:

	Populatie	Suprafata	Moneda	Densitatea populatiei
Franta	67201000	640679	EUR	104.890280
Romania	19638000	238397	RON	82.375198

Se permite folosirea de expresii de filtrare à la NumPy:

In [50]:
```
geografie.loc[geografie['Densitatea populatiei'] > 83, ['Populatie', 'Moneda']]
```

Out[50]:

	Populatie	Moneda
Franta	67201000	EUR
Grecia	11183957	EUR

Folosind indicierea, se pot modifica valorile dintr-un `DataFrame`:

In [51]:
```
#Modificarea populatiei Greciei cu iloc
geografie.iloc[1, 1] = 12000000
print(geografie)
```

```
         Populatie  Suprafata Moneda  Densitatea populatiei
Franta    67201000     640679    EUR             104.890280
Grecia    11183957   12000000    EUR              84.754556
Romania   19638000     238397    RON              82.375198
```

In [52]:
```
#Modificarea populatiei Greciei cu loc
geografie.loc['Grecia', 'Populatie'] = 11183957
print(geografie)
```

```
         Populatie  Suprafata Moneda  Densitatea populatiei
Franta    67201000     640679    EUR             104.890280
Grecia    11183957   12000000    EUR              84.754556
Romania   19638000     238397    RON              82.375198
```

Precizari:

1. daca se foloseste un singur indice la un DataFrame, atunci se considera ca se face referire la coloana:

   ```
   geografie['Moneda']
   ```

2. daca se foloseste slicing, acesta se refera la liniile (indexul) din DataFrame:

   ```
   geografie['Franta':'Romania']
   ```

3. operatiile logice se considera ca refera de asemenea linii din DataFrame:

   ```
   geografie[geografie['Densitatea populatiei'] > 83]
   ```

In [53]:
```
geografie[geografie['Densitatea populatiei'] > 83]
```

Out[53]:

	Populatie	Suprafata	Moneda	Densitatea populatiei
Franta	67201000	640679	EUR	104.890280
Grecia	11183957	12000000	EUR	84.754556

Operarea pe date

Se pot aplica functii NumPy peste obiecte Series si DataFrame. Rezultatul este de acelasi tip ca obiectul peste care se aplica iar indicii se pastreaza:

In [54]:
```
ser = pd.Series(np.random.randint(low=0, high=10, size=(5)), index=['a', 'b',
'c', 'd', 'e'])
ser
```

Out[54]:
```
a    1
b    5
c    8
d    5
e    3
dtype: int32
```

In [55]:
```
np.exp(ser)
```

Out[55]:
```
a       2.718282
b     148.413159
c    2980.957987
d     148.413159
e      20.085537
dtype: float64
```

In [56]:
```
my_df = pd.DataFrame(data=np.random.randint(low=0, high=10, size=(3, 4)), \
                     columns=['Sunday', 'Monday', 'Tuesday', 'Wednesday'], \
                    index=['a', 'b', 'c'])
print('Originar:', my_df)
print('Transformat:', np.exp(my_df))
```

```
Originar:    Sunday  Monday  Tuesday  Wednesday
a       3       3        6          0
b       4       6        0          8
c       0       1        1          2
Transformat:        Sunday      Monday     Tuesday    Wednesday
a  20.085537   20.085537  403.428793     1.000000
b  54.598150  403.428793    1.000000  2980.957987
c   1.000000    2.718282    2.718282     7.389056
```

Pentru functii binare se face alinierea obiectelor Series sau DataFrame dupa indexul lor. Aceasta poate duce la operare cu valori NaN si in consecinta obtinere de valori NaN.

In [57]:
```
area = pd.Series({'Alaska': 1723337, 'Texas': 695662, 'California': 423967}, n
ame='area')
population = pd.Series({'California': 38332521, 'Texas': 26448193, 'New York':
 19651127}, name='population')
```

In [58]:
```
population / area
```

Out[58]:
```
Alaska              NaN
California    90.413926
New York            NaN
Texas         38.018740
dtype: float64
```

In cazul unui DataFrame, alinierea se face atat pentru coloane, cat si pentru indecsii folositi la linii:

In [59]:
```
A = pd.DataFrame(data=np.random.randint(0, 10, (2, 3)), columns=list('ABC'))
B = pd.DataFrame(data=np.random.randint(0, 10, (3, 2)), columns=list('BA'))

A
```

Out[59]:

	A	B	C
0	5	2	7
1	1	4	4

In [60]:
```
B
```

Out[60]:

	B	A
0	9	5
1	9	6
2	6	5

In [61]:
```
A + B
```

Out[61]:

	A	B	C
0	10.0	11.0	NaN
1	7.0	13.0	NaN
2	NaN	NaN	NaN

Daca se doreste umplerea valorilor NaN cu altceva, se poate specifica parametrul fill_value pentru functii care implementeaza operatiile aritmetice:

Operator	Metoda Pandas
+	`add()`
-	`sub()`, `substract()`
*	`mul()`, `multiply()`
/	`truediv()`, `div()`, `divide()`
//	`floordiv()`
%	`mod()`
**	`pow()`

Daca ambele pozitii au valori lipsa (NaN), atunci valoarea finala va fi si ea lipsa (https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.add.html).

Exemplu:

In [62]:
```
A
```

Out[62]:

	A	B	C
0	5	2	7
1	1	4	4

In [63]:
```
B
```

Out[63]:

	B	A
0	9	5
1	9	6
2	6	5

In [64]:
```
A.add(B, fill_value=0)
```

Out[64]:

	A	B	C
0	10.0	11.0	7.0
1	7.0	13.0	4.0
2	5.0	6.0	NaN

Valori lipsa

Pentru cazul in care valorile dintr-o coloana a unui obiect DataFrame sunt de tip numeric, valorile lipsa se reprezinta prin NaN - care e suportat doar de tipurile in virgula mobila, nu si de intregi; aceasta din ultima observatie arata ca numerele intregi sunt convertite la floating point daca intr-o lista care le contine se afla si valori lipsa:

In [65]:
```
my_series = pd.Series([1, 2, 3, None, 5], name='my_series')
#echivalent:
my_series = pd.Series([1, 2, 3, np.NaN, 5], name='my_series')
my_series
```

Out[65]:
```
0    1.0
1    2.0
2    3.0
3    NaN
4    5.0
Name: my_series, dtype: float64
```

Functiile care se pot folosi pentru un DataFrame pentru a operare cu valori lipsa sunt:

In [66]:
```
df = pd.DataFrame([[1, 2, np.NaN], [np.NAN, 10, 20]])
df
```

Out[66]:

	0	1	2
0	1.0	2	NaN
1	NaN	10	20.0

`isnull()` - returneaza o masca de valori logice, cu `True` (`False`) pentru pozitiile unde se afla valori nule (respectiv: nenule); nul = valoare lipsa.

In [67]:
```
df.isnull()
```

Out[67]:

	0	1	2
0	False	False	True
1	True	False	False

`notnull()` - opusul functiei precedente

`dropna()` - returneaza o varianta filtrata a obiectuilui DataFrame. E posibil sa duca la un DataFrame gol.

In [68]:
```
df.dropna()
```

Out[68]:

	0	1	2

In [69]:
```
df.iloc[0] = [3, 4, 5]
print(df)
df.dropna()
```

```
     0   1     2
0  3.0   4   5.0
1  NaN  10  20.0
```

Out[69]:

	0	1	2
0	3.0	4	5.0

`fillna()` umple valorile lipsa dupa o anumita politica:

In [70]:
```
df = pd.DataFrame([[1, 2, np.NaN], [np.NAN, 10, 20]])
df
```

Out[70]:

	0	1	2
0	1.0	2	NaN
1	NaN	10	20.0

In [71]:
```
#umplere de NaNuri cu valoare constanta
df2 = df.fillna(value = 100)
df2
```

Out[71]:

	0	1	2
0	1.0	2	100.0
1	100.0	10	20.0

In [72]:
```
np.random.randn(5, 3)
```

Out[72]:
```
array([[ 0.66555778,  1.91159754, -1.53636626],
       [ 0.86838451, -0.12954157,  0.25460434],
       [ 0.14968894,  0.68119839,  1.81349619],
       [-0.39974136,  1.37547189, -1.09587758],
       [ 0.9386557 ,  1.89257724, -0.58496642]])
```

In [73]:
```
#umplere de NaNuri cu media pe coloana corespunzatoare
df = pd.DataFrame(data = np.random.randn(5, 3), columns=['A', 'B', 'C'])
df.iloc[0, 2] = df.iloc[1, 1] = df.iloc[2, 0] = df.iloc[4, 1] = np.NAN
df
```

Out[73]:

	A	B	C
0	-0.696671	0.235958	NaN
1	-0.775686	NaN	-0.279032
2	NaN	0.499031	1.535760
3	0.881599	1.901484	-0.537505
4	-0.563059	NaN	0.753198

In [74]:
```
#calcul medie pe coloana
df.mean(axis=0)
```

Out[74]:
```
A   -0.288454
B    0.878824
C    0.368105
dtype: float64
```

In [75]:
```
df3 = df.fillna(df.mean(axis=0))
df3
```

Out[75]:

	A	B	C
0	-0.696671	0.235958	0.368105
1	-0.775686	0.878824	-0.279032
2	-0.288454	0.499031	1.535760
3	0.881599	1.901484	-0.537505
4	-0.563059	0.878824	0.753198

Exista un parametru al functiei `fillna()` care permite umplerea valorilor lipsa prin copiere (https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.fillna.html):

In [76]:
```
my_ds = pd.Series(np.arange(0, 30))
my_ds[1:-1:4] = np.NaN
my_ds
```

Out[76]:
```
0      0.0
1      NaN
2      2.0
3      3.0
4      4.0
5      NaN
6      6.0
7      7.0
8      8.0
9      NaN
10    10.0
11    11.0
12    12.0
13     NaN
14    14.0
15    15.0
16    16.0
17     NaN
18    18.0
19    19.0
20    20.0
21     NaN
22    22.0
23    23.0
24    24.0
25     NaN
26    26.0
27    27.0
28    28.0
29    29.0
dtype: float64
```

In [77]:
```
# copierea ultimei valori non-null
my_ds_filled_1 = my_ds.fillna(method='ffill')
my_ds_filled_1
```

Out[77]:
```
0      0.0
1      0.0
2      2.0
3      3.0
4      4.0
5      4.0
6      6.0
7      7.0
8      8.0
9      8.0
10    10.0
11    11.0
12    12.0
13    12.0
14    14.0
15    15.0
16    16.0
17    16.0
18    18.0
19    19.0
20    20.0
21    20.0
22    22.0
23    23.0
24    24.0
25    24.0
26    26.0
27    27.0
28    28.0
29    29.0
dtype: float64
```

In [78]:
```
# copierea inapoi a urmatoarei valori non-null
my_ds_filled_2 = my_ds.fillna(method='bfill')
my_ds_filled_2
```

Out[78]:
```
0      0.0
1      2.0
2      2.0
3      3.0
4      4.0
5      6.0
6      6.0
7      7.0
8      8.0
9     10.0
10    10.0
11    11.0
12    12.0
13    14.0
14    14.0
15    15.0
16    16.0
17    18.0
18    18.0
19    19.0
20    20.0
21    22.0
22    22.0
23    23.0
24    24.0
25    26.0
26    26.0
27    27.0
28    28.0
29    29.0
dtype: float64
```

Pentru DataFrame, procesul este similar. Se poate specifica argumentul axis care spune daca procesarea se face pe linii sau pe coloane:

In [79]:
```
df = pd.DataFrame([[1, np.NAN, 2, np.NAN], [2, 3, 5, np.NaN], [np.NaN, 4, 6, n
p.NaN]])
df
```

Out[79]:

	0	1	2	3
0	1.0	NaN	2	NaN
1	2.0	3.0	5	NaN
2	NaN	4.0	6	NaN

In [80]:
```
#Umplere, prin parcurgere pe linii
df.fillna(method='ffill', axis = 1)
```

Out[80]:

	0	1	2	3
0	1.0	1.0	2.0	2.0
1	2.0	3.0	5.0	5.0
2	NaN	4.0	6.0	6.0

In [81]:
```
#Umplere, prin parcurgere pe fiecare coloana
df.fillna(method='ffill', axis = 0)
```

Out[81]:

	0	1	2	3
0	1.0	NaN	2	NaN
1	2.0	3.0	5	NaN
2	2.0	4.0	6	NaN

Combinarea de obiecte Series si DataFrame

Cea mai simpla operatie este de concatenare:

In [82]:
```
ser1 = pd.Series(['A', 'B', 'C'], index=[1, 2, 3])
ser2 = pd.Series(['D', 'E', 'F'], index=[4, 5, 6])
pd.concat([ser1, ser2])
```

Out[82]:
```
1    A
2    B
3    C
4    D
5    E
6    F
dtype: object
```

Pentru cazul in care valori de index se regasesc in ambele serii de date, indexul se va repeta:

In [83]:
```
ser1 = pd.Series(['A', 'B', 'C'], index=[1, 2, 3])
ser2 = pd.Series(['D', 'E', 'F'], index=[3, 4, 5])
ser_concat = pd.concat([ser1, ser2])
ser_concat
```

Out[83]:
```
1    A
2    B
3    C
3    D
4    E
5    F
dtype: object
```

In [84]:
```
ser_concat.loc[3]
```

Out[84]:
```
3    C
3    D
dtype: object
```

Pentru cazul in care se doreste verificarea faptului ca indecsii sunt unici, se poate folosi parametrul `verify_integrity`:

In [85]:
```
try:
    ser_concat = pd.concat([ser1, ser2], verify_integrity=True)
except ValueError as e:
    print('Value error', e)
```

```
Value error Indexes have overlapping values: Int64Index([3], dtype='int64')
```

Pentru concatenarea de obiecte `DataFrame` care au acelasi set de coloane (pentru moment):

In [86]:
```
#sursa: ref 1 din Curs 1
def make_df(cols, ind):
    """Quickly make a DataFrame"""
    data = {c: [str(c) + str(i) for i in ind] for c in cols}
    return pd.DataFrame(data, ind)
```

In [87]:
```
df1 = make_df('AB', [1, 2])
df2 = make_df('AB', [3, 4])
print(df1); print(df2);
```

```
    A   B
1  A1  B1
2  A2  B2
    A   B
3  A3  B3
4  A4  B4
```

In [88]:
```
#concatenare simpla
pd.concat([df1, df2])
```

Out[88]:

	A	B
1	A1	B1
2	A2	B2
3	A3	B3
4	A4	B4

Concatenarea se poate face si pe orizontala:

In [89]:
```
df3 = make_df('AB', [0, 1])
df4 = make_df('CD', [0, 1])
print(df3); print(df4);
```

```
    A   B
0  A0  B0
1  A1  B1
    C   D
0  C0  D0
1  C1  D1
```

In [90]:
```
#concatenare pe axa 1
pd.concat([df3, df4], axis=1)
```

Out[90]:

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1

Pentru indici duplicati, comportamentul e la fel ca la `Serie`: se pastreaza duplicatele si datele corespunzatoare:

In [91]:
```
x = make_df('AB', [0, 1])
y = make_df('AB', [0, 1])
print(x); print(y);
```

```
    A   B
0  A0  B0
1  A1  B1
    A   B
0  A0  B0
1  A1  B1
```

In [92]:
```
print(pd.concat([x, y]))
```

```
    A   B
0  A0  B0
1  A1  B1
0  A0  B0
1  A1  B1
```

In [93]:
```
try:
    df_concat = pd.concat([x, y], verify_integrity=True)
except ValueError as e:
    print('Value error', e)
```

```
Value error Indexes have overlapping values: Int64Index([0, 1], dtype='int6
4')
```

Daca se doreste ignorarea indecsilor, se poate folosi indicatorul `ignore_index`:

In [94]:
```
df_concat = pd.concat([x, y], ignore_index=True)
```

Pentru cazul in care obiectele `DataFrame` nu au exact aceleasi coloane, concatenarea poate duce la rezultate de forma:

In [95]:
```
df5 = make_df('ABC', [1, 2])
df6 = make_df('BCD', [3, 4])
print(df5); print(df6);
```

```
    A   B   C
1  A1  B1  C1
2  A2  B2  C2
    B   C   D
3  B3  C3  D3
4  B4  C4  D4
```

In [96]:
```
print(pd.concat([df5, df6]))
```

```
     A   B   C    D
1   A1  B1  C1  NaN
2   A2  B2  C2  NaN
3  NaN  B3  C3   D3
4  NaN  B4  C4   D4
```

```
C:\Anaconda3\lib\site-packages\ipykernel_launcher.py:1: FutureWarning: Sortin
g because non-concatenation axis is not aligned. A future version
of pandas will change to not sort by default.

To accept the future behavior, pass 'sort=False'.

To retain the current behavior and silence the warning, pass 'sort=True'.

  """Entry point for launching an IPython kernel.
```

De regula se vrea operatia de concatenare (join) pe obiectele DataFrame cu coloane diferite. O prima varianta este pastrarea doar a coloanelor partajate, ceea ce in Pandas este vazut ca un inner join (se remarca o necorespondenta cu terminologia din limbajul SQL):

In [97]:
```
print(df5); print(df6);
```

```
    A   B   C
1  A1  B1  C1
2  A2  B2  C2
    B   C   D
3  B3  C3  D3
4  B4  C4  D4
```

In [98]:
```
#concatenare cu inner join
pd.concat([df5, df6], join='inner')
```

Out[98]:

	B	C
1	B1	C1
2	B2	C2
3	B3	C3
4	B4	C4

Alta varianta este specificarea explicita a coloanelor care rezista in urma concatenarii, via parametrul `join_axes`:

In [99]:
```
print(df5); print(df6);
```

```
    A   B   C
1  A1  B1  C1
2  A2  B2  C2
    B   C   D
3  B3  C3  D3
4  B4  C4  D4
```

In [100]:
```
pd.concat([df5, df6], join_axes=[df5.columns])
```

Out[100]:

	A	B	C
1	A1	B1	C1
2	A2	B2	C2
3	NaN	B3	C3
4	NaN	B4	C4

Pentru implementarea de jonctiuni à la SQL se foloseste metoda `merge`. Ce mai simpla este inner join: rezulta liniile din obiectele DataFrame care au corespondent in ambele parti:

In [101]:
```
df1 = pd.DataFrame({'employee': ['Bob', 'Jake', 'Lisa', 'Sue'],
'group': ['Accounting', 'Engineering', 'Engineering', 'HR']})
df2 = pd.DataFrame({'employee': ['Lisa', 'Bob', 'Jake', 'Sue'],
'hire_date': [2004, 2008, 2012, 2014]})
```

In [102]:
```
df3=pd.merge(df1, df2)
df3
```

Out[102]:

	employee	group	hire_date
0	Bob	Accounting	2008
1	Jake	Engineering	2012
2	Lisa	Engineering	2004
3	Sue	HR	2014

In [103]:
```
df3 = pd.DataFrame({'employee': ['Jake', 'Lisa', 'Sue'],
'group': ['Engineering', 'Engineering', 'HR']})
df4 = pd.DataFrame({'employee': ['Bob', 'Jake', 'Sue'],
'hire_date': [2008, 2012, 2014]})

#demo inner join: raman dar 2 linii dupa jonctiune
pd.merge(df3, df4)
```

Out[103]:

	employee	group	hire_date
0	Jake	Engineering	2012
1	Sue	HR	2014

Se pot face asa-numite jonctiuni `many-to-one`, dar care nu sunt decat inner join. Mentionam si exemplificam insa pentru terminologie:

In [104]:
```
df4 = pd.DataFrame({'group': ['Accounting', 'Engineering', 'HR'],
'supervisor': ['Carly', 'Guido', 'Steve']})

print(df3)
print(df4)
```

```
  employee        group
0     Jake  Engineering
1     Lisa  Engineering
2      Sue           HR
         group supervisor
0   Accounting      Carly
1  Engineering      Guido
2           HR      Steve
```

In [105]:
```
pd.merge(df3, df4)
```

Out[105]:

	employee	group	supervisor
0	Jake	Engineering	Guido
1	Lisa	Engineering	Guido
2	Sue	HR	Steve

Asa-numite jonctiuni *many-to-many* se obtin pentru cazul in care coloana dupa care se face jonctiunea contine duplicate:

In [106]:
```
df5 = pd.DataFrame({'group': ['Accounting', 'Accounting',
'Engineering', 'Engineering', 'HR', 'HR'],
'skills': ['math', 'spreadsheets', 'coding', 'linux',
'spreadsheets', 'organization']})
print(df1)
print(df5)
```

```
  employee        group
0      Bob   Accounting
1     Jake  Engineering
2     Lisa  Engineering
3      Sue           HR
         group        skills
0   Accounting          math
1   Accounting  spreadsheets
2  Engineering        coding
3  Engineering         linux
4           HR  spreadsheets
5           HR  organization
```

In [107]:
```
print(pd.merge(df1, df5))
```

```
  employee        group        skills
0      Bob   Accounting          math
1      Bob   Accounting  spreadsheets
2     Jake  Engineering        coding
3     Jake  Engineering         linux
4     Lisa  Engineering        coding
5     Lisa  Engineering         linux
6      Sue           HR  spreadsheets
7      Sue           HR  organization
```