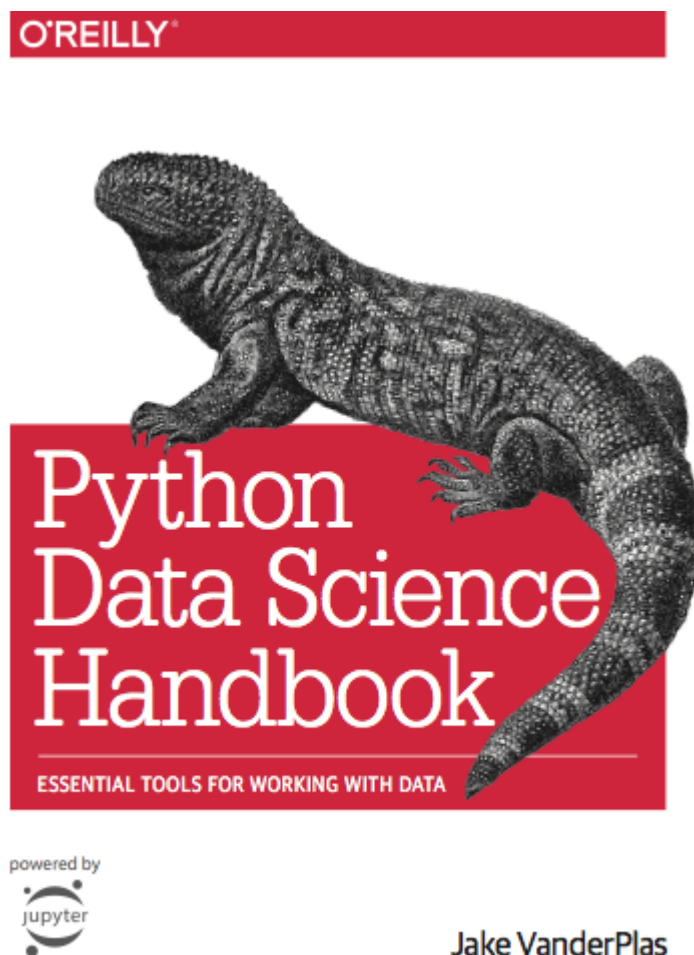


Curs 3: Pandas

Bibilografie: Python Data Science Handbook, Jake VanderPlas, disponibilă [pe pagina autorului](https://jakevdp.github.io/PythonDataScienceHandbook/) (<https://jakevdp.github.io/PythonDataScienceHandbook/>).



Incarcarea datelor

În NumPy se pot manipula colecții matriceale de date, dar se presupune că toate datele au același 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]: '1.0.1'
```

Un exemplu de set de date care combina tipuri: intregi, reale si categoriale (caracter) este [Setul Abalone](http://archive.ics.uci.edu/ml/datasets/Abalone) (<http://archive.ics.uci.edu/ml/datasets/Abalone>). E utila deci existenta tipurilor de tabel care permit coloane de tip eterogen.

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
```

```
Out[7]: a    0.25  
b    0.50  
c    0.75  
d    1.00  
dtype: float64
```

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

```
Out[8]: array([0.25, 0.5 , 0.75, 1.  ])
```

```
In [9]: data.index
```

```
Out[9]: Index(['a', 'b', 'c', 'd'], dtype='object')
```

```
In [10]: data['b']
```

```
Out[10]: 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 [11]: geografie_populatie = {'Romania': 19638000, 'Franta': 67201000, 'Grecia': 1118  
3957}  
populatie = pd.Series(geografie_populatie)  
populatie
```

```
Out[11]: Romania    19638000  
Franta    67201000  
Grecia    11183957  
dtype: int64
```

```
In [12]: populatie.index
```

```
Out[12]: Index(['Romania', 'Franta', 'Grecia'], dtype='object')
```

```
In [13]: populatie['Grecia']
```

```
Out[13]: 11183957
```

```
In [14]: # 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 [15]: 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 [16]: s_tip = pd.Series(['rosu', 1, 1.5])
s_tip
```

```
Out[16]: 0      rosu
1         1
2        1.5
dtype: object
```

Selectarea datelor in serii

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

```
In [17]: 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[17]: 25.0
```

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

```
In [18]: data['b'] = 300
print(data)

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

Se poate folosi slicing, iar aici, spre deosebire de slicing-ul din NumPy si Python, se ia inclusiv capatul din dreapta al indicilor:

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

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

sau se pot folosi liste de selectie:

```
In [20]: data[['a', 'c', 'b', 'c']]
```

```
Out[20]: a      0.0  
         c     50.0  
         b    300.0  
         c     50.0  
         dtype: float64
```

sau expresii logice:

```
In [21]: data[(data > 30) & (data < 80)] #se remarca returnarea in rezultat a indicilor  
                                             care satisfac proprietatea ceruta
```

```
Out[21]: c     50.0  
         d     75.0  
         dtype: float64
```

Se prefera folosirea urmatoarelor attribute 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 [22]: data = pd.Series([1, 2, 3], index=['a', 'b', 'c'])  
  
data
```

```
Out[22]: a      1  
         b      2  
         c      3  
         dtype: int64
```

```
In [23]: #cautare dupa index cu o singura valoare  
data.loc['b']
```

```
Out[23]: 2
```

```
In [24]: #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[24]: 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 [25]: data.iloc[0]
```

```
Out[25]: 1
```

```
In [26]: data.iloc[[0, 2]]
```

```
Out[26]: 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 [27]: df = pd.DataFrame([[1, 2, 3], [4, 5, 6]])
         df
```

```
Out[27]:
```

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

```
Out[28]:
```

	Nume	Varsta
0	Ana	20
1	Dan	30
2	Maria	40

```
In [29]: geografie_suprafata = {'Romania': 238397, 'Franta': 640679, 'Grecia': 131957}

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

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

print(geografie)
```

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

```
In [30]: print(geografie.index)

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

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

```
In [31]: geografie.columns
```

```
Out[31]: Index(['Populatie', 'Suprafata', 'Moneda'], dtype='object')
```

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

```
In [32]: print(geografie['Populatie'])
print('*****')
print(type(geografie['Populatie']))
```

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

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

```
In [33]: mydf = pd.DataFrame([1, 2, 3], columns=['values'])
mydf
```

```
Out[33]:
```

	values
0	1
1	2
2	3

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

```
In [34]: data
```

```
Out[34]: a    1  
         b    2  
         c    3  
         dtype: int64
```

```
In [35]: data = [{'a': i, 'b': 2 * i} for i in range(3)]  
         print(data)  
         pd.DataFrame(data)
```

```
[{'a': 0, 'b': 0}, {'a': 1, 'b': 2}, {'a': 2, 'b': 4}]
```

```
Out[35]:
```

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

```
Out[36]:
```

	a	b	c
0	1.0	2	NaN
1	NaN	3	4.0

```
In [37]: pd.DataFrame([{'a': 'aaa', 'b': 'bbb'}, {'b': 'bbb2', 'c': 'cccc'}])
```

```
Out[37]:
```

	a	b	c
0	aaa	bbb	NaN
1	NaN	bbb2	cccc

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


```
In [38]: pd.DataFrame(np.random.rand(3, 2), columns=['Col1', 'Col2'], index=['a', 'b', 'c'])
```

Out[38]:

	Col1	Col2
a	0.142334	0.062504
b	0.911834	0.870093
c	0.973749	0.970177

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

```
In [39]: geografie['Densitatea populatiei'] = geografie['Populatie'] / geografie['Suprafata']  
geografie
```

Out[39]:

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

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

```
In [40]: geografie.T
```

Out[40]:

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

Selectarea datelor intr-un DataFrame

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

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

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

```
In [42]: print(geografie['Moneda'])
```

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

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

```
In [43]: geografie.Moneda
```

```
Out[43]: Romania    RON
Franta     EUR
Grecia     EUR
Name: Moneda, dtype: object
```

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

```
In [44]: geografie[geografie.columns[0]]
```

```
Out[44]: Romania    19638000
Franta     67201000
Grecia     11183957
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 [45]: my_data = pd.DataFrame(np.random.rand(3, 4))

my_data
```

```
Out[45]:
```

	0	1	2	3
0	0.704184	0.020708	0.905108	0.975180
1	0.727061	0.347092	0.728168	0.988484
2	0.816263	0.220023	0.374993	0.215326

```
In [46]: my_data[0]
```

```
Out[46]: 0    0.704184
          1    0.727061
          2    0.816263
          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 [47]: #afisare ndarray si tip pentru my_data.values
print(my_data.values)
print(my_data.values.dtype)
```

```
[[0.7041844  0.02070831 0.905108   0.97518007]
 [0.72706126 0.34709152 0.72816776 0.98848392]
 [0.81626252 0.22002284 0.37499281 0.21532618]]
float64
```

```
In [48]: #afisare ndarray si tip pentru geografie.values
print(geografie.values)
print(geografie.values.dtype)
```

```
[[19638000 238397 'RON' 82.37519767446739]
 [67201000 640679 'EUR' 104.89028046806591]
 [11183957 131957 'EUR' 84.75455640852702]]
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, **cu observatia esentiala ca spre deosebire de Python si NumPy, se include si capatul din dreapta al oricarei expresii de ``felie``**:

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

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

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

```
Out[49]:
```

	Moneda	Densitatea populatiei
Romania	RON	82.375198
Franta	EUR	104.890280

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

```
In [50]: print(geografie)

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

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

```
Out[50]:
```

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

```
Out[51]:
```

	Populatie	Moneda
Franta	67201000	EUR
Grecia	11183957	EUR

Folosind indicieria, se pot modifica valorile dintr-un DataFrame :

```
In [52]: #Modificarea populatiei Greciei cu iloc
geografie.iloc[1, 1] = 12000000
print(geografie)
```

	Populatie	Suprafata	Moneda	Densitatea populatiei
Romania	19638000	238397	RON	82.375198
Franta	67201000	12000000	EUR	104.890280
Grecia	11183957	131957	EUR	84.754556

```
In [53]: #Modificarea populatiei Greciei cu loc
geografie.loc['Grecia', 'Populatie'] = 11183957
print(geografie)
```

	Populatie	Suprafata	Moneda	Densitatea populatiei
Romania	19638000	238397	RON	82.375198
Franta	67201000	12000000	EUR	104.890280
Grecia	11183957	131957	EUR	84.754556

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

Out[54]:

	Populatie	Suprafata	Moneda	Densitatea populatiei
Franta	67201000	12000000	EUR	104.890280
Grecia	11183957	131957	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 [55]: ser = pd.Series(np.random.randint(low=0, high=10, size=(5)), index=['a', 'b', 'c', 'd', 'e'])
ser
```

```
Out[55]: a      3
         b      9
         c      2
         d      6
         e      4
         dtype: int32
```

```
In [56]: np.exp(ser)
```

```
Out[56]: a      20.085537
         b    8103.083928
         c       7.389056
         d    403.428793
         e     54.598150
         dtype: float64
```

```
In [57]: 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('Original:', my_df)
print('Transformat:', np.exp(my_df))
```

```
Original:   Sunday  Monday  Tuesday  Wednesday
a         8         4         2         8
b         0         1         3         6
c         4         4         0         5
Transformat:   Sunday   Monday   Tuesday   Wednesday
a  2980.957987  54.598150   7.389056  2980.957987
b    1.000000   2.718282  20.085537   403.428793
c   54.598150  54.598150   1.000000   148.413159
```

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

```
In [59]: population / area
```

```
Out[59]: 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 [60]: 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[60]:
```

```
   A  B  C
0  1  9  9
1  2  0  2
```

In [61]:

B

Out[61]:

	B	A
0	1	3
1	6	7
2	2	5

In [62]: A + B

Out[62]:

	A	B	C
0	4.0	10.0	NaN
1	9.0	6.0	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() , subtract()
*	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) (<https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.add.html>).

Exemplu:

In [63]:

A

Out[63]:

	A	B	C
0	1	9	9
1	2	0	2

```
In [64]: B
```

```
Out[64]:
```

	B	A
0	1	3
1	6	7
2	2	5

```
In [65]: A.add(B, fill_value=0)
```

```
Out[65]:
```

	A	B	C
0	4.0	10.0	9.0
1	9.0	6.0	2.0
2	5.0	2.0	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 [66]: 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[66]: 0    1.0
1    2.0
2    3.0
3    NaN
4    5.0
Name: my_series, dtype: float64
```

Funcțiile care se pot folosi pentru un DataFrame pentru a opera cu valori lipsa sunt:

```
In [67]: df = pd.DataFrame([[1, 2, np.NaN], [np.NaN, 10, 20]])
df
```

```
Out[67]:
```

	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 [68]: df.isnull()
```

Out[68]:

	0	1	2
0	False	False	True
1	True	False	False

`notnull()` - opusul functiei precedente

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

```
In [69]: df.dropna()
```

Out[69]:

	0	1	2
--	---	---	---

```
In [70]: 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[70]:

	0	1	2
0	3.0	4	5.0

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

```
In [71]: df = pd.DataFrame([[1, 2, np.NaN], [np.NaN, 10, 20]])
df
```

Out[71]:

	0	1	2
0	1.0	2	NaN
1	NaN	10	20.0

```
In [72]: #umplere de NaNuri cu valoare constanta
df2 = df.fillna(value = 100)
df2
```

```
Out[72]:
```

	0	1	2
0	1.0	2	100.0
1	100.0	10	20.0

```
In [73]: np.random.randn(5, 3)
```

```
Out[73]: array([[ -0.67358399,  0.5341238 ,  0.77049335],
 [ -0.15142897, -0.08186663,  0.01680839],
 [  0.90616317, -2.00942048, -0.48453134],
 [  1.38162399,  0.24060471, -0.21255886],
 [  0.07510248, -0.5744459 , -1.66327426]])
```

```
In [74]: #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[74]:
```

	A	B	C
0	1.441693	1.640187	NaN
1	1.861035	NaN	-1.348852
2	NaN	-1.195495	0.590884
3	-1.270104	-2.045056	-1.155999
4	-0.606587	NaN	-0.109005

```
In [75]: #calcul medie pe coloana
df.mean(axis=0)
```

```
Out[75]: A    0.356509
B   -0.533454
C   -0.505743
dtype: float64
```

```
In [76]: df3 = df.fillna(df.mean(axis=0))
df3
```

```
Out[76]:
```

	A	B	C
0	1.441693	1.640187	-0.505743
1	1.861035	-0.533454	-1.348852
2	0.356509	-1.195495	0.590884
3	-1.270104	-2.045056	-1.155999
4	-0.606587	-0.533454	-0.109005

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) (<https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.fillna.html>):

```
In [77]: my_ds = pd.Series(np.arange(0, 30))  
my_ds[1:-1:4] = np.NaN  
my_ds
```

```
Out[77]: 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 [78]: # copierea ultimei valori non-null
my_ds_filled_1 = my_ds.fillna(method='ffill')
my_ds_filled_1
```

```
Out[78]: 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 [79]: # copierea inapoi a urmatoarei valori non-null
my_ds_filled_2 = my_ds.fillna(method='bfill')
my_ds_filled_2
```

```
Out[79]: 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 [80]: df = pd.DataFrame([[1, np.NaN, 2, np.NaN], [2, 3, 5, np.NaN], [np.NaN, 4, 6, n
p.NaN]])
df
```

```
Out[80]:
```

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

```
In [81]: #Umplere, prin parcurgere pe linii  
df.fillna(method='ffill', axis = 1)
```

```
Out[81]:
```

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

```
Out[82]:
```

	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 [83]: 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[83]: 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 [84]: 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[84]: 1    A
         2    B
         3    C
         3    D
         4    E
         5    F
dtype: object
```

```
In [85]: ser_concat.loc[3]
```

```
Out[85]: 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 [86]: 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 [87]: #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 [88]: 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 [89]: #concatenare simpla
pd.concat([df1, df2])
```

```
Out[89]:
```

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

Concatenarea se poate face si pe orizontala:

```
In [90]: 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 [91]: #concatenare pe axa 1
pd.concat([df3, df4], axis=1)
```

```
Out[91]:
```

	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 [92]: 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 [93]: print(pd.concat([x, y]))
```

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

```
In [94]: 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='int64')

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

```
In [95]: 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 [96]: 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 [97]: 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

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 [98]: 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 [99]: #concatenare cu inner join
pd.concat([df5, df6], join='inner')
```

Out[99]:

	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, prin metoda `reindex` :

```
In [100]: 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 [101]: # pd.concat([df5, df6], join_axes=[df5.columns]) # parametrul join_axes e depr
          ecated
pd.concat([df5, df6.reindex(df5.columns, axis=1)])
```

Out[101]:

	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 [102]: 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 [103]: print(df1)  
print(df2)
```

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

```
In [104]: df3=pd.merge(df1, df2)  
df3
```

Out[104]:

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

```
In [105]: df3 = pd.DataFrame({'employee': ['Jake', 'Lisa', 'Sue'],
'group': ['Engineering', 'Engineering', 'HR']})
df4 = pd.DataFrame({'employee': ['Bob', 'Jake', 'Sue'],
'hire_date': [2008, 2012, 2014]})

print(df3)
print(df4)

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

```
employee      group
0      Jake  Engineering
1      Lisa  Engineering
2      Sue      HR
employee  hire_date
0      Bob      2008
1      Jake      2012
2      Sue      2014
```

Out[105]:

	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 [106]: 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 [107]: pd.merge(df3, df4)
```

Out[107]:

	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 [108]: 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 [109]: 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