

Curs 3: Pandas

Incarcarea datelor

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

```
In [26]: import numpy as np

         tablou = np.array([[1, 2, 3], [3.5, 2, '10']])
         tablou.dtype # Unicode string

Out[26]: 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 [27]: import pandas as pd
         pd.__version__

Out[27]: '0.24.1'
```

Pandas Series

O serie Pandas este un vector unidimensional de date indexate.

```
In [28]: data = pd.Series([0.25, 0.5, 0.75, 1.0])
         data

Out[28]: 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 [29]: data.values

Out[29]: 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 [30]: data.index
Out[30]: RangeIndex(start=0, stop=4, step=1)
```

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

```
In [31]: data = pd.Series([0.25, 0.5, 0.75, 1.0], index=['a', 'b', 'c', 'd'])
In [32]: data.values
Out[32]: array([0.25, 0.5 , 0.75, 1.  ])
In [33]: data.index
Out[33]: Index(['a', 'b', 'c', 'd'], dtype='object')
In [34]: data['b']
Out[34]: 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 [35]: geografie_populatie = {'Romania': 19638000, 'Franta': 67201000, 'Grecia': 1118
    3957}
    populatie = pd.Series(geografie_populatie)
    populatie
Out[35]: Romania      19638000
    Franta        67201000
    Grecia        11183957
    dtype: int64
In [36]: populatie.index
Out[36]: Index(['Romania', 'Franta', 'Grecia'], dtype='object')
In [37]: populatie['Grecia']
Out[37]: 11183957
In [38]: # 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 [39]: 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 [40]: s_tip = pd.Series(['rosu', 1, 1.5])
         s_tip
```

```
Out[40]: 0      rosu
         1         1
         2        1.5
         dtype: object
```

Selectarea datelor in serii

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

```
In [41]: 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[41]: 25.0
```

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

```
In [42]: data['b'] = 300  
print(data)
```

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

Se poate folosi slicing:

```
In [43]: data['a':'c']
```

```
Out[43]: a      0.0  
b    300.0  
c     50.0  
dtype: float64
```

sau se pot folosi liste de selectie:

```
In [44]: data[['a', 'c', 'b', 'c']]
```

```
Out[44]: a      0.0  
c     50.0  
b    300.0  
c     50.0  
dtype: float64
```

sau expresii logice:

```
In [45]: data[(data > 30) & (data < 80)] #se remarca returnarea in rezultat a indicilor  
care satisfac proprietatea ceruta
```

```
Out[45]: 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 [18]: data = pd.Series([1, 2, 3], index=['a', 'b', 'c'])
```

```
data
```

```
Out[18]: a    1  
        b    2  
        c    3  
        dtype: int64
```

```
In [19]: #cautare dupa index cu o singura valoare  
data.loc['b']
```

```
Out[19]: 2
```

```
In [20]: #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[20]: 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 [21]: data.iloc[0]
```

```
Out[21]: 1
```

```
In [22]: data.iloc[[0, 2]]
```

```
Out[22]: 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 [46]: 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
Franta	67201000	640679	EUR
Grecia	11183957	131957	EUR
Romania	19638000	238397	RON

```
In [24]: print(geografie.index)

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

Atributul columns da lista de coloane din obiectul DataFrame:

```
In [25]: geografie.columns

Out[25]: Index(['Moneda', 'Populatie', 'Suprafata'], dtype='object')
```

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

```
In [26]: 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 [27]: mydf = pd.DataFrame([1, 2, 3], columns=['values'])
mydf
```

Out[27]:

	values
0	1
1	2
2	3

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

```
In [28]: data = [{'a': i, 'b': 2 * i} for i in range(3)]
          pd.DataFrame(data)
```

Out[28]:

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

Out[29]:

	a	b	c
0	1.0	2	NaN
1	NaN	3	4.0

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

```
In [30]: pd.DataFrame(np.random.rand(3, 2), columns=['Col1', 'Col2'], index=['a', 'b', 'c'])
```

Out[30]:

	Col1	Col2
a	0.688322	0.028589
b	0.407905	0.061880
c	0.447460	0.487577

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

```
In [31]: geografie['Densitatea populatiei'] = geografie['Populatie'] / geografie['Suprafata']

geografie
```

Out[31]:

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

Un obiect DataFrame poate fi transpus cu atributul T:

```
In [32]: geografie.T
```

Out[32]:

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

Selectarea datelor intr-un DataFrame

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

```
In [33]: print(geografie)
```

```

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

```
In [34]: 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 [35]: geografie.Moneda
```

```
Out[35]: Franta      EUR
         Grecia      EUR
         Romania     RON
         Name: Moneda, dtype: object
```

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

```
In [36]: geografie[geografie.columns[0]]
```

```
Out[36]: Franta      EUR
         Grecia      EUR
         Romania     RON
         Name: Moneda, dtype: object
```

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

```
Out[37]:
```

	0	1	2	3
0	0.955221	0.348672	0.535985	0.531402
1	0.801225	0.068939	0.902218	0.055361
2	0.206871	0.983289	0.922590	0.485234

```
In [38]: my_data[0]
```

```
Out[38]: 0    0.955221
         1    0.801225
         2    0.206871
         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 [39]: #afisare ndarray si tip pentru my_data.values
         print(my_data.values)
         print(my_data.values.dtype)

[[0.95522067 0.34867163 0.53598536 0.53140162]
 [0.80122534 0.06893906 0.90221822 0.05536105]
 [0.20687113 0.98328897 0.92258975 0.48523375]]
float64
```

```
In [40]: #afisare ndarray si tip pentru geografie.values
print(geografie.values)
print(geografie.values.dtype)
```

```
[[ 'EUR' 67201000 640679 104.89028046806591]
 [ 'EUR' 11183957 131957 84.75455640852702]
 [ 'RON' 19638000 238397 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 [41]: print(geografie)

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

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

Out[41]:

	Suprafata	Densitatea populatiei
Franta	640679	104.890280
Grecia	131957	84.754556

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

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

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

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

Out[42]:

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

Se permite folosirea de expresii de filtrare à la NumPy:

```
In [43]: geografie.loc[geografie['Densitatea populatiei'] > 83, ['Populatie', 'Moneda']]
```

Out[43]:

	Populatie	Moneda
Franta	67201000	EUR
Grecia	11183957	EUR

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

```
In [44]: #Modificarea populatiei Greciei cu iloc
geografie.iloc[1, 1] = 12000000
print(geografie)
```

	Moneda	Populatie	Suprafata	Densitatea populatiei
Franta	EUR	67201000	640679	104.890280
Grecia	EUR	12000000	131957	84.754556
Romania	RON	19638000	238397	82.375198

```
In [45]: #Modificarea populatiei Greciei cu loc
geografie.loc['Grecia', 'Populatie'] = 11183957
print(geografie)
```

	Moneda	Populatie	Suprafata	Densitatea populatiei
Franta	EUR	67201000	640679	104.890280
Grecia	EUR	11183957	131957	84.754556
Romania	RON	19638000	238397	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 [46]: geografie[geografie['Densitatea populatiei'] > 83]
```

Out[46]:

	Moneda	Populatie	Suprafata	Densitatea populatiei
Franta	EUR	67201000	640679	104.890280
Grecia	EUR	11183957	131957	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 [47]: ser = pd.Series(np.random.randint(low=0, high=10, size=(5)), index=['a', 'b', 'c', 'd', 'e'])
ser
```

```
Out[47]: a    0
         b    3
         c    0
         d    3
         e    1
         dtype: int32
```

```
In [48]: np.exp(ser)
```

```
Out[48]: a    1.000000
         b   20.085537
         c    1.000000
         d   20.085537
         e    2.718282
         dtype: float64
```

```
In [49]: 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	6	9	2	9
b	6	3	2	0
c	6	8	4	1

Transformat:	Sunday	Monday	Tuesday	Wednesday
a	403.428793	8103.083928	7.389056	8103.083928
b	403.428793	20.085537	7.389056	1.000000
c	403.428793	2980.957987	54.598150	2.718282

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

In [51]: `population / area`

Out[51]:

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 [52]:

```
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[52]:

	A	B	C
0	6	2	0
1	8	1	9

In [53]:

B

Out[53]:

	B	A
0	2	5
1	1	0
2	7	9

In [54]: `A + B`

Out[54]:

	A	B	C
0	11.0	4.0	NaN
1	8.0	2.0	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
+	<code>add()</code>
-	<code>sub()</code> , <code>subtract()</code>
*	<code>mul()</code> , <code>multiply()</code>
/	<code>truediv()</code> , <code>div()</code> , <code>divide()</code>
//	<code>floordiv()</code>
%	<code>mod()</code>
**	<code>pow()</code>

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 [55]: A

Out[55]:

	A	B	C
0	6	2	0
1	8	1	9

In [56]: B

Out[56]:

	B	A
0	2	5
1	1	0
2	7	9

```
In [57]: A.add(B, fill_value=0)
```

```
Out[57]:
```

	A	B	C
0	11.0	4.0	0.0
1	8.0	2.0	9.0
2	9.0	7.0	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 [58]: 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[58]: 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 [59]: df = pd.DataFrame([[1, 2, np.NaN], [np.NaN, 10, 20]])
df
```

```
Out[59]:
```

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

```
Out[60]:
```

	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 [61]: df.dropna()
```

```
Out[61]:
```

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

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

	0	1	2
0	3.0	4	5.0

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

```
In [63]: df = pd.DataFrame([[1, 2, np.NaN], [np.NaN, 10, 20]])
df
```

```
Out[63]:
```

	0	1	2
0	1.0	2	NaN
1	NaN	10	20.0


```
In [64]: #umplere de NaNuri cu valoare constanta
df2 = df.fillna(value = 100)
df2
```

Out[64]:

	0	1	2
0	1.0	2	100.0
1	100.0	10	20.0

```
In [65]: np.random.randn(5, 3)
```

```
Out[65]: array([[ -1.31427511,  0.78149315, -0.7192527 ],
 [  0.99474782,  0.03871079, -0.2750213 ],
 [  0.07150529,  0.6103448 , -1.19984314],
 [-0.80551759, -1.70065537,  0.91261925],
 [-1.32115032,  0.08183471, -0.7459214 ]])
```

```
In [66]: #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[66]:

	A	B	C
0	-1.842704	0.472251	NaN
1	0.107133	NaN	0.283370
2	NaN	1.733644	1.232581
3	-0.244329	-1.596947	-0.200434
4	1.160637	NaN	1.255775

```
In [67]: #calcul medie pe coloana
df.mean(axis=0)
```

```
Out[67]: A    -0.204816
B     0.202983
C     0.642823
dtype: float64
```

```
In [68]: df3 = df.fillna(df.mean(axis=0))
df3
```

Out[68]:

	A	B	C
0	-1.842704	0.472251	0.642823
1	0.107133	0.202983	0.283370
2	-0.204816	1.733644	1.232581
3	-0.244329	-1.596947	-0.200434
4	1.160637	0.202983	1.255775

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

Out[69]:

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

```
Out[70]: 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 [71]: # copierea inapoi a urmatoarei valori non-null
my_ds_filled_2 = my_ds.fillna(method='bfill')
my_ds_filled_2
```

```
Out[71]: 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 [72]: df = pd.DataFrame([[1, np.NaN, 2, np.NaN], [2, 3, 5, np.NaN], [np.NaN, 4, 6, np.NaN]])
df
```

Out[72]:

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

In [73]: *#Umplere, prin parcurgere pe linii*
`df.fillna(method='ffill', axis = 1)`

Out[73]:

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

Out[74]:

	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 [75]: `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[75]:

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 [76]: 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[76]: 1    A
         2    B
         3    C
         3    D
         4    E
         5    F
         dtype: object
```

```
In [77]: ser_concat.loc[3]
```

```
Out[77]: 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 [78]: try:
         ser_concat = pd.concat([ser1, ser2], verify_integrity=True)
       except ValueError as e:
         print('Value error', e)
```

Value error Indexes have overlapping values: [3]

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

```
In [79]: #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 [80]: 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 [81]: #concatenare simpla
pd.concat([df1, df2])
```

Out[81]:

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

Concatenarea se poate face si pe orizontala:

```
In [82]: 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 [83]: #concatenare pe axa 1
pd.concat([df3, df4], axis=1)
#echivalent:
pd.concat([df3, df4], axis=1)
```

Out[83]:

	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 [84]: 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 [85]: print(pd.concat([x, y]))
```

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

```
In [86]: try:
          df_concat = pd.concat([x, y], verify_integrity=True)
        except ValueError as e:
          print('Value error', e)
```

Value error Indexes have overlapping values: [0, 1]

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

```
In [87]: 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 [88]: 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 [89]: 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 [90]: 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 [91]: #concatenare cu inner join
pd.concat([df5, df6], join='inner')
```

```
Out[91]:
```

	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 [92]: 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 [93]: pd.concat([df5, df6], join_axes=[df5.columns])
```

```
Out[93]:
```

	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 [108]: 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 [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 [100]: 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[100]:

	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 [102]: 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	hire_date	supervisor
0	Bob	Accounting	2008	Carly
1	Jake	Engineering	2012	Guido
2	Lisa	Engineering	2004	Guido
3	Sue	HR	2014	Steve

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

```
In [109]: 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 [110]: 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
```