

Curs 3: Pachetul Pandas, reprezentari grafice

3.1. Pandas

Deși NumPy are facilitati pentru incarcarea de date in format CSV, se prefera in practica utilizarea pachetului Pandas:

```
In [1]: import pandas as pd
        pd.__version__

import numpy as np
```

Pandas Series

O serie Pandas este un vector unidimensional de date indexate.

```
In [2]: data = pd.Series([0.25, 0.5, 0.75, 1.0])
        data

Out[2]: 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 [3]: data.values

Out[3]: array([0.25, 0.5 , 0.75, 1.  ])
```

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

```
In [4]: data.index

Out[4]: RangeIndex(start=0, stop=4, step=1)
```

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

```
In [5]: data = pd.Series([0.25, 0.5, 0.75, 1.0], index=['a', 'b', 'c', 'd'])
```

```
In [6]: data.values
```

```
Out[6]: array([0.25, 0.5 , 0.75, 1.  ])
```

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

```
Out[7]: Index(['a', 'b', 'c', 'd'], dtype='object')
```

```
In [8]: data['b']
```

```
Out[8]: 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 [9]: geografie_populatie = {'Romania': 19638000, 'Franta': 67201000, 'Grecia': 11183957}
        populatie = pd.Series(geografie_populatie)
        populatie
```

```
Out[9]: Franta      67201000
        Grecia      11183957
        Romania     19638000
        dtype: int64
```

```
In [10]: populatie.index
```

```
Out[10]: Index(['Franta', 'Grecia', 'Romania'], dtype='object')
```

```
In [11]: populatie['Grecia']
```

```
Out[11]: 11183957
```

```
In [12]: # 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 [13]: 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
```

Selectarea datelor in serii

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

```
In [14]: 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[14]: 25.0

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

```
In [15]: data['b'] = 300
print(data)

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

Se poate folosi slicing:

```
In [16]: data['a':'c']
```

```
Out[16]: a      0.0
b    300.0
c     50.0
dtype: float64
```

sau se pot folosi expresii logice:

```
In [17]: data[(data > 30) & (data < 70)] #se remarca returnarea in rezultat a indicilor  
       care satisfac proprietatea ceruta
```

```
Out[17]: c    50.0  
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 [23]: 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)
```

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

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

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

Atributul columns da lista de coloane:

```
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	0.135958	0.851802
b	0.261209	0.365605
c	0.214928	0.469036

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, ele 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	0.565330	0.440061	0.062999	0.146887
1	0.414304	0.608225	0.836120	0.967140
2	0.406080	0.009127	0.137384	0.251220

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

```
Out[38]: 0    0.565330
         1    0.414304
         2    0.406080
         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)
```

```
[[0.56533024 0.44006117 0.06299924 0.14688675]
 [0.41430384 0.60822515 0.83611981 0.96713977]
 [0.4060805  0.00912671 0.13738385 0.25122011]]
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 indicierea, 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 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      2  
        b      7  
        c      9  
        d      3  
        e      9  
        dtype: int32
```

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

```
Out[48]: a      7.389056  
        b    1096.633158  
        c    8103.083928  
        d     20.085537  
        e    8103.083928  
        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         7       8         0         1
b         6       4         5         5
c         8       6         2         4
Transformat:   Sunday  Monday  Tuesday  Wednesday
a  1096.633158  2980.957987   1.000000   2.718282
b   403.428793   54.598150  148.413159  148.413159
c  2980.957987  403.428793   7.389056   54.598150
```

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	4	3	2
1	2	8	3

In [53]:

B

Out[53]:

	B	A
0	4	3
1	1	5
2	6	8

In [54]: A + B

Out[54]:

	A	B	C
0	7.0	7.0	NaN
1	7.0	9.0	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>).

Exemplu:

In [55]:

A

Out[55]:

	A	B	C
0	4	3	2
1	2	8	3

In [56]: B

Out[56]:

	B	A
0	4	3
1	1	5
2	6	8

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

Out[57]:

	A	B	C
0	7.0	7.0	2.0
1	7.0	9.0	3.0
2	8.0	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 [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 operare 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 [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	-0.184670	1.613389	NaN
1	0.118904	NaN	-0.487000
2	NaN	0.914433	-1.322057
3	-0.761169	-0.214903	-0.183949
4	-0.533982	NaN	0.967723

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

```
Out[67]: A    -0.340229
B     0.770973
C    -0.256321
dtype: float64
```



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

Out[68]:

	A	B	C
0	-0.184670	1.613389	-0.256321
1	0.118904	0.770973	-0.487000
2	-0.340229	0.914433	-1.322057
3	-0.761169	-0.214903	-0.183949
4	-0.533982	0.770973	0.967723

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)
```

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 Series: 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)
          df_concat
```

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. Coloanele pentru care se cauta echivalenta se gasesc automat pe baza numelor lor identice:


```
In [94]: 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]})
print(df1)
print(df2)
```

```
In [95]: df3=pd.merge(df1, df2)
df3
```

Out[95]:

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

```
In [96]: 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 doar 2 linii dupa jonctiune
pd.merge(df3, df4)
```

Out[96]:

	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 [97]: 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 [98]: pd.merge(df3, df4)
```

```
Out[98]:
```

	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 [99]: 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 [100]: 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

```

Implicit, coloanele care participa la jonctiune sunt acelea care au acelasi nume in obiectele DataFrame care se jonctioneaza. Daca numele nu se potrivesc, se pot specifica manual de catre programator prin parametrul on:

```
In [101]: 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 [102]: # restrictionare nume de coloana; doar cea precizata este folosita pentru jonc
tiune
pd.merge(df1, df2, on='employee')
```

```
Out[102]:
```

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

Daca numele sunt diferite, se folosesc parametrii left_on si right_on.

```
In [103]: df3 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
'salary': [70000, 80000, 120000, 90000]})

print(df1)
print(df3)
```

```
   employee      group
0      Bob  Accounting
1     Jake  Engineering
2     Lisa  Engineering
3      Sue           HR

   name  salary
0   Bob   70000
1  Jake   80000
2  Lisa  120000
3   Sue   90000
```

```
In [104]: # jonctiune dupa coloane cu nume diferit

pd.merge(df1, df3, left_on='employee', right_on='name')
```

Out[104]:

	employee	group	name	salary
0	Bob	Accounting	Bob	70000
1	Jake	Engineering	Jake	80000
2	Lisa	Engineering	Lisa	120000
3	Sue	HR	Sue	90000

Constatam placut surprinsi :) ca valorile din employee si name coincid. Putem elimina una din ele folosind metoda `drop()` a obiectului DataFrame rezultat:

```
In [105]: #eliminare de coloana redundanta

pd.merge(df1, df3, left_on='employee', right_on='name').drop('name', axis=1)
```

Out[105]:

	employee	group	salary
0	Bob	Accounting	70000
1	Jake	Engineering	80000
2	Lisa	Engineering	120000
3	Sue	HR	90000

Left, right, outer join

```
In [106]: df6 = pd.DataFrame({'name': ['Peter', 'Paul', 'Mary'], 'food': ['fish', 'beans', 'bread']},
                             columns=['name', 'food']) #specificarea parametrului columns este redundanta
df7 = pd.DataFrame({'name': ['Mary', 'Joseph'], 'drink': ['wine', 'beer']},
                   columns=['name', 'drink']) #idem
```

```
In [107]: print(df6)
          print(df7)
```

```
      name  food
0  Peter  fish
1   Paul  beans
2   Mary  bread
      name drink
0   Mary  wine
1  Joseph  beer
```

Pentru cazul in care se face merge(), implicit se face inner join:

```
In [108]: pd.merge(df6, df7)
```

Out[108]:

	name	food	drink
0	Mary	bread	wine

Parametrul how arata cum altfel se poate face jonctiunea: left, right si outer.

```
In [109]: print(df6)
          print(df7)
```

```
      name  food
0  Peter  fish
1   Paul  beans
2   Mary  bread
      name drink
0   Mary  wine
1  Joseph  beer
```

In [110]: *#outer join: se aduc liniile reunite, unde nu se regasesc valori se completeaza cu NaN*
 pd.merge(df6, df7, how='outer')

Out[110]:

	name	food	drink
0	Peter	fish	NaN
1	Paul	beans	NaN
2	Mary	bread	wine
3	Joseph	NaN	beer

In [111]: *#left join: se aduc toate liniile din partea stanga (primul DataFrame), chiar daca nu au corespondent in partea dreapta. Valorile lipsa se umplu cu NaN*
 print(df6)
 print(df7)
 pd.merge(df6, df7, how='left')

```

      name  food
0  Peter  fish
1   Paul  beans
2   Mary  bread
      name drink
0   Mary  wine
1  Joseph  beer

```

Out[111]:

	name	food	drink
0	Peter	fish	NaN
1	Paul	beans	NaN
2	Mary	bread	wine

Citirea datelor in format CSV

Pandas ofera posibilitatea de a citi fisiere CSV. Metoda `read_csv()` este versatila datorita parametrilor pe care ii permite:

```
In [112]: print(pd.__version__)  
          help(pd.read_csv)
```

0.22.0

Help on function read_csv in module pandas.io.parsers:

```
read_csv(filepath_or_buffer, sep=',', delimiter=None, header='infer', names=None, index_col=None, usecols=None, squeeze=False, prefix=None, mangle_dupe_cols=True, dtype=None, engine=None, converters=None, true_values=None, false_values=None, skipinitialspace=False, skiprows=None, nrows=None, na_values=None, keep_default_na=True, na_filter=True, verbose=False, skip_blank_lines=True, parse_dates=False, infer_datetime_format=False, keep_date_col=False, date_parser=None, dayfirst=False, iterator=False, chunksize=None, compression='infer', thousands=None, decimal=b'.', lineterminator=None, quotechar='"', quoting=0, escapechar=None, comment=None, encoding=None, dialect=None, tupleize_cols=None, error_bad_lines=True, warn_bad_lines=True, skipfooter=0, skip_footer=0, doublequote=True, delim_whitespace=False, as_recarray=None, compact_ints=None, use_unsigned=None, low_memory=True, buffer_lines=None, memory_map=False, float_precision=None)
```

Read CSV (comma-separated) file into DataFrame

Also supports optionally iterating or breaking of the file into chunks.

Additional help can be found in the `online docs for IO Tools`_ <<http://pandas.pydata.org/pandas-docs/stable/io.html>>`_.

Parameters

`filepath_or_buffer` : str, pathlib.Path, py._path.local.LocalPath or any object with a read() method (such as a file handle or StringIO)

The string could be a URL. Valid URL schemes include http, ftp, s3, and

file. For file URLs, a host is expected. For instance, a local file could

be file ://localhost/path/to/table.csv

`sep` : str, default ','

Delimiter to use. If sep is None, the C engine cannot automatically detect

the separator, but the Python parsing engine can, meaning the latter will

be used and automatically detect the separator by Python's builtin sniffer

tool, ``csv.Sniffer``. In addition, separators longer than 1 character and

different from ``'\s+'`` will be interpreted as regular expressions and

will also force the use of the Python parsing engine. Note that regex delimiters are prone to ignoring quoted data. Regex example: ``'\r\n\t'``

`delimiter` : str, default ``None``

Alternative argument name for sep.

`delim_whitespace` : boolean, default False

Specifies whether or not whitespace (e.g. ``'\s+'`` or ``'\s+|\\s+'``) will be

used as the sep. Equivalent to setting ``sep='\s+'``. If this option is set to True, nothing should be passed in for the ``delimiter`` parameter.

.. versionadded:: 0.18.1 support for the Python parser.

header : int or list of ints, default 'infer'
 Row number(s) to use as the column names, and the start of the data. Default behavior is to infer the column names: if no names are passed the behavior is identical to ``header=0`` and column names are inferred from the first line of the file, if column names are passed explicitly then the behavior is identical to ``header=None``. Explicitly pass ``header=0`` to be able to replace existing names. The header can be a list of integers that specify row locations for a multi-index on the columns e.g. [0,1,3]. Intervening rows that are not specified will be skipped (e.g. 2 in this example is skipped). Note that this parameter ignores commented lines and empty lines if ``skip_blank_lines=True``, so header=0 denotes the first line of data rather than the first line of the file.

names : array-like, default None
 List of column names to use. If file contains no header row, then you should explicitly pass header=None. Duplicates in this list will cause a ``UserWarning`` to be issued.

index_col : int or sequence or False, default None
 Column to use as the row labels of the DataFrame. If a sequence is given, a MultiIndex is used. If you have a malformed file with delimiters at the end of each line, you might consider index_col=False to force pandas to not use the first column as the index (row names)

usecols : array-like or callable, default None
 Return a subset of the columns. If array-like, all elements must either be positional (i.e. integer indices into the document columns) or strings that correspond to column names provided either by the user in `names` or inferred from the document header row(s). For example, a valid array-like `usecols` parameter would be [0, 1, 2] or ['foo', 'bar', 'baz'].

If callable, the callable function will be evaluated against the column names, returning names where the callable function evaluates to True. An example of a valid callable argument would be ``lambda x: x.upper() in ['AAA', 'BBB', 'DDD']``. Using this parameter results in much faster parsing time and lower memory usage.

as_recarray : boolean, default False
 .. deprecated:: 0.19.0
 Please call ``pd.read_csv(...).to_records()`` instead.

Return a NumPy recarray instead of a DataFrame after parsing the data.

If set to True, this option takes precedence over the `squeeze` parameter.

In addition, as row indices are not available in such a format, the `index_col` parameter will be ignored.

```

squeeze : boolean, default False
    If the parsed data only contains one column then return a Series
prefix : str, default None
    Prefix to add to column numbers when no header, e.g. 'X' for X0, X1,
...
mangle_dupe_cols : boolean, default True
    Duplicate columns will be specified as 'X.0'...'X.N', rather than
    'X'...'X'. Passing in False will cause data to be overwritten if ther
e
    are duplicate names in the columns.
dtype : Type name or dict of column -> type, default None
    Data type for data or columns. E.g. {'a': np.float64, 'b': np.int32}
    Use `str` or `object` to preserve and not interpret dtype.
    If converters are specified, they will be applied INSTEAD
    of dtype conversion.
engine : {'c', 'python'}, optional
    Parser engine to use. The C engine is faster while the python engine
is
    currently more feature-complete.
converters : dict, default None
    Dict of functions for converting values in certain columns. Keys can
either
    be integers or column labels
true_values : list, default None
    Values to consider as True
false_values : list, default None
    Values to consider as False
skipinitialspace : boolean, default False
    Skip spaces after delimiter.
skiprows : list-like or integer or callable, default None
    Line numbers to skip (0-indexed) or number of lines to skip (int)
    at the start of the file.

    If callable, the callable function will be evaluated against the row
    indices, returning True if the row should be skipped and False otherw
ise.
    An example of a valid callable argument would be ``lambda x: x in [0,
2]``.
skipfooter : int, default 0
    Number of lines at bottom of file to skip (Unsupported with engine
='c')
skip_footer : int, default 0
    .. deprecated:: 0.19.0
    Use the `skipfooter` parameter instead, as they are identical
nrows : int, default None
    Number of rows of file to read. Useful for reading pieces of large fi
les
na_values : scalar, str, list-like, or dict, default None
    Additional strings to recognize as NA/NaN. If dict passed, specific
    per-column NA values. By default the following values are interprete
d as
    NaN: '', '#N/A', '#N/A N/A', '#NA', '-1.#IND', '-1.#QNAN', '-NaN', '-
nan',
    '1.#IND', '1.#QNAN', 'N/A', 'NA', 'NULL', 'NaN', 'n/a', 'nan',
    'null'.
keep_default_na : bool, default True
    If na_values are specified and keep_default_na is False the default N

```

aN values are overridden, otherwise they're appended to.
 na_filter : boolean, default True
 Detect missing value markers (empty strings and the value of na_value
 s). In data without any NAs, passing na_filter=False can improve the perform
 ance of reading a large file
 verbose : boolean, default False
 Indicate number of NA values placed in non-numeric columns
 skip_blank_lines : boolean, default True
 If True, skip over blank lines rather than interpreting as NaN values
 parse_dates : boolean or list of ints or names or list of lists or dict,
 default False

- * boolean. If True -> try parsing the index.
- * list of ints or names. e.g. If [1, 2, 3] -> try parsing columns 1,
 2, 3 each as a separate date column.
- * list of lists. e.g. If [[1, 3]] -> combine columns 1 and 3 and par
 se as a single date column.
- * dict, e.g. {'foo' : [1, 3]} -> parse columns 1, 3 as date and call
 result 'foo'

If a column or index contains an unparseable date, the entire column
 or index will be returned unaltered as an object data type. For non-stand
 ard datetime parsing, use ``pd.to_datetime`` after ``pd.read_csv``

Note: A fast-path exists for iso8601-formatted dates.
 infer_datetime_format : boolean, default False
 If True and ``parse_dates`` is enabled, pandas will attempt to infer th
 e format of the datetime strings in the columns, and if it can be infer
 red, switch to a faster method of parsing them. In some cases this can inc
 rease the parsing speed by 5-10x.

keep_date_col : boolean, default False
 If True and ``parse_dates`` specifies combining multiple columns then
 keep the original columns.

date_parser : function, default None
 Function to use for converting a sequence of string columns to an arr
 ay of datetime instances. The default uses ``dateutil.parser.parser`` to do
 the conversion. Pandas will try to call ``date_parser`` in three different
 ways, advancing to the next if an exception occurs: 1) Pass one or more arr
 ays (as defined by ``parse_dates``) as arguments; 2) concatenate (row-wise)
 the string values from the columns defined by ``parse_dates`` into a single
 array

and pass that; and 3) call ``date_parser`` once for each row using one
or more strings (corresponding to the columns defined by ``parse_dates``)
as arguments.

`dayfirst` : boolean, default False
DD/MM format dates, international and European format

`iterator` : boolean, default False
Return `TextFileReader` object for iteration or getting chunks with
``get_chunk()``.

`chunksize` : int, default None
Return `TextFileReader` object for iteration.
See the ``IO Tools docs``
<http://pandas.pydata.org/pandas-docs/stable/io.html#io-chunking> for
more information on ``iterator`` and ``chunksize``.

`compression` : {'infer', 'gzip', 'bz2', 'zip', 'xz', None}, default 'infer'

For on-the-fly decompression of on-disk data. If 'infer' and
``filepath_or_buffer`` is path-like, then detect compression from the
following extensions: '.gz', '.bz2', '.zip', or '.xz' (otherwise no
decompression). If using 'zip', the ZIP file must contain only one da
ta file to be read in. Set to None for no decompression.

.. versionadded:: 0.18.1 support for 'zip' and 'xz' compression.

`thousands` : str, default None
Thousands separator

`decimal` : str, default '.'
Character to recognize as decimal point (e.g. use ',' for European da
ta).

`float_precision` : string, default None
Specifies which converter the C engine should use for floating-point
values. The options are ``None`` for the ordinary converter,
``high`` for the high-precision converter, and ``round_trip`` for the
round-trip converter.

`lineterminator` : str (length 1), default None
Character to break file into lines. Only valid with C parser.

`quotechar` : str (length 1), optional
The character used to denote the start and end of a quoted item. Quot
ed items can include the delimiter and it will be ignored.

`quoting` : int or csv.QUOTE_* instance, default 0
Control field quoting behavior per ``csv.QUOTE_*`` constants. Use one
of QUOTE_MINIMAL (0), QUOTE_ALL (1), QUOTE_NONNUMERIC (2) or QUOTE_NONE
(3).

`doublequote` : boolean, default ``True``
When quotechar is specified and quoting is not ``QUOTE_NONE``, indicat
e whether or not to interpret two consecutive quotechar elements INSIDE
a field as a single ``quotechar`` element.

`escapechar` : str (length 1), default None
One-character string used to escape delimiter when quoting is QUOTE_N
ONE.

`comment` : str, default None

Indicates remainder of line should not be parsed. If found at the beginning of a line, the line will be ignored altogether. This parameter must be a single character. Like empty lines (as long as `skip_blank_lines=True`), fully commented lines are ignored by the parameter `header` but not by `skiprows`. For example, if `comment='#'`, parsing `'#empty\na,b,c\n1,2,3'` with `header=0` will result in `'a,b,c'` being treated as the header.

`encoding` : str, default None
Encoding to use for UTF when reading/writing (ex. `'utf-8'`). List of Python standard encodings
<<https://docs.python.org/3/library/codecs.html#standard-encodings>>_

`dialect` : str or csv.Dialect instance, default None
If provided, this parameter will override values (default or not) for the following parameters: `delimiter`, `doublequote`, `escapechar`, `skipinitialspace`, `quotechar`, and `quoting`. If it is necessary to override values, a ParserWarning will be issued. See csv.Dialect documentation for more details.

`tupleize_cols` : boolean, default False
.. deprecated:: 0.21.0
This argument will be removed and will always convert to MultiIndex

x

Leave a list of tuples on columns as is (default is to convert to a MultiIndex on the columns)

`error_bad_lines` : boolean, default True
Lines with too many fields (e.g. a csv line with too many commas) will by default cause an exception to be raised, and no DataFrame will be returned. If False, then these "bad lines" will be dropped from the DataFrame that is returned.

`warn_bad_lines` : boolean, default True
If `error_bad_lines` is False, and `warn_bad_lines` is True, a warning for each "bad line" will be output.

`low_memory` : boolean, default True
Internally process the file in chunks, resulting in lower memory use while parsing, but possibly mixed type inference. To ensure no mixed types either set False, or specify the type with the `dtype` parameter.

r.
Note that the entire file is read into a single DataFrame regardless, use the `chunksize` or `iterator` parameter to return the data in chunks.

(Only valid with C parser)

`buffer_lines` : int, default None
.. deprecated:: 0.19.0
This argument is not respected by the parser

`compact_ints` : boolean, default False
.. deprecated:: 0.19.0

Argument moved to ``pd.to_numeric``

If `compact_ints` is `True`, then for any column that is of integer dtype, the parser will attempt to cast it as the smallest integer dtype possible, either signed or unsigned depending on the specification from the `use_unsigned` parameter.

`use_unsigned` : boolean, default `False`

.. deprecated:: 0.19.0

Argument moved to ``pd.to_numeric``

If integer columns are being compacted (i.e. `compact_ints=True`), specify whether the column should be compacted to the smallest signed or unsigned integer dtype.

`memory_map` : boolean, default `False`

If a filepath is provided for `filepath_or_buffer`, map the file object directly onto memory and access the data directly from there. Using this option can improve performance because there is no longer any I/O overhead.

Returns

result : DataFrame or TextParser

Exemplu: date din SUA

Nota: exemplul este preluat din referinta bibliografica [1] din cursul 1.

Datele folosite sunt de la adresele:

- <https://raw.githubusercontent.com/jakevdp/data-USstates/master/state-population.csv>
(<https://raw.githubusercontent.com/jakevdp/data-USstates/master/state-population.csv>)
- <https://raw.githubusercontent.com/jakevdp/data-USstates/master/state-areas.csv>
(<https://raw.githubusercontent.com/jakevdp/data-USstates/master/state-areas.csv>)
- <https://raw.githubusercontent.com/jakevdp/data-USstates/master/state-abbrevs.csv>
(<https://raw.githubusercontent.com/jakevdp/data-USstates/master/state-abbrevs.csv>)

```
In [113]: pop = pd.read_csv('./data/state-population.csv')
          areas = pd.read_csv('./data/state-areas.csv')
          abbrevs = pd.read_csv('./data/state-abbrevs.csv')
```

Vizualizarea primelor randuri din fiecare:

```
In [114]: pop.head()
```

```
Out[114]:
```

	state/region	ages	year	population
0	AL	under18	2012	1117489.0
1	AL	total	2012	4817528.0
2	AL	under18	2010	1130966.0
3	AL	total	2010	4785570.0
4	AL	under18	2011	1125763.0

```
In [115]: areas.head()
```

```
Out[115]:
```

	state	area (sq. mi)
0	Alabama	52423
1	Alaska	656425
2	Arizona	114006
3	Arkansas	53182
4	California	163707

```
In [116]: abbrevs.head()
```

```
Out[116]:
```

	state	abbreviation
0	Alabama	AL
1	Alaska	AK
2	Arizona	AZ
3	Arkansas	AR
4	California	CA

Se cere ordinarea statelor si teritoriilor dupa densitatea de populatie din 2010. Primul pas este jonctionarea datelor de populatie si de abrevieri, pentru ca in tabela de suprafete se foloseste numele intreg al statului.

```
In [117]: merged = pd.merge(pop, abbrevs, how='outer', left_on='state/region', right_on='abbreviation')
merged.head()
```

Out[117]:

	state/region	ages	year	population	state	abbreviation
0	AL	under18	2012	1117489.0	Alabama	AL
1	AL	total	2012	4817528.0	Alabama	AL
2	AL	under18	2010	1130966.0	Alabama	AL
3	AL	total	2010	4785570.0	Alabama	AL
4	AL	under18	2011	1125763.0	Alabama	AL

Coloana de abrevieri se poate omite din acest moment:

```
In [118]: merged = merged.drop('abbreviation', axis=1)
merged.head()
```

Out[118]:

	state/region	ages	year	population	state
0	AL	under18	2012	1117489.0	Alabama
1	AL	total	2012	4817528.0	Alabama
2	AL	under18	2010	1130966.0	Alabama
3	AL	total	2010	4785570.0	Alabama
4	AL	under18	2011	1125763.0	Alabama

Datele de regula sunt incomplete (cu goluri); de exemplu, se poate ca pentru coloana population sa lipseasca valori:

```
In [119]: merged.isnull().any()
```

```
Out[119]: state/region    False
ages                  False
year                  False
population            True
state                 True
dtype: bool
```

Afisarea primelor cazuri in care valorile lipsesc pentru coloana population se face cu:


```
In [120]: merged[merged['population'].isnull()].head() #PR=Puerto Rico
```

```
Out[120]:
```

	state/region	ages	year	population	state
2448	PR	under18	1990	NaN	NaN
2449	PR	total	1990	NaN	NaN
2450	PR	total	1991	NaN	NaN
2451	PR	under18	1991	NaN	NaN
2452	PR	total	1993	NaN	NaN

De asemenea, observam ca exista state pentru care valoarea e nula. Acestea sunt:

```
In [121]: merged.loc[merged['state'].isnull(), 'state/region'].unique()
```

```
Out[121]: array(['PR', 'USA'], dtype=object)
```

Se umplu valorile de 'state' cu 'Puerto Rico', respectiv 'United States of America' pentru acele cazuri cu 'state/region' 'PR' si respectiv 'USA'

```
In [122]: merged.loc[merged['state/region'] == 'PR', 'state'] = 'Puerto Rico'
merged.loc[merged['state/region'] == 'USA', 'state'] = 'United States of America'
merged.isnull().any()
```

```
Out[122]: state/region    False
ages                  False
year                  False
population             True
state                  False
dtype: bool
```

Putem face jonctiune cu colectia de suprafete (arii):

```
In [123]: final = pd.merge(merged, areas, on='state', how='left')
final.head()
```

```
Out[123]:
```

	state/region	ages	year	population	state	area (sq. mi)
0	AL	under18	2012	1117489.0	Alabama	52423.0
1	AL	total	2012	4817528.0	Alabama	52423.0
2	AL	under18	2010	1130966.0	Alabama	52423.0
3	AL	total	2010	4785570.0	Alabama	52423.0
4	AL	under18	2011	1125763.0	Alabama	52423.0

Verificare daca exista valori de null:

```
In [124]: final.isnull().any()
```

```
Out[124]: state/region    False
ages                    False
year                    False
population              True
state                   False
area (sq. mi)          True
dtype: bool
```

Eliminam liniile pe care se afla valori de null:

```
In [125]: final.dropna(inplace=True)
final.head()
```

```
Out[125]:
```

	state/region	ages	year	population	state	area (sq. mi)
0	AL	under18	2012	1117489.0	Alabama	52423.0
1	AL	total	2012	4817528.0	Alabama	52423.0
2	AL	under18	2010	1130966.0	Alabama	52423.0
3	AL	total	2010	4785570.0	Alabama	52423.0
4	AL	under18	2011	1125763.0	Alabama	52423.0

Selectam acele cazuri pentru care anul de recensamant este 2010 si se considera toate grupele de varsta = toti locuitorii:

```
In [126]: data2010 = final.query("year == 2010 & ages == 'total'")
data2010.head()
```

```
Out[126]:
```

	state/region	ages	year	population	state	area (sq. mi)
3	AL	total	2010	4785570.0	Alabama	52423.0
91	AK	total	2010	713868.0	Alaska	656425.0
101	AZ	total	2010	6408790.0	Arizona	114006.0
189	AR	total	2010	2922280.0	Arkansas	53182.0
197	CA	total	2010	37333601.0	California	163707.0

Putem face calculul densitatii intr-un obiect Series separat. Inainte de asta, e indicat sa se seteze un index pe data2010:

```
In [127]: data2010.set_index('state', inplace=True)
density = data2010['population'] / data2010['area (sq. mi)']
```

```
In [128]: density.head()
```

```
Out[128]: state
Alabama      91.287603
Alaska       1.087509
Arizona      56.214497
Arkansas     54.948667
California   228.051342
dtype: float64
```

Afisarea celor mai populate regiuni se face cu:

```
In [129]: density.sort_values(ascending=False, inplace=True)
density.head()
```

```
Out[129]: state
District of Columbia    8898.897059
Puerto Rico            1058.665149
New Jersey              1009.253268
Rhode Island            681.339159
Connecticut             645.600649
dtype: float64
```

...iar cele mai putin populate sunt:

```
In [130]: density.tail()
```

```
Out[130]: state
South Dakota    10.583512
North Dakota    9.537565
Montana         6.736171
Wyoming         5.768079
Alaska          1.087509
dtype: float64
```

%TODO: agregare si grupare, [1] pagina 158 si urmatoarele.; operatii cu serii detimp, pag 188+; high performance Pandas, pag 209+

Reprezentari grafice cu Matplotlib

Reprezentari grafice se pot obtine cu pachetele Seaborn si Matplotlib. Traditional, se folosesc urmatoarele importuri pentru pachetul Matplotlib:

```
In [131]: %matplotlib inline

import matplotlib as mpl
import matplotlib.pyplot as plt
```

Se recomanda de asemenea specificarea liniei:

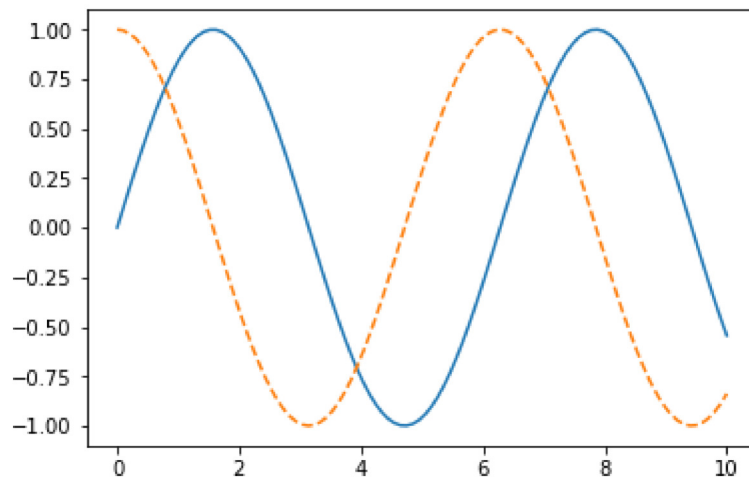
```
%matplotlib notebook
```

pentru desene interactive, sau

```
%matplotlib inline
```

pentru includerea de imagini statice in Jupyter notebook. Linia aleasa se prefera a se scrie inainte de importul de matplotlib.

```
In [132]: x = np.linspace(0, 10, 100)
fig = plt.figure()
plt.plot(x, np.sin(x), '-')
plt.plot(x, np.cos(x), '--'); #se remarca ';' din final
```



Figurile se pot salva, la dorinta, pe disc:

```
In [133]: fig.savefig('trigonometrie.png')
fig.savefig('trigonometrie.jpg')
fig.savefig('trigonometrie.eps')
fig.savefig('trigonometrie.pdf')
```

Tipurile de imagini in care se poate face salvarea sunt:

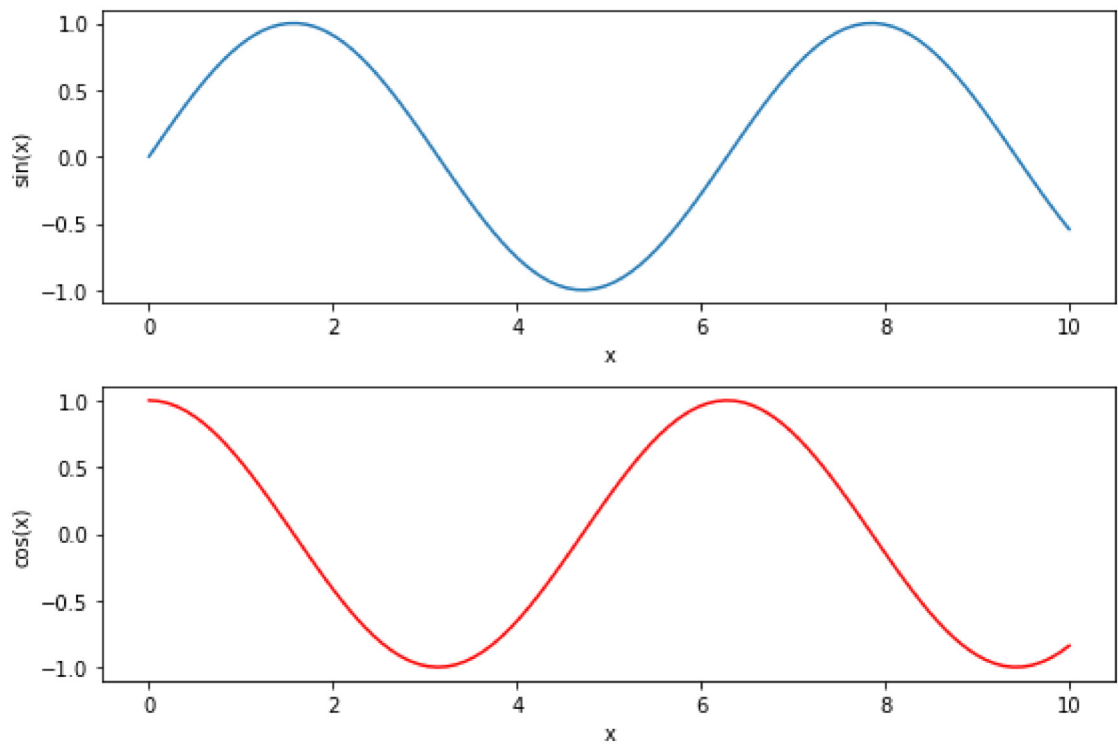
```
In [134]: fig.canvas.get_supported_filetypes()
```

```
Out[134]: {'eps': 'Encapsulated Postscript',  
          'jpeg': 'Joint Photographic Experts Group',  
          'jpg': 'Joint Photographic Experts Group',  
          'pdf': 'Portable Document Format',  
          'pgf': 'PGF code for LaTeX',  
          'png': 'Portable Network Graphics',  
          'ps': 'Postscript',  
          'raw': 'Raw RGBA bitmap',  
          'rgba': 'Raw RGBA bitmap',  
          'svg': 'Scalable Vector Graphics',  
          'svgz': 'Scalable Vector Graphics',  
          'tif': 'Tagged Image File Format',  
          'tiff': 'Tagged Image File Format'}
```

Unul din stilurile de scriere foloseste interfata plt pentru desenare, inspirat din Matlab:

```
In [135]: plt.figure(figsize = (8, 5)) # create a plot figure
# se creeaza primul grafic
plt.subplot(2, 1, 1) # (numar de 'linii' cu grafice, numar de 'coloane' cu gra
fice, numarul graficului care se deseneaza)
#etichete pe abscisa si ordonata
plt.xlabel('x')
plt.ylabel('sin(x)')
plt.plot(x, np.sin(x))
plt.tight_layout() #pentru a se vedea legenda abscisei

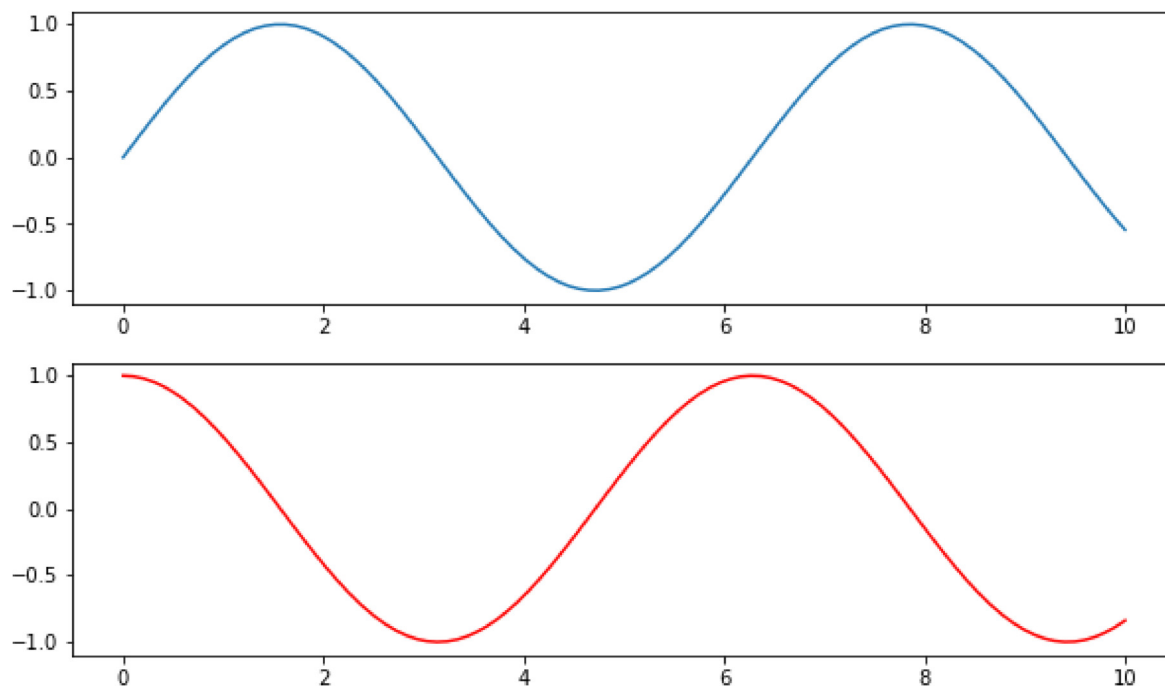
# se creeaza al doilea grafic
plt.subplot(2, 1, 2)
plt.xlabel('x')
plt.ylabel('cos(x)')
plt.plot(x, np.cos(x), 'r')
plt.show()
```



Al doilea stil este orientat pe obiecte:

```
In [136]: # se creeaza un grid pentru desenare
# ax va fi un tablou cu cele doua obiecte de tip Axe
fig, ax = plt.subplots(nrows=2, ncols=1, figsize=(10, 6))

# se apeleaza metoda plot() pe obiectul corespunzator
ax[0].plot(x, np.sin(x))
ax[1].plot(x, np.cos(x), 'r');
```



De regula, axele se auto-scaleaza pentru a cuprinde graficul desenat. Daca se vrea specificarea manuala a dimensiunii graficului, atunci:

```
In [137]: plt.style.use('seaborn-whitegrid')
plt.plot(x, np.exp(x))
plt.xlim(-2, 5)
plt.ylim(-1, 30)
plt.title('Reprezentarea functiei  $e^x$ ');
```

