.

Ampliació d'Informàtica
Biblioteca pandas

Maria Dolors Ayala Vallespí
Dept. Ciències de la computació
E.T.S.E.I.B.
Universitat Politècnica de Catalunya

Tardor 2023

Contents

1 Biblioteca pandas

Nota prèvia: Aquest capítol és una introducció a les possibilitats de la biblioteca pandas. Es imprescindible consultar sempre la documentació de la biblioteca pandas abans d'usar un mètode: allà hi trobarem l'especificació completa i actualitzada, així com altres mètodes que no s'introdueixen en aquest capítol.

pandas és una biblioteca per a la representació i anàlisi de dades.

Per usar pandas cal descarregar la biblioteca de https://pandas.pydata.org/ i importar-la:

```
import pandas as pd
```

pandas ofereix els següents tipus de dades (classes):

- `Series`: taula 1D indexada de longitud fixa
- `DataFrame`: taula 2D indexada de longitud variable

Les classes de pandas estan implementades sobre les classes del mòdul `numpy`, en particular la classe `ndarray` que és una taula multi-dimensional.

En aquest text ens centrarem en les classes `Series` i `DataFrame`. Aquestes classes estan relacionades ja que comparteixen atributs i mètodes i a més les files i columnes d'un `DataFrame` són objectes de tipus `Series` i alguns mètodes del tipus `DataFrame` retornen `Series`. Un `DataFrame` és un contenidor de `Series` i una `Series` és un contenidor de valors escalars. Aquests tipus permeten consultar, inserir i esborrar elements amb una sintaxi similar a la dels diccionaris.

Els objectes `pandas` són mutables: es podem modificar els seus valors. De totes maneres, molts mètodes són no modificadors i d'altres permeten usar un paràmetre opcional booleà (`inplace`) per indicar si el mètode ha de ser modificador (`True`) o no modificador (`False`, valor per defecte). Per altra banda la classe `Series` té una longitud fixa i, per tant, aquesta no es pot modificar.

A les bases de dades reals sovint hi ha informació que falta, dades que s'han perdut. `pandas` ofereix un marcador estàndard per aquest tipus d'informació, `NaN`, que vol dir textualment *not a number* i que significa que és informació que falta.

1.1 La classe `Series`

La classe `Series` representa taules 1D etiquetades que poden contenir qualsevol tipus de dades: enters, reals, strings i qualsevol altre tipus d'objecte:

Vegem la transcripció del mètode de creació de la classe `Series`, tal com apareix a la documentació.

```
class pandas.Series(data=None, index=None, dtype=None, name=None,
                    copy=False, fastpath=False)

data: array-like, Iterable, dict, or scalar value
      Contains data stored in Series.
```

```
index: array-like or Index (1D)

dtype: str, numpy.dtype, or ExtensionDtype, optional

name: str, optional
      The name to give to the Series.

copy: bool, default False
      Copy input data.
```

A l'exemple següent es crea una `Series` a partir d'una llista:

```
>>> s1 = pd.Series([28, 34, 59, 91, 3])
>>> s1
0    28
1    34
2    59
3    91
4     3
dtype: int64
```

L'atribut `index` és l'índex de la `Series` i l'atribut `values` fa referència als valors. L'atribut `size` indica el nombre d'elements de la `Series`:

```
>>> s1.axes
[RangeIndex(start=0, stop=5, step=1)]
>>> s1.index
RangeIndex(start=0, stop=5, step=1)
>>> s1.values
array([28, 34, 59, 91,  3])
>>> s1.size    # len(s1)
5
```

L'índex es pot redefinir:

```
>>> s1.index = ['Joe', 'Liu', 'Pol', 'Jan', 'Iu']
>>> s1
Joe    28
Liu    34
Pol    59
Jan    91
Iu      3
dtype: int64
```

Els objectes de la classe `Series` són taules 1D i es poden convertir a altres estructures unidimensionals com llistes, tuples o diccionaris. També es poden convertir a conjunts:

```
>>> list(s1)
[28, 34, 59, 91, 3]
>>> tuple(s1)
(28, 34, 59, 91, 3)
>>> dict(s1)
{'Joe': 28, 'Liu': 34, 'Pol': 59, 'Jan': 91, 'Iu': 3}
>>> set(s1)
{34, 3, 59, 91, 28}
```

Podem indexar i fer llesques a les `Series` usant l'índex o un enter indicant la posició amb els atributs `loc[]` i `iloc[]`:

```
>>> s1['Joe']
28
>>> s1.loc['Joe']
28
>>> s1.iloc[3]      # indexat per posició
91
>>> s1['Liu':'Iu'] # llesques
Liu    34
Pol    59
Jan    91
Iu      3
dtype: int64
```

La sintaxi per les operacions de modificar, inserir o esborrar un element és la següent:

```
>>> s1['Pol'] = 19   # modificació
>>> s1['Jim'] = 25   # inserció
>>> del s1['Liu']    # esborrat
>>> s1
Joe    28
Pol    19
Jan    91
Iu      3
Jim    25
dtype: int64
```

Podem crear una altra `Series` seleccionant els elements a partir d'una condició (expressió booleana). Si l'expressió booleana és composta, cal posar cada part entre parèntesis. Les operacions booleanes es representen amb els següents símbols: & (`and`), | (`or`) i $\sim$ (`not`)

```
>>> s1[s1<20]
Pol    19
Iu      3
dtype: int64

>>> s1[(s1>18) & (s1<50)]
Joe    28
Pol    19
Jim    25
dtype: int64
```

Els operadors bàsics (+, -, *, /, >, >=, <, <=, ==, !=) estan definits sobre la classe `Series`. Per operar dues `Series`, cal que tinguin el mateix índex.

```
>>> s1+10
Joe     38
Pol     29
Jan    101
Iu      13
Jim     35
dtype: int64

>>> s1+s1
Joe     56
Pol     38
Jan    182
Iu       6
```

```
Jim     50
dtype: int64

>>> s1>=18
Joe      True
Pol      True
Jan      True
Iu      False
Jim      True
dtype: bool
```

La classe `Series` disposa de molts mètodes que com veurem més endavant comparteix amb la classe `DataFrame`:

- `head` i `tail` mostren els primers n elements.
- `sum`, `mean`, `std` calculen respectivament la suma, la mitjana i la desviació estàndard
- `max` i `min` calculen el valor màxim i mínim
- `idxmax`, `idxmin` calculen l'index del màxim i del mínim

Exemples:

```
>>> s1.head(2)
Joe    28
Pol    19
dtype: int64

>>> s1.tail(2)
Iu      3
Jim    25
dtype: int64

>>> s1.sum()
166
>>> s1.max()
91
>>> s1.min()
3
>>> s1.idxmax()
'Jan'
>>> s1.mean()
33.2
>>> s1.std()
33.72239611889997
```

A més d'aquests mètodes, també suporta les funcions sum, max, min, map, iter i filter:

```
>>> sum(s1)
166
>>> max(s1)
91
>>> list(filter(lambda x: x<=25, s1))
[19, 3, 25]
```

Les `Series` són iterables:

```
>>> for e in s1:
...     print(e, end='-')
28-19-91-3-25-
```

En el següent exemple s'ha creat una `Series` a partir d'un diccionari i redefinint l'índex. Observem que la `Series` és incompleta: hi manquen dades. pandas representa aquesta manca d'informació amb el valor `NaN` (not a number). Entre altres funcionalitats que es veuran a l'apartat 1.4, pandas permet tractar aquesta informació: els mètodes `sum`, `min` i `max` tracten correctament `Series` amb manca de dades, però en canvi, les funcions de la llibreria estàndard de Python `sum`, `min` i `max`, no.

```
>>> s1 = pd.Series({'a': 0.1, 'b': 1.0, 'c': 2.4}, index=['b', 'c', 'd', 'a'
    ])
>>> s1
b    1.0
c    2.4
d    NaN
a    0.1
dtype: float64
>>> s1.sum()
3.5
>>> sum(s1)
nan
```

1.2 La classe `DataFrame`

La classe `DataFrame` representa taules 2D etiquetades que poden contenir qualsevol tipus de dades: enters, reals, strings i qualsevol altre tipus d'objecte.

Vegem la transcripció del mètode de creació de la classe `DataFrame`, tal com apareix a la documentació.

```
class pandas.DataFrame(data=None, index=None, columns=None,
                       dtype=None, copy=False)

data : numpy ndarray (structured or homogeneous), dict, or DataFrame.
       Dict can contain Series, arrays, constants, or list-like objects

index : Index or array-like.
        Index to use for resulting frame. Will default to np.arange(n)
        if no indexing information part of input data and no index provided

columns : Index or array-like.
          Column labels to use for resulting frame.
         Will default to np.arange(n) if no column labels are provided

dtype : dtype, default None.
        Data type to force. Only a single dtype is allowed. If None, infer

copy : boolean, default False.
       Copy data from inputs. Only affects DataFrame / 2d ndarray input
```

Creem un `DataFrame` creant les `Series` corresponents a les seves columnes:

```
>>> df = pd.DataFrame()
>>> df['Nom'] = ['Joe', 'Ana', 'Pol', 'Mar', 'Iu']
>>> df['Edat'] = [18, 19, 21, 18, 19]
>>> df['Nota1'] = [4.1, 8.3, 5.9, 9.2, 7.4]
>>> df['Nota2'] = [5.1, 3.3, 6.9, 9.8, 8.4]
>>> df
   Nom  Edat  Nota1  Nota2
0  Joe    18    4.1    5.1
1  Ana    19    8.3    3.3
2  Pol    21    5.9    6.9
3  Mar    18    9.2    9.8
4   Iu    19    7.4    8.4
```

A continuació es mostren els seus atributs bàsics i el resultat d'aplicar el mètode `info`:

```
### atributs
>>> df.axes
[RangeIndex(start=0, stop=5, step=1), Index(['Nom', 'Edat', 'Nota1', 'Nota2'
    ], dtype='object')]
>>> df.index
RangeIndex(start=0, stop=5, step=1)
>>> df.columns
Index(['Nom', 'Edat', 'Nota1', 'Nota2'], dtype='object')
>>> len(df)      # len(df.index)
5
>>> len(df.columns)
4
>>> df.shape
(5, 4)
>>> df.size     # nombre d'elements 5*4
20
>>> df.values
array([['Joe', 18, 4.1, 5.1],
       ['Ana', 19, 8.3, 3.3],
       ['Pol', 21, 5.9, 6.9],
       ['Mar', 18, 9.2, 9.8],
       ['Iu', 19, 7.4, 8.4]], dtype=object)

>>> df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 4 columns):
Nom      5 non-null object
Edat     5 non-null int64
Nota1    5 non-null float64
Nota2    5 non-null float64
dtypes: float64(2), int64(1), object(1)
memory usage: 240.0+ bytes
```

El mètode `set_index` permet definir una columna com a índex. Aquest mètode, entre altres, té el paràmetre opcional `inplace` que indica si el mètode és modificador (`True`) o no modificador (`False`, valor per defecte).

```
>>> df.set_index('Nom', inplace = True)
>>> df
     Edat  Nota1  Nota2
Nom
Joe    18    4.1    5.1
Ana    19    8.3    3.3
Pol    21    5.9    6.9
Mar    18    9.2    9.8
Iu     19    7.4    8.4
```

En aquest exemple, el mètode modifica el `DataFrame` `df` i no retorna res. Si la sentència fos `dfnou = df.set_index('Name')` aleshores el mètode no modifica `df` i retorna un nou `DataFrame`, que s'assigna a la variable `dfnou`.

1.2.1 `DataFrame`: indexació

Per obtenir una fila, una columna o un element d'un `DataFrame` hi ha els atributs `loc` i `iloc`. El primer utilitza les etiquetes, `df.loc[fila, columna]`, i el segon les posicions, `df.iloc[idx_fila, idx_columna]`. Es pot seleccionar tota una fila posant només l'etiqueta (posició) de la fila. En aquest cas retorna una `Series`. Exemples:

```
>>> df.loc['Mar']  # selecció d'una fila, amb etiqueta: una Series
Edat     18.0
Nota1     9.2
Nota2     9.8
Name: Mar, dtype: float64

>>> df.loc['Ana', 'Nota1']   # selecció d'un element, amb etiqueta
8.3
>>> df.iloc[2]
Edat     21.0
Nota1     5.9
Nota2     6.9
Name: Pol, dtype: float64

>>> df.iloc[2, 2]            # selecció d'un element, amb posició
6.9
```

La sintaxi següent indexa directament el `DataFrame` amb l'etiqueta d'una columna i també s'obté una `Series`:

```
>>> df['Edat']       # selecció d'una columna: una Series
Nom
Joe    18
Ana    19
Pol    21
Mar    18
Iu     19
Name: Edat, dtype: int64
```

La forma següent d'obtenir un element és menys eficient que la que s'ha vist amb els atributs `loc` i `iloc`, ja que duu a terme dos passos: primer obté la columna (una `Series`) i després l'element de la columna:

```
>>> df['Edat']['Mar']
18
```

Les files i columnes obtingudes (`Series`) es poden convertir a altres estructures lineals:

```
>>> tuple(df.loc['Mar'])
(18.0, 9.2, 9.8)
>>> dict(df.iloc[2])
{'Edat': 21.0, 'Nota1': 5.9, 'Nota2': 6.9}
>>> list(df['Edat'])
[18, 19, 21, 18, 19]
>>> dict(df['Edat'])
{'Joe': 18, 'Ana': 19, 'Pol': 21, 'Mar': 18, 'Iu': 19}
```

1.2.2 `DataFrame`: subdataframes

Els atributs `loc` i `iloc` i la indexació directa d'un `DataFrame` que acabem de veure s'utilitza també per definir `subdataframes`:

Usant llistes: `df.loc[llista files, llista cols]`. Exemple amb `loc` i `iloc`:

```
>>> df.loc[['Joe', 'Pol'], ['Edat', 'Nota2']]
     Edat  Nota2
Nom
Joe    18    5.1
Pol    21    6.9

>>> df.iloc[[0, 2], [0, 2]]
     Edat  Nota2
Nom
Joe    18    5.1
Pol    21    6.9
```

També es pot usar l'operador per llesques,
`df.iloc[idx_fil_min:idx_fil_max, idx_col_min:idx_col_max]`
i combinar-lo amb la selecció amb llistes:

```
>>> df.loc['Ana':'Mar']      # files entre 'Ana' i 'Mar'
     Edat  Nota1  Nota2
Nom
Ana    19    8.3    3.3
Pol    21    5.9    6.9
Mar    18    9.2    9.8

>>> df.iloc[::2]             # files en posicions parells
     Edat  Nota1  Nota2
Nom
Joe    18    4.1    5.1
Pol    21    5.9    6.9
Iu     19    7.4    8.4
```

```
>>> df.loc['Ana':'Mar', 'Edat':'Nota2']
     Edat  Nota1  Nota2
Nom
Ana    19    8.3    3.3
Pol    21    5.9    6.9
Mar    18    9.2    9.8

>>> df.iloc[1:4, [0, 2]]     # files amb llesca i columnes amb llista
     Edat  Nota2
Nom
Ana    19    3.3
Pol    21    6.9
Mar    18    9.8
```

La indexació directa d'un `DataFrame` només admet la selecció de les columnes mitjançant llistes (no es pot seleccionar amb llesques):

```
>>> df[['Edat', 'Nota2']]['Ana':'Mar']
     Edat  Nota2
Nom
Ana    19    3.3
Pol    21    6.9
Mar    18    9.8
```

1.2.3 `DataFrame`: modificar, inserir i esborrar

La sintaxi usada per indexar i obtenir subdataframes també s'usa per modificar un `DataFrame`:

```
>>> df.loc['Iu', 'Edat'] = 20  # modificar un element
>>> df['Nota1']['Pol'] = 6.5   # modificar un element
>>> df
     Edat  Nota1  Nota2
Nom
Joe    18    4.1    5.1
Ana    19    8.3    3.3
Pol    21    6.5    6.9
Mar    18    9.2    9.8
Iu     20    7.4    8.4

>>> df.loc['Iu'] = [21, 8.4, 7.5]  # modificar una fila
>>> df.loc['Jim'] = [20, 5.7, 8.2]  # inserir fila: 'Edat' passa a ser float
>>> df =df.astype({'Edat':'int64'})  # restaurem el tipus 'Edat' a 'int64'
>>> df['Nota3'] = [4.4, 6.8, 9.2, 8.7, 6.5, 7.2] # inserir columna
>>> df.head(2)
     Edat  Nota1  Nota2  Nota3
Nom
Joe    18    4.1    5.1    4.4
Ana    19    8.3    3.3    6.8
```

La sentència `del` està definida pel tipus `DataFrame`:

```
>>> del df['Nota2']
>>> df.head(2)
     Edat   Nota1   Nota3
Nom
Joe    18     4.1     4.4
Ana    19     8.3     6.8
```

El mètode `drop` permet esborrar. Té els paràmetres opcionals `inplace`, vist anteriorment, i `axis` per indicar files, `axis = 0` (valor per defecte), o columnes, `axis = 1`. A l'exemple següent tornem a afegir la columna `Nota2` abans de tornar-la a esborrar amb el mètode `drop`:

```
>>> df['Nota2'] = [5.1, 3.3, 6.9, 9.8, 8.4, 8.2]
>>> df.drop('Nota2', axis = 1, inplace = True) # esborrat de columna
>>> df.drop('Mar', inplace =True)              # esborrat de fila
>>> df
     Edat   Nota1   Nota3
Nom
Joe    18     4.1     4.4
Ana    19     8.3     6.8
Pol    21     6.5     9.2
Iu     21     8.4     6.5
Jim    20     5.7     7.2
```

El mètode `insert` permet inserir una columna a una posició donada. És sempre un mètode modificador:

```
>>> df.insert(2, 'Nota2', [5.1, 3.3, 6.9, 8.4, 8.2])
>>> df.head(2)
     Edat   Nota1   Nota2   Nota3
Nom
Joe    18     4.1     5.1     4.4
Ana    19     8.3     3.3     6.8
```

El mètode `replace` canvia tots els valors indicats pel nou valor indicat. A l'exemple canviem tots els valors 18 per 20:

```
>>> df.replace(18, 20)
     Edat   Nota1   Nota2   Nota3
Nom
Joe    20     4.1     5.1     4.4
Ana    19     8.3     3.3     6.8
Pol    21     6.5     6.9     9.2
Iu     21     8.4     8.4     6.5
Jim    20     5.7     8.2     7.2
```

Els operadors bàsics (+, -, *, /, >, >=, <, <=, ==, !=) estan definits sobre la classe `Series` i, per tant, es podem aplicar a columnes i files d'un `DataFrame`. A l'exemple següent restaurem el `DataFrame` inicial esborrant la columna `Nota3` i apliquem algunes de les operacions esmentades:

```
>>> del df['Nota3']                          #esborrem la columna 'Nota3
>>> df['Nota'] = df['Nota1']*0.2 + df['Nota2']*0.8  #nova columna
>>> df
     Edat   Nota1   Nota2   Nota
Nom
```

```
Joe     18     4.1     5.1   4.90
Ana     19     8.3     3.3   4.30
Pol     21     6.5     6.9   6.82
Iu      21     8.4     8.4   8.40
Jim     20     5.7     8.2   7.70

>>> df.loc['Pau'] = df.loc['Pol'] + df.loc['Jim']*2  # nova fila
>>> df
      Edat   Nota1   Nota2    Nota
Nom
Joe   18.0     4.1     5.1    4.90
Ana   19.0     8.3     3.3    4.30
Pol   21.0     6.5     6.9    6.82
Iu    21.0     8.4     8.4    8.40
Jim   20.0     5.7     8.2    7.70
Pau   61.0    17.9    23.3   22.22
```

1.2.4 `DataFrame`: selecció i selecció avançada

Tal com s'ha vist amb les `Series` podem seleccionar a partir d'una expressió booleana simple o composta amb els operadors booleans & (`and`), | (`or`) o $\sim$ (`not`). Recordem que quan l'expressió booleana és composta, cadascuna ha d'anar entre parèntesis. Exemples (primer restaurem el `DataFrame`):

```
>>> del df['Nota']
>>> df.drop('Pau', inplace =True)
>>> df =df.astype({'Edat':'int64'})

>>> df[df['Edat'] <= 19]
      Edat   Nota1   Nota2
Nom
Joe     18     4.1     5.1
Ana     19     8.3     3.3

>>> df[(df['Nota1'] > 7.5) | (df['Nota2'] > 7.5)]  # or
      Edat   Nota1   Nota2
Nom
Ana     19     8.3     3.3
Iu      21     8.4     8.4
Jim     20     5.7     8.2

>>> df[(df['Nota1'] > 7.5) & (df['Nota2'] > 7.5)]  # and
      Edat   Nota1   Nota2
Nom
Iu      21     8.4     8.4
```

1.2.5 `DataFrame`: mètodes bàsics

Recordem el `DataFrame` exemple:

```
     Edat   Nota1   Nota2
Nom
Joe    18     4.1     5.1
Ana    19     8.3     3.3
Pol    21     5.9     6.9
Mar    18     9.2     9.8
Iu     19     7.4     8.4
```

Els mètodes `head` i `tail` retornen un `DataFrame` amb les primeres i darreres files del `DataFrame`, respectivament:

```
>>> df.head(2)
     Edat   Nota1   Nota2
Nom
Joe    18     4.1     5.1
Ana    19     8.3     3.3

>>> df.tail(2)
     Edat   Nota1   Nota2
Nom
Mar    18     9.2     9.8
Iu     19     7.4     8.4
```

Els mètodes `sum`, `mean`, `std`, `max`, `min`, `idxmax` i `idxmin` fan els mateixos càlculs que els indicats per les `Series`. Ara bé, per un `DataFrame` aquests mètodes fan el càlcul per totes les columnes i retornen una `Series`.

```
>>> df.sum()
Edat     95.0
Nota1    34.9
Nota2    33.5
dtype: float64

>>> df.idxmax()
Edat     Pol
Nota1    Mar
Nota2    Mar
dtype: object

>>> df.mean()
Edat     19.00
Nota1     6.98
Nota2     6.70
dtype: float64
```

El mètode `describe` retorna un `DataFrame` amb diversos paràmetres estadístics, tal com es veu a l'exemple:

```
>>> df.describe()
            Edat     Nota1     Nota2
count   5.000000  5.000000  5.000000
mean   19.000000  6.980000  6.700000
std     1.224745  2.019158  2.581666
min    18.000000  4.100000  3.300000
25%    18.000000  5.900000  5.100000
50%    19.000000  7.400000  6.900000
```

```
75%    19.000000  8.300000  8.400000
max    21.000000  9.200000  9.800000
```

El mètode `reset_index` converteix l'índex en una columna i el redefineix com un rang d'enters. Té el paràmetre opcional `inplace`.

```
>>> df.reset_index()          # inplace = False (per defecte)
   Nom  Edat  Nota1  Nota2
0  Joe    18    4.1    5.1
1  Ana    19    8.3    3.3
2  Pol    21    5.9    6.9
3  Mar    18    9.2    9.8
4   Iu    19    7.4    8.4
```

Podem posar un nom a les columnes i a l'índex amb els atributs `columns.name` i `index.name`. I amb el mètode `rename` podem canviar el nom de les columnes i l'índex.

```
>>> df.columns.name = 'Data'
>>> df.index.name = 'idx'
>>> df.rename(columns = {'Nota1': 'Parcial', 'Nota2': 'Final'}, inplace =
    True)
>>> df.rename(index = {'Pol': 'Paul', 'Ana': 'Marta'}, inplace = True)
>>> df
Data   Edat  Parcial  Final
idx
Joe      18      4.1    5.1
Marta    19      8.3    3.3
Paul     21      5.9    6.9
Mar      18      9.2    9.8
Iu       19      7.4    8.4
```

El mètode `sort_values` permet ordenar. El paràmetre `by` indica la columna per la que s'ordena. També té els paràmetres `ascending` i `inplace`.

```
>>> df.columns.name = ''      # eliminem nom de columnes
>>> df.index.name = 'Nom'     # restaurem nom de files
>>> df.sort_values(by = 'Edat', ascending = False)
       Edat  Parcial  Final
Nom
Paul     21      5.9    6.9
Marta    19      8.3    3.3
Iu       19      7.4    8.4
Joe      18      4.1    5.1
Mar      18      9.2    9.8
```

El mètode `value_counts` retorna una taula de freqüències dels diferents valors.

```
>>> df['Edat'].value_counts()
19    2
18    2
21    1
Name: Edat, dtype: int64
```

El mètode `append` afegeix un `DataFrame` darrera d'un altre:

```
>>> dfa = pd.DataFrame()
>>> dfa['Nom'] = ['Ona', 'Sira']
```

```
>>> dfa['Edat'] = [21, 18]
>>> dfa['Parcial'] = [5.1, 3.3]
>>> dfa['Final'] = [4.1, 8.3]
>>> dfa.set_index('Nom', inplace = True)
>>> dff = df.append(dfa)
>>> dff
       Edat  Parcial  Final
Nom
Joe      18      4.1    5.1
Marta    19      8.3    3.3
Paul     21      5.9    6.9
Mar      18      9.2    9.8
Iu       19      7.4    8.4
Ona      21      5.1    4.1
Sira     18      3.3    8.3
```

El mètode `apply` permet aplicar una funció a tot el `DataFrame` o a una columna. Aquesta funció pot ser d'una biblioteca (per exemple, `round`), una funció definida explícitament (com la funció `nova_nota` de l'exemple) o una funció anònima (`lambda`). Exemples:

```
>>> df.apply(round).head(2)
       Edat  Parcial  Final
Nom
Joe      18      4.0    5.0
Marta    19      8.0    3.0

>>> df['Final'] = df['Final'].apply(round)
>>> def nova_nota(m):
...     if m >= 5 and m <= 9.5:
...        m = m + 0.5
...     return m
>>> df['Parcial'] = df['Parcial'].apply(nova_nota)
>>> df['Nota'] = df['Parcial']*0.5+df['Final']*0.5
>>> df['Nota'] = df['Nota'].apply(lambda x: round(x*10, 2))
>>> df.head(3)
       Edat  Parcial  Final  Nota
Nom
Joe      18      4.1      5  45.5
Marta    19      8.8      3  59.0
Paul     21      6.4      7  67.0

>>> df.loc['Mar'] = df.loc['Mar'].apply(round)   # apply a una fila
>>> df.tail(3)
      Edat  Parcial  Final  Nota
Nom
Paul    21      6.4      7  67.0
Mar     18     10.0     10  98.0
Iu      19      7.9      8  79.5
```

1.2.6 `DataFrame`: creació

A l'inici s'ha vist com crear un `DataFrame` definint cada columna com una `Series`. També es pot definir un `DataFrame` a partir d'una llista imbricada i a partir d'un diccionari:

```
>>> ld = [['Joe', 18, 4.1, 5.1],
...       ['Ana', 19, 8.3, 3.3],
...       ['Pol', 21, 5.9, 6.9],
...       ['Mar', 18, 9.2, 9.8],
...       ['Iu', 19, 7.4, 8.4]]
>>> idx = ['001', '002', '003', '004', '005']
>>> cols = ['Nom', 'Edat', 'Nota1', 'Nota2']
>>> df = pd.DataFrame(ld, index = idx, columns = cols) # llista imbricada
>>> df
     Nom  Edat  Nota1  Nota2
001  Joe    18    4.1    5.1
002  Ana    19    8.3    3.3
003  Pol    21    5.9    6.9
004  Mar    18    9.2    9.8
005   Iu    19    7.4    8.4

>>> dd = {'Nom': ['Joe', 'Ana', 'Pol', 'Mar', 'Iu'],
...       'Edat': [18, 19, 21, 18, 19],
...       'Nota1': [4.1, 8.3, 5.9, 9.2, 7.4],
...       'Nota2': [5.1, 3.3, 6.9, 9.8, 8.4]}
>>> cols = ['Nom', 'Edat', 'Nota1', 'Nota2']
>>> df = pd.DataFrame(dd, columns = cols)    # diccionari
>>> df
   Nom  Edat  Nota1  Nota2
0  Joe    18    4.1    5.1
1  Ana    19    8.3    3.3
2  Pol    21    5.9    6.9
3  Mar    18    9.2    9.8
4   Iu    19    7.4    8.4
```

1.3 Entrada/sortida

pandas permet llegir i escriure amb diversos formats de fitxer. En aquest text es veu només per fitxers en format `csv` (comma separated values).

La funcions de lectura és:

`df = pd.read_csv(nom, sep=';', usecols = ...)`

`sep` és un paràmetre opcional que val `','` (coma) per defecte. El paràmetre `nom` és un string amb el nom d'un fitxer o d'una url. El paràmetre `usecols` per defecte són totes les columnes, però podem seleccionar les que interessi en una llista. Exemple:

```
>>> url='https://perso.telecom-paristech.fr/eagan/class/igr204/data/cereal.
    csv'
>>> df = pd.read_csv(url, sep=';', usecols=['name', 'calories', 'sugars'])
```

La funció d'escriptura és:

`df.to_csv(filename, sep = ',', index = False)`

El paràmetre opcional `index` és `False` per defecte. Si val `True` escriu els valors de l'índex.

1.4 Mancança de dades

Sovint les bases de dades no són completes i presenten manca d'informació (missing data) en algunes cel·les. pandas permet detectar i tractar aquesta mancança tenint en compte que es pot produir enmig de dades de diversos tipus (enters, reals, strings, ...). Aquesta informació que falta es designa amb els noms NaN (not a number) o NA (not available).

Una `Series` o `DataFrame` incomplet tindrà cel·les amb el valor NaN:

```
   Nom  Edat  Nota1  Nota2
0  Joe  18.0    4.1    5.1
1  Ana  19.0    8.3    3.3
2  Pol  21.0    NaN    6.9
3  Mar   NaN    9.2    9.8
4   Iu  19.0    7.4    8.4
```

Si s'aplica el mètode `count`, es compten només les cel·les que són diferents de NaN. I tal com s'ha indicat en apartats anteriors, els mètodes `sum`, `min`, etc., també ignoren aquestes cel·les.

```
>>> dfn.count()
Nom      5
Edat     4
Nota1    4
Nota2    5
dtype: int64

>>> dfn['Edat'].max()
21.0
>>> dfn['Nota1'].mean()
7.25
```

Per detectar dades que falten hi ha el mètode `isna` (o `isnull`) que converteix tots els elements a un booleà indicant si és NaN (`True`) o no (`False`). El mètode `isna` és equivalent i el mètode `notna` fa el procés invers:

```
>>> dfn.isna()
     Nom   Edat  Nota1  Nota2
0  False  False  False  False
1  False  False  False  False
2  False  False   True  False
3  False   True  False  False
4  False  False  False  False
```

També es disposa de mètodes per tractar els valors NaN. El mètode `fillna` modifica els valors NaN amb un valor indicat i el mètode `dropna` permet esborrar les files o les columnes amb valors NaN: amb el paràmetre `axis=0` (per defecte) esborra files i si `axis=1` esborra columnes. Tots dos mètodes tenen el paràmetre `inplace`:

```
>>> dfn['Nota1'].fillna(0.0, inplace = True)
>>> dfn = dfn.dropna()           # esborra totes les files amb valors NaN
>>> dfn
   Nom  Edat  Nota1  Nota2
0  Joe  18.0    4.1    5.1
1  Ana  19.0    8.3    3.3
2  Pol  21.0    0.0    6.9
4   Iu  19.0    7.4    8.4
```

El `DataFrame` anterior s'ha creat amb les següents sentències:

```
>>> import numpy as np
>>> dfn = pd.DataFrame()
>>> dfn['Nom'] = ['Joe', 'Ana', 'Pol', 'Mar', 'Iu']
>>> dfn['Edat'] = [18, 19, 21, np.nan, 19]
>>> dfn['Nota1'] = [4.1, 8.3, np.nan, 9.2, 7.4]
>>> dfn['Nota2'] = [5.1, 3.3, 6.9, 9.8, 8.4]
```

Observem dues coses. Per definir el valor NaN hem usat la constant `np.nan` de la biblioteca `numpy` que hem importat. Per altra banda, la columna `Edat` té tots els valors enters, però degut a que en té un que és un NaN, la representa com a reals. Per tal de fer que siguin enters, podem usar el mètode `astype`:

```
>>> dfn = dfn.astype({'Edat':'int64'})
>>> dfn
   Nom  Edat  Nota1  Nota2
0  Joe    18    4.1    5.1
1  Ana    19    8.3    3.3
2  Pol    21    0.0    6.9
4   Iu    19    7.4    8.4

>>> dfn['Edat'].sum()
77
```

Finalment, quan en la creació d'un `DataFrame` hi manquen dades, pandas omple automàticament aquestes mancances amb valors NaN.

1.5 `DataFrame`: mètode `groupby`

El concepte d'agrupar (*group by*, en anglès) fa referència a un procés que aplica diversos passos:

- Subdividir (*split*) les dades en grups, utilitzant un determinat criteri
- Aplicar una funció a cada grup
- Combinar els resultats en una nova estructura de dades

Per exemple, suposem que tenim el següent `DataFrame`, on `Gclas` indica el grup de classe de l'estudiant:

```
>>> dfest
      Nom  Edat Gclas  Nota
0     Pol    23     A   5.6
1    Pere    22     A   3.5
2     Pau    23     B   2.1
3    Joan    22     A   6.7
4   Júlia    22     C   8.7
5    Laia    21     A   4.5
6    Rosa    21     B   6.7
7  Alícia    22     C   9.1
```

Si fem grups pel grup de classe tindrem 3 grups, un per cada grup de classe diferent. Aleshores a cada grup li podem aplicar una operació com, per exemple, calcular la nota mitjana i el resultat serà una `Series` amb la nota mitjana de cada grup de classe.

El mètode `groupby` realitza una distribució de la base de dades, agrupa per una o més columnes, claus, (`key`) i torna un objecte de tipus `groupby`. Per fer un agrupament múltiple les diverses claus s'indiquen amb una llista de claus.

Un objecte de tipus `groupby` és una col·lecció de tuples de la forma `(clau, DataFrame)` i és iterable.

Continuant amb l'exemple agrupem per la columna `Gclas`, és a dir, per grup de classe:

```
>>> gb1 = dfest.groupby('Gclas')  # equival a  dfest.groupby(dfest['Gclas'])
>>> type(gb1)
<class 'pandas.core.groupby.groupby.DataFrameGroupBy'>
>>> len(gb1)
3
```

Veiem que `gb1` és un objecte de tipus `groupby` amb 3 grups. Ara mostrem aquests grups iterant per la col·lecció de tuples `(clau, DataFrame)`:

```
>>> for clau, df in gb1:
...     print(clau)
...     print(df)
...
A
    Nom  Edat Gclas  Nota
0   Pol    23     A   5.6
1  Pere    22     A   3.5
3  Joan    22     A   6.7
5  Laia    21     A   4.5
B
    Nom  Edat Gclas  Nota
2   Pau    23     B   2.1
6  Rosa    21     B   6.7
C
      Nom  Edat Gclas  Nota
4   Júlia    22     C   8.7
7  Alícia    22     C   9.1
```

Observem que a dalt a l'esquerra es mostra la clau corresponent al grup de classe ('A', 'B' o 'C') i a sota el `DataFrame` corresponent.

Si només volem accedir a un dels grups obtinguts, el mètode `get_group` permet seleccionar el

`DataFrame` corresponent a la clau donada. Aquest mètode retorna un `DataFrame`. A l'exemple següent obtenim el grup corresponent al grup de classe `'A'`, seleccionem la columna `Nota` a la qual s'aplica (amb el mètode `apply`) la funció `round`.

```
>>> gb1.get_group('A')        # retorna un DataFrame
    Nom  Edat Gclas  Nota
0   Pol    23     A   5.6
1  Pere    22     A   3.5
3  Joan    22     A   6.7
5  Laia    21     A   4.5

>>> gb1.get_group('A')['Nota'].apply(round)
0    6
1    4
3    7
5    4
Name: Nota, dtype: int64
```

Si volem fer un estudi de tots els grups, després d'aquest primer pas de subdivisió del `DataFrame` continuarem amb els passos següents d'aplicar una funció i combinar els resultats. A l'exemple següent, apliquem el mètode `size` a cada `DataFrame` de l'objecte `groupby` i es combinen els resultats obtenint una `Series` amb la mida de cada `DataFrame`:

```
>>> gb1.size()
Gclas
A    4
B    2
C    2
dtype: int64
```

Els mètodes amb els que podem fer el mateix procés són: `first` (la 1a fila de cada `DataFrame`), `last` (la darrera) `nth` (la fila enèsima) i els mètodes ja vistos `sum`, `min`, `idxmin`, `count`, `mean`, etc. Al següent exemple s'aplica el mètode `first` al `groupby` obtenint un `DataFrame` amb les primeres files de cada `DataFrame` del `groupby`. Quan apliquem el mètode `sum`, només suma las columnes numèriques:

```
>>> gb1.first()               # equival a gb1.nth(0)
         Nom  Edat  Nota
Gclas
A        Pol    23   5.6
B        Pau    23   2.1
C      Júlia    22   8.7

>>> gb1.sum()
       Edat  Nota
Gclas
A        88  20.3
B        44   8.8
C        44  17.8
```

També es pot seleccionar una columna en un objecte `groupby` i aplicar el mètode només a la columna. Al següent exemple es calcula la nota mitjana de cada grup de classe:

```
>>> gb1['Nota'].mean()
Gclas
```

```
A    5.075
B    4.400
C    8.900
Name: Nota, dtype: float64
```

La sentència `gb1.mean()['Nota']` dóna el mateix resultat que l'anterior (`gb1['Nota'].mean()`) però és menys eficient ja que primer calcula un `DataFrame` amb les mitjanes de totes les columnes numèriques i després selecciona la columna `Nota`.

A continuació es mostra un altre exemple. Donat un `DataFrame` com el de l'inici d'aquest apartat, volem classificar els estudiants per edat i obtenir, per cada edat diferent, l'estudiant que ha obtingut la nota màxima.

Primer, amb el mètode `set_index` fem que la columna `Nom` sigui l'índex en el `DataFrame` donat, després subdividim per `Edat` i finalment seleccionem la columna `Nota` i apliquem el mètode `idxmax`. Per tal que aquest mètode ens retorni el nom en lloc de l'índex numèric inicial, hem fet el canvi d'índex inicial.

```
>>> df1 = dfest.set_index('Nom')
>>> gb2 = df1.groupby('Edat')
>>> for clau, df in gb2:
...     print(clau)
...     print(df)
...
21
      Edat  Gclas  Nota
Nom
Laia    21      A   4.5
Rosa    21      B   6.7
22
        Edat  Gclas  Nota
Nom
Pere      22      A   3.5
Joan      22      A   6.7
Júlia     22      C   8.7
Alícia    22      C   9.1
23
     Edat  Gclas  Nota
Nom
Pol    23      A   5.6
Pau    23      B   2.1

>>> gb2['Nota'].idxmax()
Edat
21      Rosa
22    Alícia
23       Pol
Name: Nota, dtype: object
```

Un objecte `groupby` té un atribut anomenat `groups` que és un diccionari on les claus del diccionari són les claus del `groupby` i els valors són l'índex de cada grup.

```
>>> gb2.groups
{21: Index(['Laia', 'Rosa'], dtype='object', name='Nom'), 22: Index(['Pere',
    'Joan', 'Júlia', 'Alícia'], dtype='object', name='Nom'), 23: Index(['Pol'
    , 'Pau'], dtype='object', name='Nom')}
```

```
>>> gb2.groups.keys()
dict_keys([21, 22, 23])

>>> gb2.groups[22]
Index(['Pere', 'Joan', 'Júlia', 'Alícia'], dtype='object', name='Nom')
```

1.5.1 `DataFrame`: groupby jeràrquic

La subdivisió d'un `DataFrame` amb el mètode `groupby` es pot fer amb més d'un criteri (clau). En aquest cas, el mètode `groupby` rep una llista de claus i la subdivisió múltiple es fa de forma jeràrquica començant per la clau que està en primer lloc i continuant amb les altres: `groupby([clau1, clau2, ...])`.

En el següent exemple, partim del `DataFrame` `dfest`, definit en aquest apartat, i fem una subdivisió jeràrquica primer per grup de classe i després per edat. Com que hi ha 3 grups de classe i 3 edats diferents hi ha 9 grups potencials, però només 6 no són nuls. En aquest cas el `groupby` és una col·lecció de tuples de la forma `(clau_mult, DataFrame)` on `clau_mult` és la clau múltiple i es representa amb un tuple amb les dues claus donades.

```
>>> gb3 = dfest.groupby(['Gclas', 'Edat'])
>>> for (clau1, clau2), df in gb3:
...     print(clau1, clau2)
...     print(df)
...
A 21
    Nom  Edat Gclas  Nota
5  Laia    21     A   4.5
A 22
    Nom  Edat Gclas  Nota
1  Pere    22     A   3.5
3  Joan    22     A   6.7
A 23
   Nom  Edat Gclas  Nota
0  Pol    23     A   5.6
B 21
    Nom  Edat Gclas  Nota
6  Rosa    21     B   6.7
B 23
   Nom  Edat Gclas  Nota
2  Pau    23     B   2.1
C 22
      Nom  Edat Gclas  Nota
4   Júlia    22     C   8.7
7  Alícia    22     C   9.1

>>> gb3.size()
Gclas  Edat
A      21      1
       22      2
       23      1
B      21      1
       23      1
C      22      2
```

```
dtype: int64
```

Veiem que l'estructura de `gb3.size()` és una `Series` multi-indexada.

1.5.2 `DataFrame`: aggregate (agg)

Amb el mètode `agg` del tipus `groupby` podem aplicar diferents mètodes (o funcions) a cada columna dels objectes `DataFrame` obtinguts amb el `groupby`. Cal un diccionari per indicar quins mètodes s'han d'aplicar a cada columna. El resultat és un `DataFrame` jeràrquic multi-indexat. Exemple:

```
>>> gb1.agg({'Edat':['max', 'min'], 'Nota': ['sum', 'size', 'mean']})
      Edat       Nota
       max min   sum size   mean
Gclas
A       23  21  20.3    4  5.075
B       23  21   8.8    2  4.400
C       22  22  17.8    2  8.900
```

1.6 Operacions amb strings

Per operar amb strings no podem usar els mètodes de tipus `str` ni l'operador `in`. Cal usar els mètodes específics de la biblioteca `pandas`. Aquests mètodes exclouen els valors NaN de forma automàtica. S'accedeixen a través de l'atribut `str` i tenen noms i especificacions iguals o similars als dels seus homòlegs del tipus `str` de `Python`:

```
>>> dfest
     nom  edat grup  nota
0    Pol    23    A   5.6
1   Pere    22    A   3.5
2   Laia    23    B   2.1
3   Joan    22    A   6.7
4  Júlia    22    C   8.7

>>> dfest['nom'].str.len()    # longitud dels noms
0    3
1    4
2    4
3    4
4    5
Name: nom, dtype: int64

>>> dfest['nom'].str.lower()           # posa els noms en minúsciles
0      pol
1     pere
2     laia
3     joan
4    júlia
Name: nom, dtype: object
```

```
>>> dfest['nom'].str.split('a')         # split per lletra 'a'
0       [Pol]
1      [Pere]
2    [L, i, ]
3     [Jo, n]
4    [Júli, ]
Name: nom, dtype: object

>>> dfest['nom'].str.replace('a', '@')
0      Pol
1     Pere
2     L@i@
3     Jo@n
4    Júli@
Name: nom, dtype: object
```

El mètode `str.cat` fa una funció equivalent al mètode `join` dels strings i el mètode `contains` equival a l'operador `in`:

```
>>> dfest['nom'].str.cat(sep = ' - ')
'Pol - Pere - Laia - Joan - Júlia'

>>> dfest['nom'].str.contains('o')      # Series amb True o False
0     True
1    False
2    False
3     True
4    False
Name: nom, dtype: bool

>>> dfest['nom'].str[0]                 # les inicials
0    P
1    P
2    L
3    J
4    J
Name: nom, dtype: object

>>> dfest['grup'].str.match('B')       # equival a dfest['grup'] == 'B'
0    False
1    False
2     True
3    False
4    False
Name: grup, dtype: bool
```

1.7 Mètodes `where` i `mask`

Aquests mètodes permeten modificar o crear un nou `DataFrame` amb una modificació selectiva.

L'esquema que segueix el mètode `where` és:

```
if condicio:
    mantenir_valor
else:
    modificar_valor
```

En el següent exemple es modifiquen selectivamenent les columnes `Edat` i `Pobl` del `DataFrame` `df1`. Tots els elements de la columna `Edat` inferiors a 50 no es modifiquen, i els superiors o iguals a 50 es modifiquem amb el valor 55. A la columna `Pobl` tots els elements amb un nom de més de 4 caràcters es modifiquen i passen a tenir el valor `'Gurb'`.

```
>>> df1
      Nom  Edat     Pobl
0    Joan    23      Vic
1    Anna    12  Cardona
2     Roc    56     Reus
3   Paula    55     Flix
4   Berta    37   Mataró
5  Carles    45     Seva
6   Diana    58    Rupit

>>> df1['Edat'] = df1['Edat'].where(df1['Edat']<50, 55)
>>> df1['Pobl'] = df1['Pobl'].where(df1['Pobl'].str.len() <=4, 'Gurb')
>>> df1
      Nom  Edat  Pobl
0    Joan    23   Vic
1    Anna    12  Gurb
2     Roc    55  Reus
3   Paula    55  Flix
4   Berta    37  Gurb
5  Carles    45  Seva
6   Diana    55  Gurb
```

L'esquema que segueix el mètode `mask` és el contrari del que segueix `where`:

```
if condicio:
    modificar_valor
else:
    mantenir_valor
```

Exemple on es modifica la columna `Edat`:

```
>>> df1
      Nom  Edat     Pobl
0    Joan    23      Vic
1    Anna    12  Cardona
2     Roc    56     Reus
3   Paula    55     Flix
4   Berta    37   Mataró
5  Carles    45     Seva
6   Diana    58    Rupit

>>> df1['Edat'] = df1['Edat'].mask(df1['Edat']<50, 20)
>>> df1
      Nom  Edat     Pobl
0    Joan    20      Vic
1    Anna    20  Cardona
2     Roc    56     Reus
3   Paula    55     Flix
4   Berta    20   Mataró
5  Carles    20     Seva
6   Diana    58    Rupit
```

1.8 Funcions `merge` i `concat`

La funció `merge` permet fusionar dos objectes `DataFrame`. El paràmetre opcional `on` indica la columna que comparteixen els `DataFrame`. El paràmetre opcional `how` té els valors i significats següents:

- `left`: torna només les files del 1r `DataFrame` donat
- `right`: torna només les files del 2n `DataFrame` donat
- `outer`: torna totes les files de tots dos `DataFrame` (fa la unió dels `DataFrame`)
- `inner` (valor per defecte): torna només les files compartides pels dos `DataFrame` (fa la intersecció dels `DataFrame`)

En el següent exemple tenim dos `DataFrame` que comparteixen la columna corresponent al nom d'una persona tot i que no comparteixen les mateixes persones. En el primer, a més, hi ha l'edat i població de residència. En el segon, hi ha la ciutat que voldrien visitar i el mitja de transport que voldrien usar:

```
>>> df1
     Nom  Edat     Pobl
0   Joan    23      Vic
1   Anna    56  Cardona
2    Roc    12     Reus
3  Paula    37     Flix
4   Erna    25    Canet

>>> df2
     Nom     ciutat transport
0   Joan    Londres   autocar
1  Marta     Atenes     cotxe
2    Roc      Hanoi      avió
3  Paula      París      tren
4   Marc  Nova York      avió
```

A continuació apliquem la funció `merge` amb diversos valors del paràmetre `how`:

```
>>> pd.merge(df1, df2, on='Nom')       # how = 'inner' (valor per defecte)
     Nom  Edat  Pobl   ciutat transport
0   Joan    23   Vic  Londres   autocar
1    Roc    12  Reus    Hanoi      avió
2  Paula    37  Flix    París      tren

>>> pd.merge(df1, df2, on='Nom', how = 'outer')
     Nom  Edat     Pobl     ciutat transport
0   Joan  23.0      Vic    Londres   autocar
1   Anna  56.0  Cardona        NaN       NaN
2    Roc  12.0     Reus      Hanoi      avió
3  Paula  37.0     Flix      París      tren
4   Erna  25.0    Canet        NaN       NaN
5  Marta   NaN      NaN     Atenes     cotxe
6   Marc   NaN      NaN  Nova York      avió
```

```
>>> pd.merge(df1, df2, on='Nom', how = 'left')
     Nom  Edat     Pobl   ciutat transport
0   Joan    23      Vic  Londres   autocar
1   Anna    56  Cardona      NaN       NaN
2    Roc    12     Reus    Hanoi      avió
3  Paula    37     Flix    París      tren
4   Erna    25    Canet      NaN       NaN
```

Observem com en fer la fusió hi ha cel·les del nou `DataFrame` que no tenen el valor definit i `pandas` els hi posa el valor `NaN` de forma automàtica.

La funció `concat` concatena dos `DataFrame`. El paràmetre opcional `axis` permet indicar si la concatenació és vertical (`axis = 0`, per defecte) o horitzontal (`axis = 1`). Exemples:

```
>>> pd.concat([df1, df2], sort = False)         # vertical (axis = 0)
     Nom  Edat     Pobl     ciutat transport
0   Joan  23.0      Vic        NaN       NaN
1   Anna  56.0  Cardona        NaN       NaN
2    Roc  12.0     Reus        NaN       NaN
3  Paula  37.0     Flix        NaN       NaN
4   Erna  25.0    Canet        NaN       NaN
0   Joan   NaN      NaN    Londres   autocar
1  Marta   NaN      NaN     Atenes     cotxe
2    Roc   NaN      NaN      Hanoi      avió
3  Paula   NaN      NaN      París      tren
4   Marc   NaN      NaN  Nova York      avió

>>> pd.concat([df1, df2], axis = 1)          # horitzontal (axis = 1)
     Nom  Edat     Pobl    Nom     ciutat transport
0   Joan    23      Vic   Joan    Londres   autocar
1   Anna    56  Cardona  Marta     Atenes     cotxe
2    Roc    12     Reus    Roc      Hanoi      avió
3  Paula    37     Flix  Paula      París      tren
4   Erna    25    Canet   Marc  Nova York      avió
```

El paràmetre opcional `sort` es pot posar a `True` i aleshores ordena les columnes (`axis = 1`) o les files (`axis = 0`).

Nota: Vegeu també els mètodes `merge` i `concat`.

1.9 Mètodes `stack` i `unstack`

Aquests dos mètodes permeten canviar l'estructura d'un `DataFrame` pivotant o canviant la jerarquia.

El mètode `stack` retorna un `DataFrame` multi-index, on l'índex del `DataFrame` original passa a ser l'índex principal i les columnes passen a ser el 2n índex. En el següent exemple el mètode `stack` retorna una `Series` multi-indexada:

```
>>> df
      nom  edat grup  nota
0    Joan    22    A   6.7
1   Júlia    22    C   8.7
2    Laia    21    A   4.5
3    Rosa    21    B   6.7
4  Alícia    22    C   9.1

>>> df1 = df.stack()
>>> df1
0  nom        Joan
   edat         22
   grup          A
   nota        6.7
1  nom       Júlia
   edat         22
   grup          C
   nota        8.7
2  nom        Laia
   edat         21
   grup          A
   nota        4.5
3  nom        Rosa
   edat         21
   grup          B
   nota        6.7
4  nom      Alícia
   edat         22
   grup          C
   nota        9.1
dtype: object
```

El mètode `unstack` fa l'operació inversa. La sentència `df1.unstack()` retorna un `DataFrame` igual que el `df` inicial.

En aquest altre exemple es crea un `DataFrame` multi-índex on el primer índex és el grup de classe i el segon és l'edat. Usem el mètode `from_tuples` per crear un multi-índex d'aquestes característiques i apliquem el mètode `unstack`:

```
>>> grup = ['A', 'A', 'A', 'A', 'B', 'B', 'B']
>>> edat = [22, 23, 21, 20, 23, 22, 21]
>>> tuples = list(zip(grup, edat))
>>> multindex = pd.MultiIndex.from_tuples(tuples, names = ['grup', 'edat'])
>>> dfa = pd.DataFrame(index = multindex)
>>> dfa['nom'] = ['Jan', 'Ann', 'Iu', 'Roc', 'Bea', 'Pau', 'Mar']
>>> dfa['nota'] = [6.7, 8.7, 4.5, 6.7, 9.1, 3.2, 5.5]
>>> dfa
           nom  nota
grup edat
A    22    Jan   6.7
     23    Ann   8.7
     21     Iu   4.5
     20    Roc   6.7
B    23    Bea   9.1
     22    Pau   3.2
     21    Mar   5.5
```

```
>>> dff = dfa.unstack()
>>> dff
      nom                 nota
edat   20   21   22   23   20   21   22   23
grup
A     Roc   Iu  Jan  Ann  6.7  4.5  6.7  8.7
B     NaN  Mar  Pau  Bea  NaN  5.5  3.2  9.1
```

Observem, en primer lloc, que també apareixen cel·les amb valors no definits. En fer la reestructuració del `DataFrame` inicial com que no hi ha cap estudiant del grup B que tingui 20 anys, al lloc corresponent al seu nom i nota hi ha el valor `NaN`.

En segon lloc observem que el mètode `unstack` ha convertit l'índex segon (`Edat`) en les columnes. Per defecte, aquest mètode agafa l'índex més intern. La sentència `dff = dfa.unstack()` és equivalent a `dff = dfa.unstack('Edat')`. Ara bé, l'índex es pot seleccionar. En el següent exemple, apliquem el mètode `unstack` a l'altre índex (grup):

```
>>> dfg = dfa.unstack('grup')
>>> dfg
      nom       nota
grup    A    B    A    B
edat
20    Roc  NaN  6.7  NaN
21     Iu  Mar  4.5  5.5
22    Jan  Pau  6.7  3.2
23    Ann  Bea  8.7  9.1
```

A continuació es mostra un altre exemple on partim del `DataFrame dfa` amb dos índexs i li apliquem el mètode `stack`: obtenim una `Series` amb 3 índexs, `sb`:

```
>>> dfa
            nom  nota
grup edat
A    22     Jan   6.7
     23     Ann   8.7
     21      Iu   4.5
     20     Roc   6.7
B    23     Bea   9.1
     22     Pau   3.2
     21     Mar   5.5

>>> sb= dfa.stack()
>>> sb                 #  Series amb 3 indexs
grup  edat
A     22    nom      Jan
            nota     6.7
      23    nom      Ann
            nota     8.7
      21    nom       Iu
            nota     4.5
      20    nom      Roc
            nota     6.7
B     23    nom      Bea
            nota     9.1
      22    nom      Pau
```

```
            nota    3.2
      21    nom     Mar
            nota    5.5
dtype: object
```

A continuació, apliquem a aquesta `Series` el mètode `unstack`. Primer fent servir l'índex més intern que correspon a les columnes (obtenim el `DataFrame` `dfc`); després fem servir l'índex `edat` (`DataFrame` `dfd`), i finalment l'índex `grup` (`DataFrame` `dfe`).

```
>>> dfc = sb.unstack()
>>> dfc
           nom nota
grup edat
A    20    Roc  6.7
     21     Iu  4.5
     22    Jan  6.7
     23    Ann  8.7
B    21    Mar  5.5
     22    Pau  3.2
     23    Bea  9.1

>>> dfd = sb.unstack('edat')
>>> dfd
edat         20   21   22   23
grup
A    nom    Roc   Iu  Jan  Ann
     nota   6.7  4.5  6.7  8.7
B    nom    NaN  Mar  Pau  Bea
     nota   NaN  5.5  3.2  9.1

>>> dfe = sb.unstack('grup')
>>> dfe
grup          A    B
edat
20   nom    Roc  NaN
     nota   6.7  NaN
21   nom     Iu  Mar
     nota   4.5  5.5
22   nom    Jan  Pau
     nota   6.7  3.2
23   nom    Ann  Bea
     nota   8.7  9.1
```