09-Introdução ao Pandas

July 29, 2020

1 Introdução

Pandas é uma biblioteca essencial para a Ciência e Análise de Dados com Python. Grande parte do trabalho de um cientista de dados é a preparação de dados. A etapa de preparação dos dados pode exigir a sua manipulação de diversas maneiras; por exemplo, o tratamento de dados ausentes. O Pandas fornece um conjunto de ferramentas para análise e para a manipulação de dados.

Assim como muitas bibliotecas de computação cientifica o Pandas depende da biblioteca NumPy. Veremos alguns detalhes do NumPy na próxima semana. O Pandas também depende das bibliotecas **python-dateutil** e **pytz** para lidar com datas e fusos horários (*timezones*), respectivamente.

1.1 NumPy

O NumPy é um dos pacotes/bibliotecas base para a computação cientifica, muitos outros pacotes/bibliotecas usam as funcionalidades do NumPy, e principalmente, usam os tipos de dados do NumPy como uma língua franca para a troca de dados entre diferentes pacotes/bibliotecas.

O NumPy oferece um *array* multidimensional muito eficiente, **ndarray**, uma série de funções matemáticas eficientes sobre esses *arrays* (sem a necessidade de fazer loops) e ferramentas para a leitura e escrita eficiente destes *arrays*. O NumPy ainda oferece implementações de álgebra linear, geração de números aleatórios e transformada de Fourier.

Algumas funções do NumPy que usaremos (argumentos entre colchetes[] são opcionais):

- np.arange([inicio,] fim, [passo,] dtype=None): retorna valores igualmente espaçados iniciando em **inicio** e terminando *antes* de **fim** com o intervalo **passo**. **dtype** define o tipo de dados dos elementos, se não for passado o tipo será inferido pelo tipo dos outros argumentos.
- np.reshape(a, newshape, ...): Dá um novo formato, definido por **newshape**, ao *array* **a**, sem alterar os dados. **newshape** pode ser um inteiro para um valor de uma dimensão (1D) ou uma tupla para definir multimedições.
- np.exp(x, ...): calcula o exponencial natural de cada elemento do **ndarray** x (x também pode ser um escalar). O exponencial natural de um número y é o número de Euler, e, elevado a y (e**y).

1.2 Importando Pandas e NumPy

Ao importar a biblioteca Pandas, usualmente atribuímos um nome local mais curto **pd**, e para o NumPy usamos **np**, como mostrado abaixo:

[1]:
```
import pandas as pd
import numpy as np
```

2 Estruturas de dados

As duas principais estruturas de dados do Pandas são **Series** e **DataFrame (DF)**. Você pode preferir importar estas duas estruturas diretamente:

[2]:
```
from pandas import Series, DataFrame
```

Desta forma você pode escolher entre usar:

```
sr = pd.Series(...)
df = pd.DataFrame(...)
```

Ou, equivalentemente:

```
sr = Series(...)
df = DataFrame(...)
```

2.1 Series

Uma **Series** é um objeto de uma dimensão (1D) parecido com um *array* que contém uma sequencia de valores (de tipos equivalentes aos tipos da biblioteca **NumPy**) e um *array* de rótulos (*labels*) dos dados, chamado *index*.

Podemos construir uma **Series** com base nos valores de uma **list** conforme abaixo:

[3]:
```
sr1 = Series([1, 3, -5, 7])
print(sr1)
```

```
0    1
1    3
2   -5
3    7
dtype: int64
```

Observe que ao imprimir temos uma coluna com os *index* e outra como os *values* e ainda o *dtype* que informa o tipo dos dados em *value*. Podemos acessar cada um destes campos individualmente:

[4]:
```
print(sr1.index)
print(sr1.values)
print(sr1.dtype)
```

```
RangeIndex(start=0, stop=4, step=1)
[ 1  3 -5  7]
int64
```

Podemos criar uma **Series** especificando os valores de cada *index*, que podem inclusive ser *strings*:

[5]:
```
sr2 = Series([1, 3, -5, 7], index=['a', 'b', 'c', 'd'])
print(sr2)
```

```
a    1
b    3
c   -5
```

```
d    7
dtype: int64
```

Podemos utilizar estes *index* para selecionar os respectivos valores:

[6]:
```
print(sr1[0])
print(sr1[1])
print(sr1[3])

print("Na sr2:")
print(sr2['a'])
print(sr2['c'])
```

```
1
3
7
Na sr2:
1
-5
```

2.1.1 Aplicando operações

Podemos aplicar uma série de operações (filtro de de *array* booleano, multiplicação por escalar e operações no **NumPy**). E mesmo com filtros utilizando *array* booleano os índices são preservados, observe os exemplos abaixo:

[7]:
```
print(sr2[sr2 > 0])  # Filtra mantendo apenas os positivos não negativos
```

```
a    1
b    3
d    7
dtype: int64
```

[8]:
```
print(sr2 * 2)  # Multiplicação por escalar
```

```
a     2
b     6
c   -10
d    14
dtype: int64
```

[9]:
```
import numpy as np
print(np.exp(sr2))  # usando funções do NumPy
```

```
a       2.718282
b      20.085537
c       0.006738
d    1096.633158
dtype: float64
```

2.1.2 Relação com dicionários e listas

Podemos testar a pertinência de um determinado elemento no *index* utilizando **in**, da mesma maneira que fazemos com chaves em dicionários ou valores em listas

[10]:
```
print('a' in sr2)
print('y' in sr2)
```

```
True
False
```

Podemos converter um dicionário em uma **Series**, chaves são mapeadas como *index* e os valores para *value*:

[11]:
```
dict_data = {'a': 12, 'b': 65, 'c': -12, 'd': 42}
sr3 = Series(dict_data)
print(sr3)
```

```
a    12
b    65
c   -12
d    42
dtype: int64
```

Quando construindo uma **Series** com base em um **dict** o *index* da **Series** será por padrão construído pela ordenação das chaves do **dict**. Porém podemos sobrescrever este padrão passando as chaves na ordem desejada.

[12]:
```
sr4 = Series(dict_data, index=['e', 'c', 'b', 'a'])
print(sr4)
```

```
e     NaN
c   -12.0
b    65.0
a    12.0
dtype: float64
```

Observe que incluímos a chave **'e'** que não possui um valor definido no dicionário e receberá valor **NaN** (Not a Number) na **Series**.

Observe também que não incluímos em **index** a chave **'d'** que não é portanto incluída na **Series** *sr4*.

2.1.3 NaN

NaN (Not a Number) é utilizado pelo Pandas para marcar valores não existentes, isto é, valores ausentes. Para detectar dados ausentes temos as seguintes duas funções que retornam uma **Series** com valores booleanos (**True** e **False**):

- **isnull**: Valor **True** para dados ausentes e **False** caso contrário; e
- **notnull**: Valor **True** para dados presentes e **False** caso contrário.

[13]:
```
print(sr4.isnull())
```

```
e     True
c    False
b    False
a    False
dtype: bool
```

[14]:
```
print(sr4.notnull())
```

```
e    False
c     True
b     True
a     True
dtype: bool
```

Alternativamente podemos chamar estes métodos como funções passando a **Series** como parâmetro:

[15]:
```
print(pd.isnull(sr4))
print(pd.notnull(sr4))
```

```
e     True
c    False
b    False
a    False
dtype: bool
e    False
c     True
b     True
a     True
dtype: bool
```

2.1.4 Alinhamento de dados com NaN

O Pandas alinha os *index* automaticamente com operações, observe o que acontece quando somamos *sr3* com *sr4*:

[16]:
```
print("sr3:")
print(sr3)
print("sr4:")
print(sr4)
print("sr3 + sr4:")
print(sr3 + sr4)
```

```
sr3:
a    12
b    65
c   -12
d    42
```

```
dtype: int64
sr4:
e     NaN
c   -12.0
b    65.0
a    12.0
dtype: float64
sr3 + sr4:
a     24.0
b    130.0
c    -24.0
d      NaN
e      NaN
dtype: float64
```

Observe que os valores nas posições: 'a', 'b' e 'c' foram somados conforme o valor do seu *index* (e não da sua posição/ordem na **Series**)

Observe também que se um dos valores está ausente (tanto não existir quando valer **NaN**) o resultado é sempre **NaN**. Assim, temos que ter em mente que: Qualquer operação aritmética envolvendo **NaN** resulta em **NaN**.

Você pode pensar em alinhamento de *index* como similar a uma operação de **join** em um banco de dados.

2.1.5 name

Tanto a **Series** quanto seu *index* possuem um atributo **name**. Este atributo se integra com outras partes do Pandas.

[17]:
```
sr4.name = 'Frequência'
sr4.index.name = 'Letra'

print(sr4)
```

```
Letra
e     NaN
c   -12.0
b    65.0
a    12.0
Name: Frequência, dtype: float64
```

2.1.6 Alterando o *index in-place*

Podemos alterar o *index in-place*:

[18]:
```
print(sr1)
sr1.index = ['id 1', 'id 2', 'id 3', 'outro id']
print(sr1)
```

```
0    1
1    3
2   -5
3    7
dtype: int64
id 1        1
id 2        3
id 3       -5
outro id    7
dtype: int64
```

2.2 DataFrame (DF)

Um **DataFrame** (DF) representa uma tabela de dados e contém uma coleção ordenada de colunas. Cada coluna pode ser de um tipo diferente (numérica, *string*, booleana, ...). O DF possui dois índices: linha (*index*) e coluna (*columns*).

Há varias maneiras de se construir um DF, a mais usual é com base em um dicionário de listas, aonde cada entrada no dicionário é uma coluna no DF:

[19]:
```
data = {
    'estado': ['Paraná', 'Santa Caratina', 'Rio Grande do Sul', 'São Paulo',␣
 ↪'Rio de Janeiro', 'Espirito Santos', 'Minas Gerais'],
    'UF': ['PR', 'SC', 'RS', 'SP', 'RJ', 'ES', 'MG'],
    'pop': [11433957, 7164788, 11377239, 45919049, 17264943, 4018650,␣
 ↪21168791],  # população
    'ano': [2019, 2019, 2018, 2019, 2019, 2018, 2019],  # Ano da estimativa da␣
 ↪população
}

df1 = DataFrame(data)
print(df1)
```

```
              estado  UF       pop   ano
0             Paraná  PR  11433957  2019
1     Santa Caratina  SC   7164788  2019
2  Rio Grande do Sul  RS  11377239  2018
3          São Paulo  SP  45919049  2019
4     Rio de Janeiro  RJ  17264943  2019
5    Espirito Santos  ES   4018650  2018
6       Minas Gerais  MG  21168791  2019
```

Da mesma forma que acontece com **Series** o *index* do DF (índice de linha) é preenchido automaticamente em ordem crescente iniciando em 0.

Quando trabalhamos com muitos dados podemos mostrar apenas os **n** primeiros com **.head(n)**, se **n** o padrão é 5:

[20]:
```
print(df1.head())
```

```
              estado  UF       pop   ano
0             Paraná  PR  11433957  2019
1     Santa Caratina  SC   7164788  2019
2  Rio Grande do Sul  RS  11377239  2018
3          São Paulo  SP  45919049  2019
4     Rio de Janeiro  RJ  17264943  2019
```

[21]:
```
print(df1.head(3))
```

```
              estado  UF       pop   ano
0             Paraná  PR  11433957  2019
1     Santa Caratina  SC   7164788  2019
2  Rio Grande do Sul  RS  11377239  2018
```

Ao criar um DF com base em um **dict** (no caso abaixo usamos o **dict** *data* já definido acima) podemos especificar quais colunas e qual a ordem das colunas queremos:

[22]:
```
df2 = DataFrame(data, columns=['UF', 'pop', 'estado'])
print(df2)
```

```
   UF       pop             estado
0  PR  11433957             Paraná
1  SC   7164788     Santa Caratina
2  RS  11377239  Rio Grande do Sul
3  SP  45919049          São Paulo
4  RJ  17264943     Rio de Janeiro
5  ES   4018650    Espirito Santos
6  MG  21168791       Minas Gerais
```

Se for passada uma coluna que não existe no dicionário ela será incluída com todos os valores **NaN** (veja o exemplo da coluna 'Renda' abaixo):

[23]:
```
df3 = DataFrame(data, columns=['UF', 'pop', 'Renda', 'estado'])
print(df3)
```

```
   UF       pop Renda             estado
0  PR  11433957   NaN             Paraná
1  SC   7164788   NaN     Santa Caratina
2  RS  11377239   NaN  Rio Grande do Sul
3  SP  45919049   NaN          São Paulo
4  RJ  17264943   NaN     Rio de Janeiro
5  ES   4018650   NaN    Espirito Santos
6  MG  21168791   NaN       Minas Gerais
```

Podemos também especificar o valor dos indices de linha (*index*):

[24]:
```
df3 = DataFrame(data, columns=['UF', 'pop', 'Renda', 'estado'], index=['um',␣
 ↪'dois', 'três', 'quatro', 'cinco', 'seis', 'sete'])
print(df3)
```

```
        UF       pop Renda             estado
um      PR  11433957   NaN             Paraná
dois    SC   7164788   NaN     Santa Caratina
três    RS  11377239   NaN  Rio Grande do Sul
quatro  SP  45919049   NaN          São Paulo
cinco   RJ  17264943   NaN     Rio de Janeiro
seis    ES   4018650   NaN    Espirito Santos
sete    MG  21168791   NaN       Minas Gerais
```

Podemos selecionar uma coluna do DF como uma **Series**:

[25]:
```
print(df3['UF'])
```

```
um        PR
dois      SC
três      RS
quatro    SP
cinco     RJ
seis      ES
sete      MG
Name: UF, dtype: object
```

Ou de uma forma mais conveniente pelo 'açúcar sintático' abaixo:

[26]:
```
print(df3.UF)
```

```
um        PR
dois      SC
três      RS
quatro    SP
cinco     RJ
seis      ES
sete      MG
Name: UF, dtype: object
```

2.2.1 Modificando colunas por atribuição

Atribuir um único valor atribuirá este valor em todas as linhas da coluna:

[27]:
```
df3['Renda'] = 12000.0
print(df3)
```

```
        UF       pop    Renda             estado
um      PR  11433957  12000.0             Paraná
dois    SC   7164788  12000.0     Santa Caratina
três    RS  11377239  12000.0  Rio Grande do Sul
quatro  SP  45919049  12000.0          São Paulo
cinco   RJ  17264943  12000.0     Rio de Janeiro
seis    ES   4018650  12000.0    Espirito Santos
sete    MG  21168791  12000.0       Minas Gerais
```

Observe que utilizamos o valor **float** 12000.0 ao invés do **int** 12000. Veja o caso com **int** abaixo (nos demais exemplos usaremos **float** para 'Renda'):

[28]:
```
df3['Renda'] = 12000
print(df3)
```

```
        UF       pop  Renda             estado
um      PR  11433957  12000             Paraná
dois    SC   7164788  12000     Santa Caratina
três    RS  11377239  12000  Rio Grande do Sul
quatro  SP  45919049  12000          São Paulo
cinco   RJ  17264943  12000     Rio de Janeiro
seis    ES   4018650  12000    Espirito Santos
sete    MG  21168791  12000       Minas Gerais
```

Também podemos usar **array** do **NumPy** (veremos mais sobre **NumPy** na próxima semana):

[29]:
```
df3['Renda'] = (np.arange(7.) * 1000) + 8000
print(df3)
```

```
        UF       pop    Renda             estado
um      PR  11433957   8000.0             Paraná
dois    SC   7164788   9000.0     Santa Caratina
três    RS  11377239  10000.0  Rio Grande do Sul
quatro  SP  45919049  11000.0          São Paulo
cinco   RJ  17264943  12000.0     Rio de Janeiro
seis    ES   4018650  13000.0    Espirito Santos
sete    MG  21168791  14000.0       Minas Gerais
```

Outra alternativa é usar uma lista:

[30]:
```
lista = [8000.0, 9000.0, 10000.0, 11000.0, 12000.0, 13000.0, 14000.0]
df3['Renda'] = lista
print(df3)
```

```
        UF       pop    Renda             estado
um      PR  11433957   8000.0             Paraná
dois    SC   7164788   9000.0     Santa Caratina
três    RS  11377239  10000.0  Rio Grande do Sul
quatro  SP  45919049  11000.0          São Paulo
cinco   RJ  17264943  12000.0     Rio de Janeiro
seis    ES   4018650  13000.0    Espirito Santos
sete    MG  21168791  14000.0       Minas Gerais
```

Ao utilizarmos uma lista ou **array** do **NumPy** o tamanho da lista/**array** deve ser o mesmo do *index* do DF, mas podemos utilizar uma **Series**. Neste caso os rótulos dos *index* serão realinhados e aos dados faltantes será atribuído **NaN**:

[31]:
```
renda = Series([9000, 13000, 10000], index=[1, 5, 2])
df3['Renda'] = renda
```

```
print(df3)
```

```
        UF       pop  Renda             estado
um      PR  11433957    NaN             Paraná
dois    SC   7164788    NaN     Santa Caratina
três    RS  11377239    NaN  Rio Grande do Sul
quatro  SP  45919049    NaN          São Paulo
cinco   RJ  17264943    NaN     Rio de Janeiro
seis    ES   4018650    NaN    Espirito Santos
sete    MG  21168791    NaN       Minas Gerais
```

Note que toda a coluna é substituída.

Atribuição a uma coluna não existente criará uma nova coluna:

[32]:
```
capitais = Series(['Porto Alegre', 'Florianópolis', 'Curitiba'], index=['três',␣
 ↪'dois', 'um'])
df3['Capital'] = capitais
print(df3)
```

```
        UF       pop  Renda             estado        Capital
um      PR  11433957    NaN             Paraná       Curitiba
dois    SC   7164788    NaN     Santa Caratina  Florianópolis
três    RS  11377239    NaN  Rio Grande do Sul   Porto Alegre
quatro  SP  45919049    NaN          São Paulo            NaN
cinco   RJ  17264943    NaN     Rio de Janeiro            NaN
seis    ES   4018650    NaN    Espirito Santos            NaN
sete    MG  21168791    NaN       Minas Gerais            NaN
```

Utilizamos **del** para remover uma coluna:

[33]:
```
del df3['Capital']
print(df3)
print()
print("Colunas:", df3.columns)
```

```
        UF       pop  Renda             estado
um      PR  11433957    NaN             Paraná
dois    SC   7164788    NaN     Santa Caratina
três    RS  11377239    NaN  Rio Grande do Sul
quatro  SP  45919049    NaN          São Paulo
cinco   RJ  17264943    NaN     Rio de Janeiro
seis    ES   4018650    NaN    Espirito Santos
sete    MG  21168791    NaN       Minas Gerais

Colunas: Index(['UF', 'pop', 'Renda', 'estado'], dtype='object')
```

2.2.2 Construindo DF com aninhamento de dicionário

Se um dicionário de dicionário for passado para **DataFrame** o Pandas considerá as chaves do dicionário externo como nomes de colunas e dos dicionários internos como *index*. O Pandas fará o alinhamento dos índices:

[34]:
```
data2 = {
    'UF': {'um': 'PR', 'dois': 'SC', 'três': 'RS'},
    'pop': {'um': 11433957, 'três': 11377239, 'quatro': 45919049},
}

df4 = DataFrame(data2)
print(df4)
```

```
          UF         pop
um        PR  11433957.0
dois      SC         NaN
três      RS  11377239.0
quatro   NaN  45919049.0
```

2.2.3 Transposição

A transposição (inversão dos índices de linha (*index*) e colunas (*columns*)) do DF pode ser obtida com **.T**:

[35]:
```
print(df4.T)
```

```
             um dois          três       quatro
UF           PR   SC            RS          NaN
pop  1.1434e+07  NaN  1.13772e+07  4.5919e+07
```

2.3 Objeto Index

Os índices (**Index**) no Pandas são responsáveis por manter os rótulos (*labels*) dos eixos e outros tipos de metadados.

Qualquer tipo de sequencia passada como *index* (e.g. **list**) é convertida em um objeto do tipo **Index**.

O Objeto **Index** é imutável e portanto não pode ser alterado, mas o *index* de um DF ou **Series** ou um *columns* de um DF podem ser substituído por outro objeto **Index**.

[36]:
```
sr5 = Series(range(0, 3), index=['a', 'b', 'c'])
print(sr5.index)
```

```
Index(['a', 'b', 'c'], dtype='object')
```

3 Principais Funções do Pandas

3.1 reindex

Cria um novo objeto com os dados organizados conforme um novo índice passado como argumento:

[37]:
```
sr6 = Series([1.0, 0.5, -1.5, 2.2], index=['d', 'a', 'c', 'b'])
print(sr6)
```

```
d    1.0
a    0.5
c   -1.5
b    2.2
dtype: float64
```

[38]:
```
sr6 = sr6.reindex(['a', 'b', 'c', 'd', 'e'])
print(sr6)
```

```
a    0.5
b    2.2
c   -1.5
d    1.0
e    NaN
dtype: float64
```

Note que a chamada a **reindex** reorganiza os dados de **sr5** em uma nova **Series**, **sr6** conforme um novo índice, [‘a’, ‘b’, ‘c’, ‘d’, ‘e’], passado como argumento.

3.1.1 method

Para dados ordenados pelo índice pode ser desejável preencher lacunas (valores **NaN**) ao usar **reindex**, para tal podemos utilizar o argumento **method** do **reindex** que pode receber os seguintes valores (strings ou NoneType):

- None (valor padrão): Não preencher as lacunas;
- ‘pad’ ou ‘ffill’: Propagar o último valor válido observado;
- ‘backfill’ ou ‘bfill’: Usar o próximo valor observado; e
- ‘nearest’: Usar o valor válido mais próximo.

[39]:
```
sr7 = pd.Series(['a', 'b', 'c'], index=[0, 2, 4])
print(sr7)
sr7 = sr7.reindex(range(6), method='ffill')
print(sr7)
```

```
0    a
2    b
4    c
dtype: object
0    a
1    a
```

```
2    b
3    b
4    c
5    c
dtype: object
```

Para **DataFrames**, **reindex** pode alterar índices de colunas (*columns*), linhas (*index*) ou ambos. Por padrão o **reindex** em um **DataFrame** é aplicado nos índices de linha (*index*).

[40]:
```
array = np.arange(9).reshape((3, 3))
print(array)
```

```
[[0 1 2]
 [3 4 5]
 [6 7 8]]
```

[41]:
```
df5 = DataFrame(array,
                index=['a', 'b', 'c'],
                columns=['c1', 'c2', 'c3']
               )
print(df5)
```

```
   c1  c2  c3
a   0   1   2
b   3   4   5
c   6   7   8
```

[42]:
```
df6 = df5.reindex(['a', 'b', 'c', 'd'])
print(df6)
```

```
    c1   c2   c3
a  0.0  1.0  2.0
b  3.0  4.0  5.0
c  6.0  7.0  8.0
d  NaN  NaN  NaN
```

[43]:
```
df7 = df5.reindex(columns=['c0', 'c1', 'c2'])
print(df7)
```

```
   c0  c1  c2
a NaN   0   1
b NaN   3   4
c NaN   6   7
```

reindex retorna por padrão um novo objeto (argumento **copy** é por padrão **True**).

3.2 drop

Usamos **drop** para retornar um novo objeto (**Series** ou **DataFrame**) sem o(s) valor(es) deletado(s) de um eixo:

[44]:
```
sr8 = pd.Series(np.arange(5.), index=['a', 'b', 'c', 'd', 'e'])
print(sr8)

sr9 = sr8.drop('c')
print(sr9)
```

```
a    0.0
b    1.0
c    2.0
d    3.0
e    4.0
dtype: float64
a    0.0
b    1.0
d    3.0
e    4.0
dtype: float64
```

[45]:
```
sr10 = sr8.drop(['a', 'e'])
print(sr10)
```

```
b    1.0
c    2.0
d    3.0
dtype: float64
```

Para **DataFrames**, **drop** pode remover com base nos índices de colunas (*columns*), linhas (*index*) ou ambos:

[46]:
```
df8 = DataFrame(np.arange(16).reshape((4, 4)),
                  index=['a', 'b', 'c', 'd'],
                  columns=['c1', 'c2', 'c3', 'c4']
                 )

print(df8)
```

```
   c1  c2  c3  c4
a   0   1   2   3
b   4   5   6   7
c   8   9  10  11
d  12  13  14  15
```

[47]:
```
print(df8.drop(['a', 'c']))  # por padrão o eixo removido é o das linhas␣
↪(axis=0) i.e. index
```

```
   c1  c2  c3  c4
b   4   5   6   7
d  12  13  14  15
```

Equivalentemente podemos fazer:

[48]:
```
print(df8.drop(index=['a', 'c']))
```

```
   c1  c2  c3  c4
b   4   5   6   7
d  12  13  14  15
```

Para removermos colunas usamos **columns** OU **axis=1**:

[49]:
```
print(df8.drop(columns=['c1', 'c2']))
```

```
   c3  c4
a   2   3
b   6   7
c  10  11
d  14  15
```

Ao invés de especificar **columns**, podemos trocar o eixo padrão de **axis=0** (*index* i.e. linhas) para **axis=1** (*columns*). Ou seja o código abaixo é equivalente ao acima:

[50]:
```
print(df8.drop(['c1', 'c2'], axis=1))  # axis = 1 faz com que seja removida a␣
↪coluna
```

```
   c3  c4
a   2   3
b   6   7
c  10  11
d  14  15
```

[51]:
```
print(df8.drop(columns=['c1', 'c2'], index=['a', 'b']))
```

```
   c3  c4
c  10  11
d  14  15
```

3.3 Indexando Series

Podemos indexar itens com [], observe:

[52]:
```
sr11 = pd.Series(np.arange(4.), index=['a', 'b', 'c', 'd'])
print(sr11)
print()  # pulo de linha
print(sr11['a'])
print(sr11['c'])
```

```
a    0.0
b    1.0
c    2.0
d    3.0
```

```
dtype: float64

0.0
2.0
```

Mesmo com índices do formato **string**, podemos indexar também pela ordem, usando inteiros:

[53]:
```
print(sr11[0])
print(sr11[1])
```

```
0.0
1.0
```

3.4 Indexando DataFrames

Ao indexar um **DataFrame**, indexamos a sua coluna:

[54]:
```
print(df8)
```

```
   c1  c2  c3  c4
a   0   1   2   3
b   4   5   6   7
c   8   9  10  11
d  12  13  14  15
```

[55]:
```
print(df8['c2'])
```

```
a     1
b     5
c     9
d    13
Name: c2, dtype: int64
```

Podemos passar como índice uma lista (**list**) de colunas a serem selecionadas. Podemos inclusive definir uma ordem diferente:

[56]:
```
colunas = ['c2', 'c1', 'c4']
print(df8[colunas])
```

```
   c2  c1  c4
a   1   0   3
b   5   4   7
c   9   8  11
d  13  12  15
```

Clara podemos usar a lista direto dentro do operador de indexação/seleção [[]]. Note a sintaxe:

- Os colchetes **externos** se referem ao operador de indexação/seleção.
- Os colchetes **internos** são os construtores da lista (**list**):

[57]:
```
print(df8[['c1', 'c2']])
```

```
   c1  c2
a   0   1
b   4   5
c   8   9
d  12  13
```

3.5 Slice de Series

O **slice** funciona normalmente, usamos [**inicio:fim**]. Porém além dos valores inteiros podemos usar os índices. Cuidado, com índices o valor de fim é inclusivo, diferente do que normalmente ocorre com listas em Python.

[58]:
```
print(sr11[0:2])
```

```
a    0.0
b    1.0
dtype: float64
```

[59]:
```
print(sr11['a':'c'])
```

```
a    0.0
b    1.0
c    2.0
dtype: float64
```

3.6 Slice de DataFrame

Nos **DataFrame**s o slice é aplicado aos índices de linha:

[60]:
```
print(df8)
```

```
   c1  c2  c3  c4
a   0   1   2   3
b   4   5   6   7
c   8   9  10  11
d  12  13  14  15
```

[61]:
```
print(df8[0:2])
```

```
   c1  c2  c3  c4
a   0   1   2   3
b   4   5   6   7
```

[62]:
```
print(df8[1:3])
```

```
   c1  c2  c3  c4
b   4   5   6   7
c   8   9  10  11
```

[63]:
```
print(df8[:2])  # nenhum valor antes dos : significa desde o início, o mesmo
 ↪que 0
```

```
   c1  c2  c3  c4
a   0   1   2   3
b   4   5   6   7
```

[64]:
```
print(df8[1:])  # nenhum valor após os : significa até o fim, inclusive
```

```
   c1  c2  c3  c4
b   4   5   6   7
c   8   9  10  11
d  12  13  14  15
```

3.7 Filtros

Além de podemos passar um índice, lista de índices ou slices nos seletores [] podemos passar uma série booleana que fará a seleção. Observe:

[65]:
```
print(sr11)
```

```
a    0.0
b    1.0
c    2.0
d    3.0
dtype: float64
```

[66]:
```
filtro = sr11 < 1.5  # compara cada valor da Series
print(filtro)
```

```
a     True
b     True
c    False
d    False
dtype: bool
```

[67]:
```
print(sr11[filtro])
```

```
a    0.0
b    1.0
dtype: float64
```

Também podemos facilmente condensar os códigos acima de maneira equivalente:

[68]:
```
print(sr11[sr11 < 1.5])
```

```
a    0.0
b    1.0
dtype: float64
```

O mesmo pode ser aplicado a um **DataFrame**.

[69]:
```
print(df8)
```

```
    c1  c2  c3  c4
a    0   1   2   3
b    4   5   6   7
c    8   9  10  11
d   12  13  14  15
```

Primeiramente obtemos uma **Series** de booleanos com base em uma das colunas do DataFrame:

[70]:
```
filtro = df8['c1'] <= 8
print(filtro)
```

```
a     True
b     True
c     True
d    False
Name: c1, dtype: bool
```

Então aplicamos o filtro no **DataFrame**:

[71]:
```
print(df8[filtro])
```

```
    c1  c2  c3  c4
a    0   1   2   3
b    4   5   6   7
c    8   9  10  11
```

Condensando tudo em uma única linha:

[72]:
```
print(df8[df8["c1"] <= 8])
```

```
    c1  c2  c3  c4
a    0   1   2   3
b    4   5   6   7
c    8   9  10  11
```

Podemos fazer uma expressão mais complexa com o filtro envolvendo várias colunas:

[73]:
```
print(df8[(df8["c1"] == 4) | (df8["c3"] == 10)])
```

```
    c1  c2  c3  c4
b    4   5   6   7
c    8   9  10  11
```

Nota: no exemplo acima é importante colocar cada condição entre parêntesis!

3.8 Seleção com loc e iloc

loc e **iloc** são operadores especiais para a seleção que permitem selecionar um subconjunto de linhas e colunas de um **DataFrame** usando tanto rótulos (**loc**) ou inteiros (**iloc**).

Vamos considerar o objeto abaixo (já definido previamente):

[74]:
```
print(df8)
```

```
   c1  c2  c3  c4
a   0   1   2   3
b   4   5   6   7
c   8   9  10  11
d  12  13  14  15
```

Vamos selecionar uma única linha e duas colunas

[75]:
```
print(df8.loc['a', ['c1', 'c3']])
```

```
c1    0
c3    2
Name: a, dtype: int64
```

Vamos fazer a mesma seleção mostrada acima, mas utilizando índices inteiros com **iloc**:

[76]:
```
print(df8.iloc[0, [0, 2]])
```

```
c1    0
c3    2
Name: a, dtype: int64
```

Vamos selecionar as linhas 'a' e 'd' com as colunas 'c1' e 'c3':

[77]:
```
print(df8.loc[['a', 'd'], ['c1', 'c3']])
```

```
   c1  c3
a   0   2
d  12  14
```

Selecionar todas as colunas de uma linha:

[78]:
```
print(df8.loc['a'])
```

```
c1    0
c2    1
c3    2
c4    3
Name: a, dtype: int64
```

Usando **slice**:

[79]:
```
print(df8.loc[:'c', 'c1'])
```

```
a    0
b    4
c    8
Name: c1, dtype: int64
```

[80]:
```
print(df8.iloc[:, :3])
```

```
   c1  c2  c3
a   0   1   2
b   4   5   6
c   8   9  10
d  12  13  14
```

Usando filtros:

[81]:
```
print(df8.loc[df8.c1 <= 7])
```

```
   c1  c2  c3  c4
a   0   1   2   3
b   4   5   6   7
```

4 Operações Aritméticas

Podemos realizar várias operações aritméticas envolvendo **Series** e **DataFrames**. Nestes casos o Pandas cuida de fazer automaticamente o alinhamento, isto é, as operações são aplicadas entre os valores dos dois objetos envolvidos que estão no mesmo índice.

Vamos considerar as duas **Series** abaixo definidas:

[82]:
```
print("sr12:")
sr12 = Series([1, 3, 5, 7], index=['a', 'b', 'c', 'd'])
print(sr12)

print()  # pulo de linha

print("sr13:")
sr13 = Series([40, 30, 20, 10], index=['d', 'c', 'b', 'a'])
print(sr13)
```

```
sr12:
a    1
b    3
c    5
d    7
dtype: int64

sr13:
d    40
c    30
```

```
b    20
a    10
dtype: int64
```

E realizar uma operação de soma:

[83]:
```
resultado = sr12 + sr13
print(resultado)
```

```
a    11
b    23
c    35
d    47
dtype: int64
```

Observe que os valores das **Series sr12** e **sr13** foram alinhados conforme o valor do rótulo (*index*), isto é 'a', 'b', 'c' e 'd'; eles não foram alinhados por sua ordem. Assim, por exemplo, o valor de **resultado['a']** é definido por ***1 + 10 = 11**.

Considere o **sr14** definido abaixo que contém índices diferentes aos dos dois abjetos definidos anteriormente:

[84]:
```
sr14 = Series([100, 200, 300, 400], index = ['z', 'b', 'c', 'y'])
```

Observe o que acontece quando não há um índice correspondente à alguns valores durante uma operação:

[85]:
```
print(sr12 + sr14)
```

```
a      NaN
b    203.0
c    305.0
d      NaN
y      NaN
z      NaN
dtype: float64
```

Apenas índices que contém valores definidos nos dois objetos geram algum resultado de soma, os demais índices ficam com **NaN**. Observe que o objeto resultantes:

- Tem como índice todos índices dos dois objetos operados, isto é, é feita a união dos dois conjunto de índices
- Apenas índices da intersecção do conjunto de índices são valores definidos, os demais recebem **NaN**.

Em muitas aplicações, especialmente em ciência de dados, pode ser desejado tratar esses valores faltantes, por exemplo, completar os valores faltantes com um valor padrão, para isso aplicamos a operação utilizando métodos (e.g., **add**) ao ínvés do operador (e.g. +) e passamos o argumento **fill_value**:

[86]:
```
resultado = sr12.add(sr14, fill_value=0)
print(resultado)
```

```
a      1.0
b    203.0
c    305.0
d      7.0
y    400.0
z    100.0
dtype: float64
```

Observe no exemplo acima que os valores faltantes de **sr12** e **sr14** são preenchidos com o valor informado pelo argumento **fill_value**, neste caso 0.

Veja abaixo uma lista das operações e respectivo método. Cada método tem uma contraparte, iniciando com a letra **r**, que inverte a ordem dos argumentos, então ambos os casos do exemplo abaixo são equivalentes:

[87]:
```
print(1 / sr12)
print(sr12.rdiv(1))
```

```
a    1.000000
b    0.333333
c    0.200000
d    0.142857
dtype: float64
a    1.000000
b    0.333333
c    0.200000
d    0.142857
dtype: float64
```

Métodos aritméticos:

- add, radd: + (soma)
- sub, rsub: - (subtração)
- div, rdiv: / (divisão inteira, se nenhum valor **float** estiver envolvido)
- floordiv, rfloordiv: // (divisão de **float**)
- mul, rmul: * (multiplicação)
- pow, rpow: ** (exponenciação)

4.1 Operações aritméticas em DataFrames

As operações aritméticas também podem ser realizadas em **DataFrames**, neste caso o alinhamento considera os índices das linhas e das colunas.

Considere os **DataFrames** definidos abaixo:

[88]:
```
df9 = DataFrame(np.arange(16.0).reshape((4, 4)), index=['l1', 'l2', 'l3',␣
 ↪'l4'], columns=['c1', 'c2', 'c3', 'c4'])
print(df9)
```

```
      c1    c2    c3    c4
l1   0.0   1.0   2.0   3.0
```

```
l2   4.0   5.0   6.0   7.0
l3   8.0   9.0  10.0  11.0
l4  12.0  13.0  14.0  15.0
```

[89]:
```
df10 = DataFrame(np.arange(16.0).reshape((4, 4)) * 100.0, index=['l1', 'l2',␣
 ↪'l3', 'l4'], columns=['c1', 'c2', 'c3', 'c4'])
print(df10)
```

```
        c1      c2      c3      c4
l1     0.0   100.0   200.0   300.0
l2   400.0   500.0   600.0   700.0
l3   800.0   900.0  1000.0  1100.0
l4  1200.0  1300.0  1400.0  1500.0
```

Observe o uso do operador de soma abaixo:

[90]:
```
print(df9 + df10)
```

```
        c1      c2      c3      c4
l1     0.0   101.0   202.0   303.0
l2   404.0   505.0   606.0   707.0
l3   808.0   909.0  1010.0  1111.0
l4  1212.0  1313.0  1414.0  1515.0
```

Vamos agora construir outro **DataFrame**, **d11**, com os mesmos valores de **df10**, mas com alguns índices diferentes para analisarmos o alinhamento:

[91]:
```
df11 = DataFrame(np.arange(16.0).reshape((4, 4)) * 100.0, index=['l0', 'l1',␣
 ↪'l2', 'l5'], columns=['c1', 'c2', 'c3', 'c5'])
print(df11)
```

```
        c1      c2      c3      c5
l0     0.0   100.0   200.0   300.0
l1   400.0   500.0   600.0   700.0
l2   800.0   900.0  1000.0  1100.0
l5  1200.0  1300.0  1400.0  1500.0
```

Observe, que da mesma forma que ocorre com as operações com **Series**, em operações com **DataFrames** nos quais nem todos os índices casam o resultado contém valores **NaN**:

[92]:
```
print(df9 + df11)
```

```
       c1     c2      c3  c4  c5
l0    NaN    NaN     NaN NaN NaN
l1  400.0  501.0   602.0 NaN NaN
l2  804.0  905.0  1006.0 NaN NaN
l3    NaN    NaN     NaN NaN NaN
l4    NaN    NaN     NaN NaN NaN
l5    NaN    NaN     NaN NaN NaN
```

E igualmente podemos usar **fill_value**:

[93]:
```
print(df9.add(df11, fill_value=0))
```

```
        c1      c2      c3    c4      c5
l0     0.0   100.0   200.0   NaN   300.0
l1   400.0   501.0   602.0   3.0   700.0
l2   804.0   905.0  1006.0   7.0  1100.0
l3     8.0     9.0    10.0  11.0     NaN
l4    12.0    13.0    14.0  15.0     NaN
l5  1200.0  1300.0  1400.0   NaN  1500.0
```

4.2 Operações entre DataFrame e Series

Por padrão operações envolvendo **DataFrame** e **Series** faz o casamento dos **índices** da **Series** com as **colunas** do **DataFrame**, aplicando a mesma operação em cada linha do **DataFrame**, o que é conhecido como **Broadcast**. Em resumo é como se cada linha do **DataFrame** fosse considerado um **Series** e feita várias operações de **Series** com **Series**. Veja o exemplo abaixo:

[94]:
```
sr15 = Series([0., 1., 2., 3.], index=['c1', 'c2', 'c3', 'c4'])
print(sr15)
```

```
c1    0.0
c2    1.0
c3    2.0
c4    3.0
dtype: float64
```

[95]:
```
print(df9 - sr15)
```

```
      c1    c2    c3    c4
l1   0.0   0.0   0.0   0.0
l2   4.0   4.0   4.0   4.0
l3   8.0   8.0   8.0   8.0
l4  12.0  12.0  12.0  12.0
```

No exemplo acima, para cada linha do **DataFrame**, **df9**, foi mantido a primeira coluna e então removido 1.0, 2.0 e 3.0 das colunas 'c2', 'c3' e 'c4', respectivamente

Para fazer o **Broadcast** com o casamento sobre as linhas do **DataFrame** usamos o operador como um método e definimos **axis='index'**:

[96]:
```
sr16 = Series([0., 1., 2., 3.], index=['l1', 'l2', 'l3', 'l4'])
print(sr16)
```

```
l1    0.0
l2    1.0
l3    2.0
l4    3.0
dtype: float64
```

[97]:
```
print(df9.sub(sr16, axis='index'))
```

```
     c1    c2    c3    c4
l1  0.0   1.0   2.0   3.0
l2  3.0   4.0   5.0   6.0
l3  6.0   7.0   8.0   9.0
l4  9.0  10.0  11.0  12.0
```

5 Aplicando funções

Algo muito comum é aplicar uma função para cada linha ou coluna do **DataFrame**:

[98]:
```
df12 = DataFrame([[1, 3, 5, 7], [-3, 6, 3, 9], [0, 4, -7, 3], [5, 8, 3, 4]],␣
 ↪index=['l1', 'l2', 'l3', 'l4'], columns=['c1', 'c2', 'c3', 'c4'])
print(df12)
fn = lambda x: x.max() - x.min()
print(df12.apply(fn))
```

```
    c1  c2  c3  c4
l1   1   3   5   7
l2  -3   6   3   9
l3   0   4  -7   3
l4   5   8   3   4
c1     8
c2     5
c3    12
c4     6
dtype: int64
```

O função acima calcula a diferença entre o maior e o menor valor de cada coluna do **DataFrame**. A função **fn**, no caso acima, é chamada 4 vezes, em cada vez **x** são todos os valores da coluna.

max e **min** são funções estatísticas, e há um conjunto destas funções disponíveis no Pandas.

6 Funções estatísticas (sumário)

A melhor forma de se iniciar uma análise com Pandas é usar **describe**. **describe** relata uma série de informações estatísticas, veja:

[99]:
```
print(df12.describe())
```

```
             c1        c2        c3        c4
count  4.000000  4.000000  4.000000  4.000000
mean   0.750000  5.250000  1.000000  5.750000
std    3.304038  2.217356  5.416026  2.753785
min   -3.000000  3.000000 -7.000000  3.000000
25%   -0.750000  3.750000  0.500000  3.750000
50%    0.500000  5.000000  3.000000  5.500000
```

```
75%    2.000000  6.500000  3.500000  7.500000
max    5.000000  8.000000  5.000000  9.000000
```

Para cada coluna, **describe** retorna:

- count: Número de itens
- mean: Média
- std: Desvio Padrão
- min: menor valor
- max: maior valor

Ainda, **describe** retorna alguns Percentis. Você pode definir quais deles deseja pelo argumento **percentiles**:

[100]:
```
print(df12.describe(percentiles=[.33, .67]))
```

```
             c1        c2        c3        c4
count  4.000000  4.000000  4.000000  4.000000
mean   0.750000  5.250000  1.000000  5.750000
std    3.304038  2.217356  5.416026  2.753785
min   -3.000000  3.000000 -7.000000  3.000000
33%   -0.030000  3.990000  2.900000  3.990000
50%    0.500000  5.000000  3.000000  5.500000
67%    1.040000  6.020000  3.020000  7.020000
max    5.000000  8.000000  5.000000  9.000000
```

[101]:
```
print(df12.describe(percentiles=[.33, .67, .70, .75, .99]))
```

```
             c1        c2        c3        c4
count  4.000000  4.000000  4.000000  4.000000
mean   0.750000  5.250000  1.000000  5.750000
std    3.304038  2.217356  5.416026  2.753785
min   -3.000000  3.000000 -7.000000  3.000000
33%   -0.030000  3.990000  2.900000  3.990000
50%    0.500000  5.000000  3.000000  5.500000
67%    1.040000  6.020000  3.020000  7.020000
70%    1.400000  6.200000  3.200000  7.200000
75%    2.000000  6.500000  3.500000  7.500000
99%    4.880000  7.940000  4.940000  8.940000
max    5.000000  8.000000  5.000000  9.000000
```

Note que 50% sempre é incluso.

Podemos solicitar diretamente cada uma estatísticas desejadas:

[102]:
```
print(df12.max())
```

```
c1    5
c2    8
c3    5
```

```
c4    9
dtype: int64
```

[103]:
```
print(df12.min())
```

```
c1   -3
c2    3
c3   -7
c4    3
dtype: int64
```

[104]:
```
print(df12.std())
```

```
c1    3.304038
c2    2.217356
c3    5.416026
c4    2.753785
dtype: float64
```

[105]:
```
print(df12.var())  # Variância
```

```
c1    10.916667
c2     4.916667
c3    29.333333
c4     7.583333
dtype: float64
```

[106]:
```
print(df12.mean())
```

```
c1    0.75
c2    5.25
c3    1.00
c4    5.75
dtype: float64
```

[107]:
```
print(df12.count())
```

```
c1    4
c2    4
c3    4
c4    4
dtype: int64
```

Algumas outras estatísticas também estão disponíveis:

[108]:
```
print(df12.sum())  # retorna a somatória de todos os valores de cada coluna
```

```
c1     3
c2    21
```

```
c3     4
c4    23
dtype: int64
```

[109]:
```
print(df12.cumsum())  # Somatória acumulativa
```

```
    c1  c2  c3  c4
l1   1   3   5   7
l2  -2   9   8  16
l3  -2  13   1  19
l4   3  21   4  23
```

Note que os valores na última linha de **cumsum** é o retornado por **sum**.

[110]:
```
print(df12.median())  # Mediana
```

```
c1    0.5
c2    5.0
c3    3.0
c4    5.5
dtype: float64
```

[111]:
```
print(df12.mad())  # Desvio médio absoluto da média
```

```
c1    2.25
c2    1.75
c3    4.00
c4    2.25
dtype: float64
```

[112]:
```
print(df12.prod())  # Produtório dos valores de cada coluna (multiplicação de␣
↪todos os valores)
```

```
c1      0
c2    576
c3   -315
c4    756
dtype: int64
```

6.1 Modificando eixo:

Podemos sumarizar sobre os valores de cada linha ao invés das colunas. Para isso passamos o argumento **axis='columns'**, observe:

[113]:
```
print(df12.sum(axis='columns'))
```

```
l1    16
l2    15
l3     0
```

```
14    20
dtype: int64
```

7 Ordenação

A ordenação retorna um novo objeto ordenado. Podemos ordenar pelos índices de linha e no caso de **DataFrames** também os índices de coluna.

7.1 Ordenando índices

Para ordenar **DataFrames** e **Series** lexicograficamente pelo seu índice (de linha ou coluna) usamos **sort_index**:

7.1.1 Ordenando Series

[114]:
```
sr16 = Series([0., 1., 2., 3.], index=['c3', 'c2', 'c4', 'c1'])
print(sr16)
```

```
c3    0.0
c2    1.0
c4    2.0
c1    3.0
dtype: float64
```

[115]:
```
print(sr16.sort_index())  # Ordenar a Series sr16 pelo seus índices
```

```
c1    3.0
c2    1.0
c3    0.0
c4    2.0
dtype: float64
```

7.1.2 Ordenando DataFrames

Podemos ordenar **DataFrames** lexicograficamente pelos índices da linha ou coluna. O padrão é ordenar pelos índices de linha.

[116]:
```
df13 = DataFrame(np.arange(16.).reshape((4, 4)), index=['c', 'da', 'db', 'a'],␣
 ↪columns=['cc', 'ca', 'cd', 'cb'])
print(df13)
```

```
      cc    ca    cd    cb
c    0.0   1.0   2.0   3.0
da   4.0   5.0   6.0   7.0
db   8.0   9.0  10.0  11.0
a   12.0  13.0  14.0  15.0
```

[117]:
```
print(df13.sort_index())  # Ordena pelos índices de linha (padrão)
```

```
      cc    ca    cd    cb
a   12.0  13.0  14.0  15.0
c    0.0   1.0   2.0   3.0
da   4.0   5.0   6.0   7.0
db   8.0   9.0  10.0  11.0
```

Usamos **axis='columns'** para ordenar pelos índices de coluna:

[118]:
```
print(df13.sort_index(axis='columns'))  # Ordena pelas colunas
```

```
      ca    cb    cc    cd
c    1.0   3.0   0.0   2.0
da   5.0   7.0   4.0   6.0
db   9.0  11.0   8.0  10.0
a   13.0  15.0  12.0  14.0
```

7.2 Ordem ascendente ou descendente

Por padrão a ordenação é ascendente, isto é, em ordem crescente (e.g. 1, 2, 3, 4). Mas podemos modificar com o argumento **ascending=False**:

[119]:
```
print(df13.sort_index(ascending=False))
```

```
      cc    ca    cd    cb
db   8.0   9.0  10.0  11.0
da   4.0   5.0   6.0   7.0
c    0.0   1.0   2.0   3.0
a   12.0  13.0  14.0  15.0
```

7.3 Ordenando pelos valores

Para ordenar pelos valores utilizamos o método **sort_values**

7.3.1 Series

[120]:
```
sr17 = pd.Series([3, 6, -4, 1])
print(sr17)
```

```
0    3
1    6
2   -4
3    1
dtype: int64
```

[121]:
```
print(sr17.sort_values())
```

```
2   -4
3    1
0    3
```

```
1    6
dtype: int64
```

7.3.2 DataFrames

Quando ordenamos **DataFrames** podemos usar os valores de uma ou mais colunas (ou linhas se alterarmos o eixo), definimos tais valores pelo método **by**. **by** pode receber o nome de um índice ou uma lista com o nome dos índices na ordem de precedência (por padrão nome de colunas). O valor do segundo índice é usado para "desempatar" quando os valores no primeiro índice são iguais.

[122]:
```
df14 = DataFrame([[6, 9], [5, 7], [6, 8], [4,10], [0,100], [6, 2], [5, 9],␣
↪[5,10], [0, 12]], columns=['a', 'b'])
print(df14)
```

```
   a    b
0  6    9
1  5    7
2  6    8
3  4   10
4  0  100
5  6    2
6  5    9
7  5   10
8  0   12
```

[123]:
```
print(df14.sort_values(by='a'))
```

```
   a    b
4  0  100
8  0   12
3  4   10
1  5    7
6  5    9
7  5   10
0  6    9
2  6    8
5  6    2
```

[124]:
```
print(df14.sort_values(by=['a', 'b']))
```

```
   a    b
8  0   12
4  0  100
3  4   10
1  5    7
6  5    9
7  5   10
5  6    2
```

```
2  6    8
0  6    9
```

[125]:
```
print(df14.sort_values(by=['b', 'a']))
```

```
   a    b
5  6    2
1  5    7
2  6    8
6  5    9
0  6    9
3  4   10
7  5   10
8  0   12
4  0  100
```

Anteriormente selecionamos uma ou mais colunas para ordenar as linhas, ou seja, analisamos cada linha como um objeto/registro e comparamos com base em uma (ou mais) coluna(s) específica(s).

Podemos usar **axis='columns'** para ordenar as colunas com base nos seus valores em uma (ou mais) linha(s) específica(s)

[126]:
```
print(df12)
print()  # Pulo de linha
print(df12.sort_values(by='l2', axis='columns'))
```

```
    c1  c2  c3  c4
l1   1   3   5   7
l2  -3   6   3   9
l3   0   4  -7   3
l4   5   8   3   4

    c1  c3  c2  c4
l1   1   5   3   7
l2  -3   3   6   9
l3   0  -7   4   3
l4   5   3   8   4
```

8 Bibliografia

IDRIS, I. Python Data Analysis: Learn how to apply powerful data analysis techniques with popular open source Python modules. UK: Packt Publishing Ltd, 2014.

MCKINNEY, W. Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython. Segunda edição, O'Reilly, 2017.

GUTTAG, J. Introduction to Computation and Programming Using Python: With Application to Understanding Data Second Edition. USA: MIT Press, 2016.