

Introdução à Python

Altino Dantas

PARTE II

Objetivos

- Explorar as principais funcionalidades **Pandas**;
- Aprender a lidar com índices, funções, ordenações;
- Gerar gráficos simples diretamente de objetos **Pandas**;



Reindexando

- Um método crítico em objetos pandas é reindex, o que significa criar um novo objeto com os dados em conformidade com um novo índice. Considere um exemplo simples de cima:

```
obj = Series([4.5, 7.2, -5.3, 3.6],  
             index=['d', 'b', 'a', 'c'])  
obj
```

Out[110]:

```
d    4.5  
b    7.2  
a   -5.3  
c    3.6  
dtype: float64
```

In [113]:

```
obj2 = obj.reindex(['a', 'b', 'c', 'd', 'e'], fill_value=0)  
obj2
```

Out[113]:

```
a   -5.3  
b    7.2  
c    3.6  
d    4.5  
e    0.0  
dtype: float64
```





Reindexando

- Para dados ordenados, como séries temporais, pode ser desejável fazer alguma interpolação ou preenchimento de valores ao reindexar.
- O parâmetro **method** nos permite fazer isso, usando um método como **ffill**, que preenche os valores:

```
obj3 = Series(['blue', 'purple', 'yellow'],  
              index=[0, 2, 4])  
obj3.reindex(range(6), method='ffill')
```

Out[119]:

```
0      blue  
1      blue  
2    purple  
3    purple  
4    yellow  
5    yellow  
dtype: object
```

```
obj3 = Series(['blue', 'purple', 'yellow'],  
              index=[0, 2, 4])  
obj3.reindex(range(6), method='bfill')
```

Out[120]:

```
0      blue  
1    purple  
2    purple  
3    yellow  
4    yellow  
5         NaN  
dtype: object
```

Reindexando

- Com o DataFrame, o **reindex** pode alterar o índice (linha), as colunas ou ambos.
- Quando passada apenas uma sequência, as linhas são reindexadas:

```
frame = DataFrame(np.arange(9).reshape((3, 3)),  
                  index=['a', 'c', 'd'],  
                  columns=['Ohio', 'Texas', 'California'])  
frame
```

	Ohio	Texas	California
a	0	1	2
c	3	4	5
d	6	7	8

```
frame2 = frame.reindex(['a', 'b', 'c', 'd'])  
frame2
```

	Ohio	Texas	California
a	0.0	1.0	2.0
b	NaN	NaN	NaN
c	3.0	4.0	5.0
d	6.0	7.0	8.0



Reindexando

- As colunas podem ser reindexadas usando a palavra-chave **columns**;
- Ambos podem ser reindexados de uma só vez, embora a interpolação se aplique apenas nas linhas (eixo 0):

Out[1]:

	Ohio	Texas	California
a	0	1	2
c	3	4	5
d	6	7	8

In [2]:

```
states = ['Texas', 'Utah', 'California']  
frame.reindex(columns=states)
```

Out[2]:

	Texas	Utah	California
a	1	NaN	2
c	4	NaN	5
d	7	NaN	8





Descartando dados de um eixo

- Eliminar uma ou mais entradas de um eixo é fácil se você tiver um array ou lista de índices sem essas entradas.
- Como isso pode ser um trabalhoso, o método **drop** retornará um novo objeto com o valor indicado ou valores excluídos de um eixo.

```
In [39]: obj = Series(np.arange(5.),  
                    index=['a', 'b', 'c', 'd', 'e'])
```

```
In [40]: obj
```

```
Out[40]: a    0.0  
        b    1.0  
        c    2.0  
        d    3.0  
        e    4.0  
        dtype: float64
```

```
In [41]: new_obj = obj.drop('c')
```

```
In [42]: new_obj
```

```
Out[42]: a    0.0  
        b    1.0  
        d    3.0  
        e    4.0  
        dtype: float64
```

Descartando dados de um eixo

- Com o **DataFrame**, os dados indexados podem ser excluídos de qualquer eixo;

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

```
data.drop(['Colorado', 'Ohio'])
```

	one	two	three	four
Utah	8	9	10	11
New York	12	13	14	15

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

```
data.drop('two',axis=1)
```

	one	three	four
Ohio	0	2	3
Colorado	4	6	7
Utah	8	10	11
New York	12	14	15



Indexação, seleção e filtragem

A indexação em **Serie** funciona de maneira análoga à indexação de *arrays* **NumPy**, exceto pelo fato de você poder usar os valores de índice da **Serie** em vez de apenas inteiros.

```
obj = Series(np.arange(4.), index=['a', 'b', 'c', 'd'])  
obj
```

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

```
In [103]: obj['b']  
Out[103]: 1.0
```

```
In [104]: obj[1]  
Out[104]: 1.0
```

```
In [105]: obj[2:4]  
Out[105]:
```

```
c    2  
d    3
```

```
In [106]: obj[['b', 'a', 'd']]  
Out[106]:
```

```
b    1  
a    0  
d    3
```

```
In [107]: obj[[1, 3]]  
Out[107]:
```

```
b    1  
d    3
```

```
In [108]: obj[obj < 2]  
Out[108]:
```

```
a    0  
b    1
```



Indexação, seleção e filtragem

Indexação em um **DataFrame** é para recuperar uma ou mais colunas com um único valor ou sequência:

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

```
data['two']
```

```
Ohio      1
Colorado  5
Utah       9
New York  13
Name: two, dtype: int32
```

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

```
data[['three', 'one']]
```

	three	one
Ohio	2	0
Colorado	6	4
Utah	10	8
New York	14	12





Indexação, seleção e filtragem

- A indexação como essa tem alguns casos especiais. Primeiro, selecionando linhas por fatiamento ou uma **array** booleano:

```
In [116]: data[:2]
```

```
Out[116]:
```

Ohio	0	1	2	3
Colorado	4	5	6	7

```
In [117]: data[data['three'] > 5]
```

```
Out[117]:
```

Colorado	4	5	6	7
Utah	8	9	10	11
New York	12	13	14	15



Indexação, seleção e filtragem

- Outro caso de uso está na indexação com um **DataFrame** booleano, como um produzido por uma comparação escalar:

```
data < 5
```

	one	two	three	four
Ohio	True	True	True	True
Colorado	True	False	False	False
Utah	False	False	False	False
New York	False	False	False	False

```
data[data < 5] = 0  
data
```

	one	two	three	four
Ohio	0	0	0	0
Colorado	0	5	6	7
Utah	8	9	10	11
New York	12	13	14	15



Indexação, seleção e filtragem

- É possível selecionar um subconjunto das linhas e colunas de um **DataFrame** com uma notação similar ao NumPy (**loc** e **iloc**):

	one	two	three	four
Ohio	0	0	0	0
Colorado	0	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

```
data.loc['Colorado', ['two', 'three']]
```

```
two      5  
three    6  
Name: Colorado, dtype: int32
```

```
data.iloc[[1], [3, 0, 1]]
```

	four	one	two
Colorado	7	0	5

Operações Aritméticas

- Em operações aritméticas entre objetos indexados de maneira diferente, você pode querer preencher com um valor especial, como 0, quando um rótulo de eixo é encontrado em um objeto, mas não em outro:

```
df1 = DataFrame(np.arange(12.).reshape((3, 4)),  
                columns=list('abcd'))  
df1
```

	a	b	c	d
0	0.0	1.0	2.0	3.0
1	4.0	5.0	6.0	7.0
2	8.0	9.0	10.0	11.0

```
df2 = DataFrame(np.arange(20.).reshape((4, 5)),  
                columns=list('abcde'))  
df2
```

	a	b	c	d	e
0	0.0	1.0	2.0	3.0	4.0
1	5.0	6.0	7.0	8.0	9.0
2	10.0	11.0	12.0	13.0	14.0
3	15.0	16.0	17.0	18.0	19.0



Operações Aritméticas

- Usando o método **add** no **df1** e passando **df2** e um argumento para **fill_value**:



```
df1 + df2
```

	a	b	c	d	e
0	0.0	2.0	4.0	6.0	NaN
1	9.0	11.0	13.0	15.0	NaN
2	18.0	20.0	22.0	24.0	NaN
3	NaN	NaN	NaN	NaN	NaN

```
df1.add(df2, fill_value=0)
```

	a	b	c	d	e
0	0.0	2.0	4.0	6.0	4.0
1	9.0	11.0	13.0	15.0	9.0
2	18.0	20.0	22.0	24.0	14.0
3	15.0	16.0	17.0	18.0	19.0

Method	Description
add	Method for addition (+)
sub	Method for subtraction (-)
div	Method for division (/)
mul	Method for multiplication (*)





Operações Aritméticas

- Por padrão, a aritmética entre **DataFrame** e **Series** corresponde ao índice da série nas colunas do DataFrame;
- Opera sobre as linhas do DataFrame

DataFrame
frame

	b	d	e
Utah	0.0	1.0	2.0
Ohio	3.0	4.0	5.0
Texas	6.0	7.0	8.0
Oregon	9.0	10.0	11.0

Serie
series

```
b    0.0
d    1.0
e    2.0
Name: Utah, dtype: float64
```

frame - series

	b	d	e
Utah	0.0	0.0	0.0
Ohio	3.0	3.0	3.0
Texas	6.0	6.0	6.0
Oregon	9.0	9.0	9.0

Aplicação de função

- **NumPy *ufuncs*** (funções aplicadas em elementos de arrays) funcionam bem com objetos **pandas**:

	b	d	e
Utah	-1.483807	-0.956676	0.924737
Ohio	-0.432996	-0.520583	1.323618
Texas	0.634368	-0.394914	-0.869185
Oregon	0.635156	-0.598228	-0.533513

```
np.abs(frame)
```

	b	d	e
Utah	1.483807	0.956676	0.924737
Ohio	0.432996	0.520583	1.323618
Texas	0.634368	0.394914	0.869185
Oregon	0.635156	0.598228	0.533513



Aplicação de função

- Outra operação frequente é aplicar uma função em *arrays* 1D a cada coluna ou linha. O método **apply** do **DataFrame** faz exatamente isso:

```
In [140]: f = lambda x: x.max()  
          frame.apply(f)
```

```
Out[140]: b    0.635156  
          d   -0.394914  
          e    1.323618  
          dtype: float64
```

```
In [139]: frame.apply(f, axis=1)
```

```
Out[139]: Utah      2.408544  
          Ohio      1.844201  
          Texas      1.503553  
          Oregon     1.233384  
          dtype: float64
```




Aplicação de função

- A função passada não precisa retornar um valor escalar, ela também pode retornar uma série com múltiplos valores:

```
def f(x):  
    return Series([x.min(), x.max()], index=['min', 'max'])  
frame.apply(f, axis=0)
```

	b	d	e
min	-1.483807	-0.956676	-0.869185
max	0.635156	-0.394914	1.323618



```
frame.apply(f, axis=1)
```

	min	max
Utah	-1.483807	0.924737
Ohio	-0.520583	1.323618
Texas	-0.869185	0.634368
Oregon	-0.598228	0.635156



Aplicação de função

- Suponha que você queira obter uma string formatada de cada valor de ponto flutuante no **DataFrame**; Para isto existe o **applymap**;
- A razão para o nome **applymap** é que o Series já possui uma função **map**.

```
format = lambda x: '%.2f' % x
```

```
frame.applymap(format)
```

	b	d	e
Utah	-1.48	-0.96	0.92
Ohio	-0.43	-0.52	1.32
Texas	0.63	-0.39	-0.87
Oregon	0.64	-0.60	-0.53

```
frame['e'].map(format)
```

```
Utah      -0.90  
Ohio       0.04  
Texas      0.58  
Oregon    -0.28  
Name: e, dtype: object
```



Ordenação

- Para classificar lexicograficamente por índice de linha ou coluna, use-se o método **sort_index**, que retorna um novo objeto ordenado:

```
obj = Series(range(4),  
             index=['d', 'a', 'b', 'c'])
```

```
obj.sort_index()
```

```
a    1  
b    2  
c    3  
d    0  
dtype: int64
```



Ordenação

- No caso de **DataFrame** a ordenação pode ser tanto por linhas quanto colunas:

```
frame = DataFrame(np.arange(8).reshape((2, 4)),  
                  index=['three', 'one'],  
                  columns=['d', 'a', 'b', 'c'])
```

	d	a	b	c
three	0	1	2	3
one	4	5	6	7

```
frame.sort_index()
```

	d	a	b	c
one	4	5	6	7
three	0	1	2	3

```
frame.sort_index(axis=1,  
                 ascending=False)
```

	d	c	b	a
three	0	3	2	1
one	4	7	6	5

Ordenação

- Para classificar uma **série** por seus valores, use o método **order()**.

```
In [175]: obj = Series([4, 7, -3, 2])
```

```
In [176]: obj.order()
```

```
Out[176]:
```

2	-3
3	2
0	4
1	7

```
In [177]: obj = Series([4, np.nan, 7, np.nan, -3, 2])
```

```
In [178]: obj.order()
```

```
Out[178]:
```

4	-3
5	2
0	4
2	7
1	NaN
3	NaN





Ordenação

- No **DataFrame**, você pode querer classificar pelos valores em uma ou mais colunas. Para fazer isso, passe um ou mais nomes de colunas para a opção `by`:

```
In [179]: frame = DataFrame({'b': [4, 7, -3, 2], 'a': [0, 1, 0, 1]})
```

```
In [180]: frame
```

```
Out[180]:
```

	a	b
0	0	4
1	1	7
2	0	-3
3	1	2

```
In [181]: frame.sort_index(by='b')
```

```
Out[181]:
```

	a	b
2	0	-3
3	1	2
0	0	4
1	1	7

```
In [182]: frame.sort_index(by=['a', 'b'])
```

```
Out[182]:
```

	a	b
2	0	-3
0	0	4
3	1	2
1	1	7



Gráficos alto nível

- Oferece uma forma mais “alto nível” para geração gráficos baseados na biblioteca **matplotlib**;
- Fornece um conjunto de opções que possibilitam a criação de gráficos diretamente a partir dos objetos **Pandas**;
- Para gráficos mais complexos, de fato, é necessário usar a própria **matplotlib** ou outra;
- Cada objeto Pandas já possui um método `.plot()` que aceita vários parâmetro de formatação;

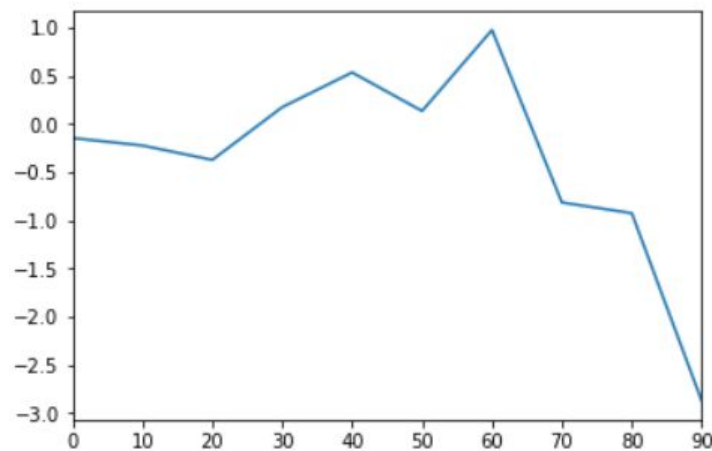
<http://pandas.pydata.org/pandas-docs/version/0.15.0/visualization.ht>

Plot Series

- O método `.plot()` depende da biblioteca **matplotlib**;

```
from pandas import Series, DataFrame
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

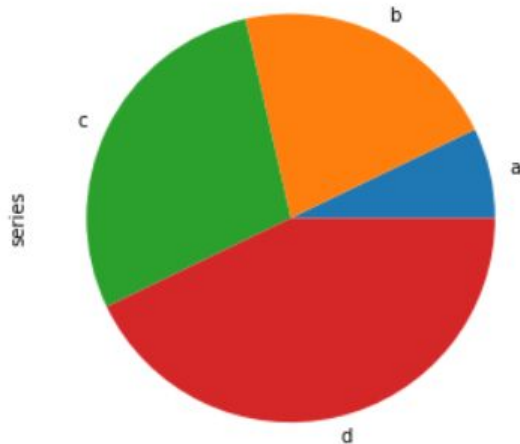
s = Series(np.random.randn(10).cumsum(),
           index=np.arange(0, 100, 10))
s.plot();
```



Plot Series

- Alterando o parâmetro **kind** é possível definir o tipo de gráfico:

```
series = Series([1,3,4,6],  
                index=['a', 'b', 'c', 'd'],  
                name='series')  
  
series.plot(kind='pie', figsize=(5, 5));
```



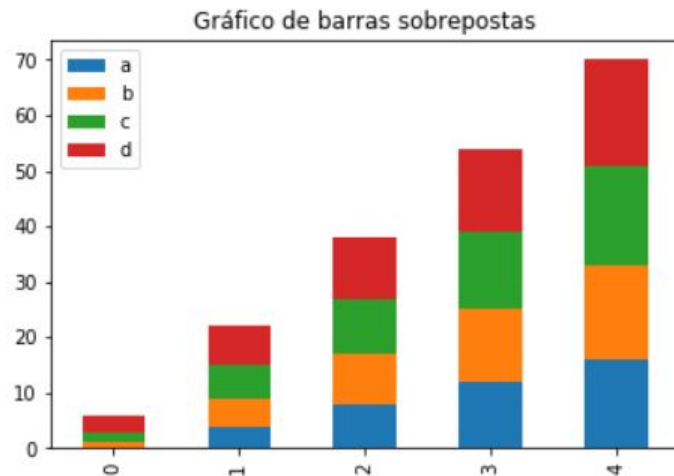
- Lista completa dos parâmetros admitidos:
<https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.plot.html>



Plot DataFrame

- Analogamente, é possível gerar gráficos diretamente a partir de DataFrames:

```
df2 = DataFrame(np.arange(20).reshape(5, 4),  
                columns=['a', 'b', 'c', 'd'])  
df2.plot(kind='bar',  
         title='Gráfico de barras sobrepostas',  
         stacked=True,);
```



- Lista completa dos parâmetros pode ser vista em:

<https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.plot.html>





Um repositório de instâncias de dados de diversos cenários

www.kaggle.com

Exercício

- Repita os dois primeiros pontos do exercício anterior;
- Gere um dataframe com apenas os registros de jogos a partir da década de noventa;
- Gere um DataFrame com todos os registros de jogos do Brasil;
- Encontre a partida com maior público (**Attendance**).



Obrigado

altinobasilio@inf.ufg.br

Dúvidas ou sugestões?

