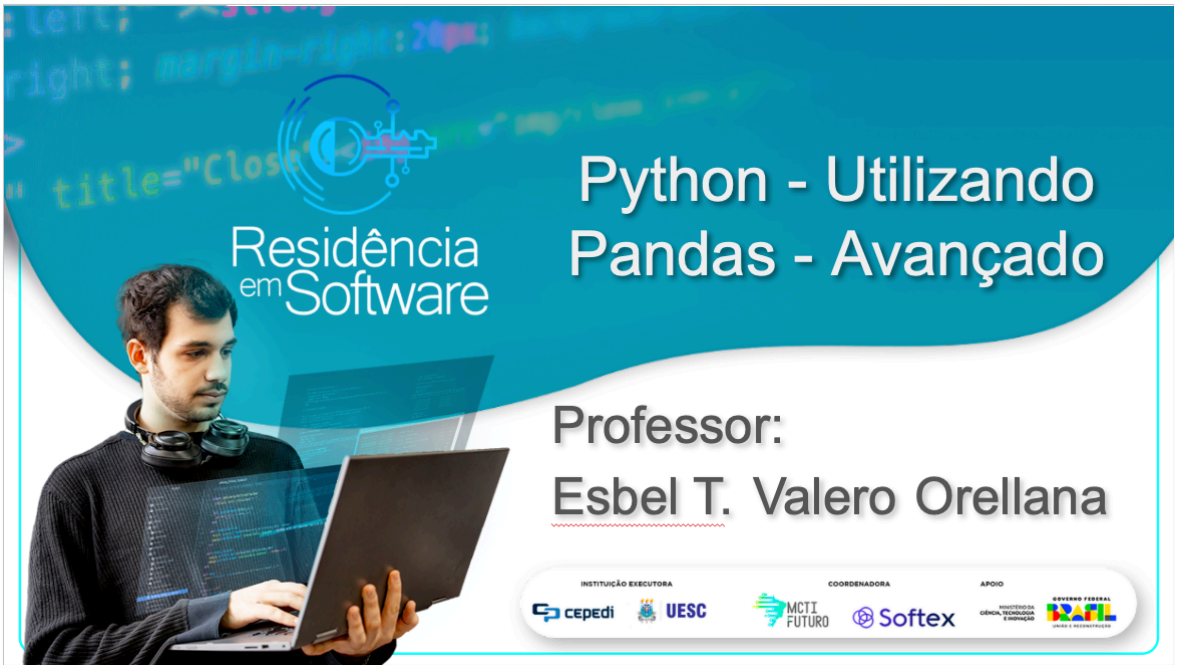
Módulo de Programação Python

Trilha Python - Aula 19: Utilizando Pandas - Avançado



Objetivo: Trabalhar com pacotes e módulos disponíveis em **Python**: Pandas: Apresentar recursos do **Pandas** para trabalhar conjuntos de dados.

Conteúdo: Agregação e Agrupamento: Agregação simples em Pandas. GroupBy: Dividir, Aplicar, Combinar. Operações de String Vetorizadas: Apresentando as operações de string do Pandas. Métodos de string Pandas.

Agregação e agrupamento

Uma parte essencial da análise de grandes volumes de dados é a compilação eficiente dos mesmo computando com funções de agregações como `sum()`, `mean()`, `median()`, `min()` e `max()`, em que um único número fornece informações sobre a natureza de um conjunto de dados potencialmente grande.

Nesta aula, exploraremos agregações no **Pandas**, desde operações simples semelhantes às que vimos em arrays **NumPy**, até operações mais sofisticadas baseadas no conceito de `groupby`.

In [1]:
```
import numpy as np
import pandas as pd
print("NumPy: ", np.__version__)
print("Pandas: ", pd.__version__)
```

```
Intel MKL WARNING: Support of Intel(R) Streaming SIMD Extensions
4.2 (Intel(R) SSE4.2) enabled only processors has been deprecated.
Intel oneAPI Math Kernel Library 2025.0 will require Intel(R) Adva
nced Vector Extensions (Intel(R) AVX) instructions.
Intel MKL WARNING: Support of Intel(R) Streaming SIMD Extensions
4.2 (Intel(R) SSE4.2) enabled only processors has been deprecated.
Intel oneAPI Math Kernel Library 2025.0 will require Intel(R) Adva
nced Vector Extensions (Intel(R) AVX) instructions.
NumPy:  1.26.2
Pandas:  2.1.4
```

In [2]:
```
class Display(object):
    """Permite exibir representação HTML de vários objetos"""

    template = """<div style="float: left; padding: 10px;">
    <p style='font-family:"Courier New", Courier, monospace'>{0}</p
    </div>"""
    def __init__(self, *args):
        self.args = args

    def _repr_html_(self):
        return '\n'.join(self.template.format(a, eval(a)._repr_html_
                         for a in self.args)

    def __repr__(self):
        return '\n\n'.join(a + '\n' + repr(eval(a))
                           for a in self.args)
```

Dados de Planetas

Aqui usaremos o conjunto de dados `Planets`, disponível no pacote Seaborn (http://seaborn.pydata.org/).

Este *dataset* fornece informações sobre planetas que os astrônomos descobriram em torno de outras estrelas, conhecidos como *planetas extrassolares* ou *exoplanetas*, para abreviar.

In [4]:
```
import seaborn as sns
planetas = sns.load_dataset('planets')
planetas.shape
```

Out[4]:
```
(1035, 6)
```

In [5]:
```
planetas.head()
```

Out[5]:

	method	number	orbital_period	mass	distance	year
0	Radial Velocity	1	269.300	7.10	77.40	2006
1	Radial Velocity	1	874.774	2.21	56.95	2008
2	Radial Velocity	1	763.000	2.60	19.84	2011
3	Radial Velocity	1	326.030	19.40	110.62	2007
4	Radial Velocity	1	516.220	10.50	119.47	2009

Agregação simples em Pandas

Anteriormente, exploramos algumas das agregações de dados disponíveis para matrizes **NumPy** .

Tal como acontece com um *ndarray* **NumPy** unidimensional, para um **Pandas** `Series` as funções de agregação retornam um único valor.

In [6]:
```
rng = np.random.RandomState(42)
ser = pd.Series(rng.rand(5))
ser
```

Out[6]:
```
0    0.374540
1    0.950714
2    0.731994
3    0.598658
4    0.156019
dtype: float64
```

In [7]:
```
ser.sum()
```

Out[7]:
```
2.811925491708157
```

In [8]:
```
ser.mean()
```

Out[8]:
```
0.5623850983416314
```

Já para um `DataFrame` , por padrão as funções de agregação retornam resultados por coluna.

In [9]:
```
df = pd.DataFrame({'A': rng.rand(5),
                   'B': rng.rand(5)})
df
```

Out[9]:

	A	B
0	0.155995	0.020584
1	0.058084	0.969910
2	0.866176	0.832443
3	0.601115	0.212339
4	0.708073	0.181825

In [10]:
```
df.sum()
```

Out[10]:
```
A    2.389442
B    2.217101
dtype: float64
```

In [11]:
```
df.mean()
```

Out[11]:
```
A    0.477888
B    0.443420
dtype: float64
```

Se especificar o argumento `axis` , você pode agregar por linha linha.

In [12]:
```
df.sum(axis='columns')
```

Out[12]:
```
0    0.176579
1    1.027993
2    1.698619
3    0.813454
4    0.889898
dtype: float64
```

In [13]:
```
df.mean(axis='columns')
```

Out[13]:
```
0    0.088290
1    0.513997
2    0.849309
3    0.406727
4    0.444949
dtype: float64
```

Os objetos `Series` e `DataFrame` do **Pandas** incluem todas as funções de agregação comuns que analisamos no **NumPy**.

Além disso, existe um método muito útil, o `describe()`, que calcula vários valores agregados comuns para cada coluna e retorna o resultado.

In [14]:
```
planetas.dropna().describe()
```

Out[14]:

	number	orbital_period	mass	distance	year
count	498.00000	498.000000	498.000000	498.000000	498.000000
mean	1.73494	835.778671	2.509320	52.068213	2007.377510
std	1.17572	1469.128259	3.636274	46.596041	4.167284
min	1.00000	1.328300	0.003600	1.350000	1989.000000
25%	1.00000	38.272250	0.212500	24.497500	2005.000000
50%	1.00000	357.000000	1.245000	39.940000	2009.000000
75%	2.00000	999.600000	2.867500	59.332500	2011.000000
max	6.00000	17337.500000	25.000000	354.000000	2014.000000

Essa pode ser uma maneira útil de começar a compreender as propriedades gerais de um conjunto de dados.

Por exemplo, vemos na coluna do ano que, embora os exoplanetas tenham sido descobertos já em 1989, metade de todos os exoplanetas conhecidos não foram descobertos em 2010 ou depois.

Isto se deve em grande parte à missão Kepler, que é um telescópio espacial projetado especificamente para encontrar planetas eclipsantes em torno de outras estrelas.

A tabela a seguir resume algumas outras funções de agregação do **Pandas**.

Função de agregação	O que faz
`count()`	Número total de itens
`first()` , `last()`	Primeiro e último item
`mean()` , `median()`	Média e mediana
`min()` , `max()`	Mínimo e máximo
`std()` , `var()`	Desvio padrão e variância
`mad()`	Desvio médio absoluto
`prod()`	Produto de todos os itens
`sum()`	Soma de todos os itens

Estes métodos estão dirponíveis nos objetos `DataFrame e Series` .

Para aprofundar nas características do conjunto de dados, no entanto, função de agregação simples muitas vezes não são suficientes.

O próximo nível de resumo de dados é a operação `groupby` , que permite calcular agregados em subconjuntos de dados de forma rápida e eficiente.

GroupBy: Dividir, Aplicar, Combinar

Operações de agregações simples podem lhe dar uma ideia do seu conjunto de dados, mas muitas vezes preferiríamos agregar condicionalmente em algum rótulo ou índice. Estetipo de operação é implementado na chamada operação `groupby` .

O nome "groupby" vem de um comando na linguagem de banco de dados SQL, mas talvez seja mais esclarecedor pensar nele nos termos cunhados pela primeira vez por Hadley Wickham, famoso pelo Rstats: *dividir, aplicar, combinar*.

Dividir, aplicar, combinar

Um exemplo canônico desta operação split-apply-combine, onde "apply" é uma agregação de soma, é ilustrado nesta figura:

Isso deixa claro o que o `groupby` realiza:

- O *split* envolve dividir e agrupar um `DataFrame` dependendo do valor da chave especificada.
- A etapa *aplicar* envolve calcular alguma função, geralmente uma agregação, transformação ou filtragem, dentro dos grupos individuais.
- A etapa *combine* mescla os resultados dessas operações em uma matriz de saída.

Embora isso certamente possa ser feito manualmente usando alguma combinação dos comandos de mascaramento, agregação e mesclagem abordados anteriormente, uma constatação importante é que *as divisões intermediárias não precisam ser explicitamente instanciadas*.

Em vez disso, `GroupBy` pode (frequentemente) fazer isso em uma única passagem pelos dados, atualizando a soma, média, contagem, mínimo ou outro agregado para cada grupo ao longo do caminho.

O poder do `GroupBy` é que ele abstrai essas etapas: o usuário não precisa pensar sobre *como* o cálculo é feito nos bastidores, mas sim pensar sobre a *operação como um todo*.

Como exemplo concreto, vamos dar uma olhada no uso do **Pandas** para o cálculo mostrado neste diagrama.

Começaremos criando a entrada `DataFrame`

In [15]:
```
df = pd.DataFrame({'key': ['A', 'B', 'C', 'A', 'B', 'C'],
                   'data': range(6)}, columns=['key', 'data'])
df
```

Out[15]:

	key	data
0	A	0
1	B	1
2	C	2
3	A	3
4	B	4
5	C	5

A operação mais básica de dividir-aplicar-combinar pode ser calculada com o método groupby() de DataFrames, passando o nome da coluna-chave desejada:

In [16]:
```
df.groupby('key')
```

Out[16]:
```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x7fd1714e9f30>
```

Repare que o que é retornado não é um conjunto de `DataFrame` s, mas um objeto `DataFrameGroupBy` .

Para produzir um resultado, podemos aplicar uma agregação a este objeto `DataFrameGroupBy` , que executará as etapas de aplicação/combinação apropriadas para produzir o resultado desejado.

In [17]:
```
df.groupby('key').sum()
```

Out[17]:

key	data
A	3
B	5
C	7

O método `sum()` é apenas uma possibilidade.

Podemos aplicar praticamente qualquer função de agregação comum do **Pandas** ou **NumPy**, bem como praticamente qualquer operação `DataFrame` válida.

A classe GroupBy

Os objetos da classe `GroupBy` são uma abstração muito flexível.

De muitas maneiras, você pode simplesmente tratá-lo como se fosse uma coleção de `DataFrame` s, e ele faz as coisas difíceis nos bastidores. Vejamos alguns exemplos usando os dados dos Planetas.

Indexação de colunas

O objeto `GroupBy` suporta indexação de colunas da mesma forma que `DataFrame` , e retorna um objeto `GroupBy` modificado.

In [18]:
```
planetas.head()
```

Out[18]:

	method	number	orbital_period	mass	distance	year
0	Radial Velocity	1	269.300	7.10	77.40	2006
1	Radial Velocity	1	874.774	2.21	56.95	2008
2	Radial Velocity	1	763.000	2.60	19.84	2011
3	Radial Velocity	1	326.030	19.40	110.62	2007
4	Radial Velocity	1	516.220	10.50	119.47	2009

In [19]:
```
planetas.groupby('method')
```

Out[19]:
```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x7fd1714eb670>
```

In [20]:
```
planetas.groupby('method')['orbital_period']
```

Out[20]:
```
<pandas.core.groupby.generic.SeriesGroupBy object at 0x7fd1808383d0>
```

Aqui selecionamos um `Series` específico do grupo `DataFrame` original referenciando o nome de sua coluna.

Assim como acontece com o objeto `GroupBy` , nenhum cálculo é feito até que chamemos alguma agregação no objeto.

In [21]:
```
planetas.groupby('method')['orbital_period'].median()
```

Out[21]:
```
method
Astrometry                         631.180000
Eclipse Timing Variations         4343.500000
Imaging                          27500.000000
Microlensing                      3300.000000
Orbital Brightness Modulation        0.342887
Pulsar Timing                       66.541900
Pulsation Timing Variations       1170.000000
Radial Velocity                    360.200000
Transit                              5.714932
Transit Timing Variations           57.011000
Name: orbital_period, dtype: float64
```

Isto dá uma ideia da escala geral dos períodos orbitais (em dias) aos quais cada método é sensível.

Iteração sobre grupos

O objeto `GroupBy` suporta iteração direta sobre os grupos, retornando cada grupo como um `Series` ou `DataFrame` .

In [22]:
```
for (method, group) in planets.groupby('method'):
    print("{0:30s} shape={1}".format(method, group.shape))
```

```
Astrometry                     shape=(2, 6)
Eclipse Timing Variations      shape=(9, 6)
Imaging                        shape=(38, 6)
Microlensing                   shape=(23, 6)
Orbital Brightness Modulation  shape=(3, 6)
Pulsar Timing                  shape=(5, 6)
Pulsation Timing Variations    shape=(1, 6)
Radial Velocity                shape=(553, 6)
Transit                        shape=(397, 6)
Transit Timing Variations      shape=(4, 6)
```

Isto pode ser útil para fazer certas coisas manualmente, embora muitas vezes seja muito mais rápido usar a funcionalidade integrada `apply` , que discutiremos em posteriormente.

Métodos de envio

Através de alguma mágica de classe **Python**, qualquer método não explicitamente implementado pelo objeto `GroupBy` será passado e chamado nos grupos, sejam eles objetos `DataFrame` ou `Series` .

Por exemplo, você pode usar o método `describe()` de `DataFrame` s para realizar um conjunto de agregações que descrevem cada grupo nos dados.

In [26]:
```
planetas.groupby('method')['year'].describe()
```

Out[26]:

method	count	mean	std	min	25%	50%	75%	max
Astrometry	2.0	2011.500000	2.121320	2010.0	2010.75	2011.5	2012.25	2013.0
Eclipse Timing Variations	9.0	2010.000000	1.414214	2008.0	2009.00	2010.0	2011.00	2012.0
Imaging	38.0	2009.131579	2.781901	2004.0	2008.00	2009.0	2011.00	2013.0
Microlensing	23.0	2009.782609	2.859697	2004.0	2008.00	2010.0	2012.00	2013.0
Orbital Brightness Modulation	3.0	2011.666667	1.154701	2011.0	2011.00	2011.0	2012.00	2013.0
Pulsar Timing	5.0	1998.400000	8.384510	1992.0	1992.00	1994.0	2003.00	2011.0
Pulsation Timing Variations	1.0	2007.000000	NaN	2007.0	2007.00	2007.0	2007.00	2007.0
Radial Velocity	553.0	2007.518987	4.249052	1989.0	2005.00	2009.0	2011.00	2014.0
Transit	397.0	2011.236776	2.077867	2002.0	2010.00	2012.0	2013.00	2014.0
Transit Timing Variations	4.0	2012.500000	1.290994	2011.0	2011.75	2012.5	2013.25	2014.0

Olhar para esta tabela ajuda-nos a compreender melhor os dados.

Por exemplo, a grande maioria dos planetas foram descobertos pelos métodos de Velocidade Radial e Trânsito, embora este último só se tenha tornado comum (devido a telescópios novos e mais precisos) na última década.

Os métodos mais recentes parecem ser a Variação do Tempo de Trânsito e a Modulação de Brilho Orbital, que não foram usados para descobrir um novo planeta até 2011.

Observe que este método foi aplicados *a cada grupo individual*, e os resultados são então combinados em `GroupBy` e retornados.

Novamente, qualquer método `DataFrame` / `Series` válido pode ser usado no objeto `GroupBy` correspondente, o que permite algumas operações muito flexíveis e poderosas!

Agregar, filtrar, transformar, aplicar

A discussão anterior focou na agregação para a operação combinada, mas há mais opções disponíveis.

Em particular, objetos `GroupBy` possuem métodos `agregate()`, `filter()`, `transform()` e `apply()` que implementam eficientemente uma variedade de funções úteis. operações antes de combinar os dados agrupados.

In [27]:
```
rng = np.random.RandomState(0)
df = pd.DataFrame({'key': ['A', 'B', 'C', 'A', 'B', 'C'],
                   'data1': range(6),
                   'data2': rng.randint(0, 10, 6)},
                   columns = ['key', 'data1', 'data2'])
df
```

Out[27]:

	key	data1	data2
0	A	0	5
1	B	1	0
2	C	2	3
3	A	3	3
4	B	4	7
5	C	5	9

Agregação

Agora estamos familiarizados com agregações `GroupBy` com `sum()`, `median()`, e similares, mas o método `agregate()` permite ainda mais flexibilidade.

Este método pode pegar uma string, uma função ou uma lista delas e calcular todas as agregações de uma vez.

In [28]:
```
df.groupby('key').aggregate(['min', np.median, max])
```

```
/var/folders/k8/m4nwfbmx7517ksbqt7fkxclm0000gn/T/ipykernel_46566/9
68873422.py:1: FutureWarning: The provided callable <function medi
an at 0x7fd18067fbe0> is currently using SeriesGroupBy.median. In
a future version of pandas, the provided callable will be used dir
ectly. To keep current behavior pass the string "median" instead.
  df.groupby('key').aggregate(['min', np.median, max])
/var/folders/k8/m4nwfbmx7517ksbqt7fkxclm0000gn/T/ipykernel_46566/9
68873422.py:1: FutureWarning: The provided callable <built-in func
tion max> is currently using SeriesGroupBy.max. In a future versio
n of pandas, the provided callable will be used directly. To keep
current behavior pass the string "max" instead.
  df.groupby('key').aggregate(['min', np.median, max])
```

Out[28]:

	data1			data2		
	min	median	max	min	median	max
key						
A	0	1.5	3	3	4.0	5
B	1	2.5	4	0	3.5	7
C	2	3.5	5	3	6.0	9

Outra opção útil é passar nomes de colunas de mapeamento de dicionário para operações a serem aplicadas nessas colunas.

In [29]:
```
df.groupby('key').aggregate({'data1': 'min',
                             'data2': 'max'})
```

Out[29]:

	data1	data2
key		
A	0	5
B	1	7
C	2	9

Filtragem

Uma operação de filtragem permite eliminar dados com base nas propriedades do grupo. Por exemplo, podemos querer manter todos os grupos nos quais o desvio padrão é maior que algum valor crítico.

In [33]:
```
def filter_func(x):
    return x['data2'].std() > 4
Display('df', "df.groupby('key').std()", "df.groupby('key').filter(
```

Out[33]:

df

	key	data1	data2
0	A	0	5
1	B	1	0
2	C	2	3
3	A	3	3
4	B	4	7
5	C	5	9

df.groupby('key').std()

	data1	data2
key		
A	2.12132	1.414214
B	2.12132	4.949747
C	2.12132	4.242641

df.groupby('key').filter(filter_func)

	key	data1	data2
1	B	1	0
2	C	2	3
4	B	4	7
5	C	5	9

A função de filtro deve retornar um valor booleano especificando se o grupo passa na filtragem. Aqui, como o grupo A não tem desvio padrão maior que 4, ele é eliminado do resultado.

Transformação

Embora a agregação deva retornar uma versão reduzida dos dados, a transformação pode retornar alguma versão transformada dos dados completos para recombinação.

Para tal transformação, a saída tem o mesmo formato da entrada. Um exemplo comum é centralizar os dados subtraindo a média do grupo.

In [34]:
```
Display('df', "df.groupby('key').transform(lambda x: x - x.mean())"
```

Out[34]:

df

	key	data1	data2
0	A	0	5
1	B	1	0
2	C	2	3
3	A	3	3
4	B	4	7
5	C	5	9

df.groupby('key').transform(lambda x: x - x.mean())

	data1	data2
0	-1.5	1.0
1	-1.5	-3.5
2	-1.5	-3.0
3	1.5	-1.0
4	1.5	3.5
5	1.5	3.0

O método apply()

O método `apply()` permite aplicar uma função arbitrária aos resultados do grupo. A função deve receber um `DataFrame` e retornar um objeto Pandas (por exemplo, `DataFrame` , `Series`) ou um escalar; a operação de combinação será adaptada ao tipo de saída retornada.

Por exemplo, aqui está um `apply()` que normaliza a primeira coluna pela soma da segunda.

In [36]:
```
def norm_by_data2(x):
    # x is a DataFrame of group values
    x['data1'] /= x['data2'].sum()
    return x

Display('df', "df.groupby('key').apply(norm_by_data2)")
```

Out[36]:

df

	key	data1	data2
0	A	0	5
1	B	1	0
2	C	2	3
3	A	3	3
4	B	4	7
5	C	5	9

df.groupby('key').apply(norm_by_data2)

		key	data1	data2
key				
A	**0**	A	0.000000	5
	3	A	0.375000	3
B	**1**	B	0.142857	0
	4	B	0.571429	7
C	**2**	C	0.166667	3
	5	C	0.416667	9

O método `apply()` de um `GroupBy` é bastante flexível.

O único critério é que a função pegue um `DataFrame` e retorne um objeto Pandas ou escalar.

O que você faz no meio é com você!

Especificando a chave ``split```

Nos exemplos simples apresentados anteriormente, dividimos o `DataFrame` em um único nome de coluna.

Esta é apenas uma das muitas opções pelas quais os grupos podem ser definidos, e examinaremos algumas outras opções para especificação de grupos.

A chave na forma de uma lista

Uma lista ou array, pode ser utilizada para especificar o índice que fornece as chaves de agrupamento.

A chave pode ser qualquer série ou lista com comprimento correspondente ao `DataFrame` .

In [45]:
```
L = [0, 1, 0, 1, 2, 0]
Display('df', 'df.groupby(L).sum()')
```

Out[45]:

df

	key	data1	data2
0	A	0	5
1	B	1	0
2	C	2	3
3	A	3	3
4	B	4	7
5	C	5	9

df.groupby(L).sum()

	key	data1	data2
0	ACC	7	17
1	BA	4	3
2	B	4	7

Claro, isso significa que há outra maneira mais detalhada de realizar o `df.groupby('key')` .

In [47]:
```
Display('df', "df.groupby(df['key']).sum()")
```

Out[47]:

df

	key	data1	data2
0	A	0	5
1	B	1	0
2	C	2	3
3	A	3	3
4	B	4	7
5	C	5	9

df.groupby(df['key']).sum()

key	data1	data2
A	3	8
B	5	7
C	7	12

A chave como um dicionário

Outro método é fornecer um dicionário que mapeie valores de índice para as chaves do grupo.

In [48]:
```
df2 = df.set_index('key')
Display('df', 'df2')
```

Out[48]:

df

	key	data1	data2
0	A	0	5
1	B	1	0
2	C	2	3
3	A	3	3
4	B	4	7
5	C	5	9

df2

	data1	data2
key		
A	0	5
B	1	0
C	2	3
A	3	3
B	4	7
C	5	9

In [50]:
```
mapping = {'A': 'vowel', 'B': 'consonant', 'C': 'consonant'}
Display('df2', 'df2.groupby(mapping).sum()')
```

Out[50]:

df2

	data1	data2
key		
A	0	5
B	1	0
C	2	3
A	3	3
B	4	7
C	5	9

df2.groupby(mapping).sum()

	data1	data2
key		
consonant	12	19
vowel	3	8

Qualquer função Python

Semelhante ao mapeamento, você pode passar qualquer função **Python** que irá inserir o valor do índice e gerar o grupo.

In [52]:
```
Display('df2', 'df2.groupby(str.lower).mean()')
```

Out[52]:

df2

key	data1	data2
A	0	5
B	1	0
C	2	3
A	3	3
B	4	7
C	5	9

df2.groupby(str.lower).mean()

key	data1	data2
a	1.5	4.0
b	2.5	3.5
c	3.5	6.0

Uma lista de chaves válidas

Além disso, qualquer uma das opções principais anteriores pode ser combinada para agrupar em um índice múltiplo.

In [53]:
```
df2.groupby([str.lower, mapping]).mean()
```

Out[53]:

key	key	data1	data2
a	vowel	1.5	4.0
b	consonant	2.5	3.5
c	consonant	3.5	6.0

Exemplo de agrupamento

Como exemplo disso, em algumas linhas de código Python podemos juntar tudo isso e contar os planetas descobertos por método e por década.

In [55]:
```
decade = 10 * (planetas['year'] // 10)
decade = decade.astype(str) + 's'
decade.name = 'decade'
planetas.groupby(['method', decade])['number'].sum().unstack().fill
```

Out[55]:

decade	1980s	1990s	2000s	2010s
method				
Astrometry	0.0	0.0	0.0	2.0
Eclipse Timing Variations	0.0	0.0	5.0	10.0
Imaging	0.0	0.0	29.0	21.0
Microlensing	0.0	0.0	12.0	15.0
Orbital Brightness Modulation	0.0	0.0	0.0	5.0
Pulsar Timing	0.0	9.0	1.0	1.0
Pulsation Timing Variations	0.0	0.0	1.0	0.0
Radial Velocity	1.0	52.0	475.0	424.0
Transit	0.0	0.0	64.0	712.0
Transit Timing Variations	0.0	0.0	0.0	9.0

Isso mostra o poder de combinar muitas das operações que discutimos até agora ao observar conjuntos de dados realistas. Adquirimos imediatamente uma compreensão aproximada de quando e como os planetas foram descobertos nas últimas décadas!

Aqui eu sugeriria aprofundar essas poucas linhas de código e avaliar as etapas individuais para ter certeza de que você entendeu exatamente o que elas estão fazendo com o resultado.

Certamente é um exemplo um tanto complicado, mas a compreensão dessas peças lhe dará os meios para explorar de forma semelhante seus próprios dados.

Tabelas dinâmicas

Até aqui vimos como a abstração `GroupBy` nos permite explorar relacionamentos dentro de um conjunto de dados.

Uma tabela dinâmica (*pivot tables*) é uma operação semelhante, comumente vista em planilhas e outros programas que operam em dados tabulares.

A tabela dinâmica usa dados simples de colunas como entrada e agrupa as entradas em uma tabela bidimensional que fornece um resumo multidimensional dos dados.

A diferença entre tabelas dinâmicas e `GroupBy` às vezes pode causar confusão. Você pode pensar nas tabelas dinâmicas como, essencialmente, uma versão *multidimensional* da agregação `GroupBy` .

Ou seja, você divide-aplica-combina, mas tanto a divisão quanto a combinação acontecem não em um índice unidimensional, mas em uma grade bidimensional.

In [56]:
```
titanic = sns.load_dataset('titanic')
titanic.head()
```

Out[56]:

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_m
0	0	3	male	22.0	1	0	7.2500	S	Third	man	T
1	1	1	female	38.0	1	0	71.2833	C	First	woman	Fa
2	1	3	female	26.0	0	0	7.9250	S	Third	woman	Fa
3	1	1	female	35.0	1	0	53.1000	S	First	woman	Fa
4	0	3	male	35.0	0	0	8.0500	S	Third	man	T

Este dataset contém varias informações sobre cada passageiro da viagem malfadada do Titanic, incluindo sexo, idade, classe, tarifa paga e muito mais.

Tabelas dinâmicas manualmente

Para começar a aprender mais sobre estes dados, podemos começar por agrupar de acordo com o gênero, e se sobreviveram ou não ou alguma combinação destas colunas.

In [57]:
```
titanic.groupby('sex')[['survived']].mean()
```

Out[57]:

sex	survived
female	0.742038
male	0.188908

Isto imediatamente nos dá uma ideia: no geral, três em cada quatro mulheres a bordo sobreviveram, enquanto apenas um em cada cinco homens sobreviveu!

Gostaríamos de ir um pouco mais fundo e analisar a sobrevivência tanto por sexo como, digamos, por classe.

Usando o vocabulário de `GroupBy`, podemos prosseguir usando algo assim: nós *agrupamos* por classe e gênero, *selecionamos* a sobrevivência, *aplicamos* um agregado médio, *combinamos* os grupos resultantes e então *desempilhamos* o índice hierárquico para revelar a multidimensionalidade oculta.

In [58]:
```
titanic.groupby(['sex', 'class'])['survived'].aggregate('mean').uns
```

```
/var/folders/k8/m4nwfbmx7517ksbqt7fkxclm0000gn/T/ipykernel_46566/2
603839867.py:1: FutureWarning: The default of observed=False is de
precated and will be changed to True in a future version of panda
s. Pass observed=False to retain current behavior or observed=True
to adopt the future default and silence this warning.
  titanic.groupby(['sex', 'class'])['survived'].aggregate('mean').
unstack()
```

Out[58]:

class	First	Second	Third
sex			
female	0.968085	0.921053	0.500000
male	0.368852	0.157407	0.135447

Isto dá-nos uma ideia melhor de como tanto o gênero como a classe influenciaram na taxa de sobrevivência, mas o código está começando a parecer um pouco complexo.

Embora cada etapa desse pipeline faça sentido à luz das ferramentas que discutimos anteriormente, a longa sequência de código não é particularmente fácil de ler ou usar.

Este `GroupBy` bidimensional é comum o suficiente para que o **Pandas** inclua uma rotina, `pivot_table`, que lida sucintamente com esse tipo de agregação multidimensional.

Sintaxe de tabelaq dinâmicaq

Aqui está o equivalente à operação anterior usando o método `pivot_table` de `DataFrame` s:

In [59]:
```
titanic.pivot_table('survived', index='sex', columns='class')
```

Out[59]:

sex	class	First	Second	Third
female		0.968085	0.921053	0.500000
male		0.368852	0.157407	0.135447

Este código é mais legível que a abordagem `groupby` e produz o mesmo resultado.

Como seria de esperar de um cruzeiro transatlântico do início do século XX, o gradiente de sobrevivência favorece tanto as mulheres como as classes mais altas.

Tabelas dinâmicas multinível

Assim como em `GroupBy` , o agrupamento em tabelas dinâmicas pode ser especificado com múltiplos níveis e através de diversas opções.

Por exemplo, podemos estar interessados em considerar a idade como uma terceira dimensão. Iremos agrupar a idade usando a função `pd.cut` :

In [60]:
```
age = pd.cut(titanic['age'], [0, 18, 80])
titanic.pivot_table('survived', ['sex', age], 'class')
```

Out[60]:

sex	age	class	First	Second	Third
female	(0, 18]		0.909091	1.000000	0.511628
	(18, 80]		0.972973	0.900000	0.423729
male	(0, 18]		0.800000	0.600000	0.215686
	(18, 80]		0.375000	0.071429	0.133663

Também podemos aplicar a mesma estratégia ao trabalhar com as colunas; vamos adicionar informações sobre a tarifa paga usando `pd.qcut` para calcular quantis automaticamente.

In [61]:
```
fare = pd.qcut(titanic['fare'], 2)
titanic.pivot_table('survived', ['sex', age], [fare, 'class'])
```

Out[61]:

	fare	(-0.001, 14.454]			(14.454, 512.329]		
	class	First	Second	Third	First	Second	Third
sex	age						
female	(0, 18]	NaN	1.000000	0.714286	0.909091	1.000000	0.318182
	(18, 80]	NaN	0.880000	0.444444	0.972973	0.914286	0.391304
male	(0, 18]	NaN	0.000000	0.260870	0.800000	0.818182	0.178571
	(18, 80]	0.0	0.098039	0.125000	0.391304	0.030303	0.192308

O resultado é uma agregação quadridimensional com índices hierárquicos, que demonstra a relação entre os valores.

Opções adicionais de tabela dinâmica

A assinatura completa da chamada do método `pivot_table` de `DataFrame` s é a seguinte:

```
# Método em andas 2.14
DataFrame.pivot_table(values=None, index=None, columns=Non
e, aggfunc='mean',
                      fill_value=None, margins=False, dropn
a=True,
                      margins_name='All', observed=False, s
ort=True)
```

Duas das opções, `fill_value` e `dropna` , têm a ver com dados faltantes e são bastante diretas.

A palavra-chave `aggfunc` controla que tipo de agregação é aplicada, que por padrão é a média.

Como no GroupBy, a especificação de agregação pode ser uma string representando uma das várias escolhas comuns (por exemplo, `'sum'` , `'mean'` , `'count'` , `'min'` , `'max'` , etc.) ou uma função que implementa uma agregação (por exemplo, `np.sum()` , `min()` , `sum()` , etc.).

Além disso, pode ser especificado como um dicionário mapeando uma coluna para qualquer uma das opções desejadas acima.

In [62]:
```
titanic.pivot_table(index='sex', columns='class',
                    aggfunc={'survived':sum, 'fare':'mean'})
```

```
/var/folders/k8/m4nwfbmx7517ksbqt7fkxclm0000gn/T/ipykernel_46566/3
92095072.py:1: FutureWarning: The provided callable <built-in func
tion sum> is currently using SeriesGroupBy.sum. In a future versio
n of pandas, the provided callable will be used directly. To keep
current behavior pass the string "sum" instead.
  titanic.pivot_table(index='sex', columns='class',
```

Out[62]:

	fare			survived		
class	First	Second	Third	First	Second	Third
sex						
female	106.125798	21.970121	16.118810	91	70	72
male	67.226127	19.741782	12.661633	45	17	47

Repare que aqui omitimos a palavra-chave `values` ; ao especificar um mapeamento para `aggfunc` , isso é determinado automaticamente.

Às vezes é útil calcular valores totais ao longo de cada agrupamento. Isso pode ser feito através da palavra-chave `margins`

In [63]:
```
titanic.pivot_table('survived', index='sex', columns='class', margi
```

Out[63]:

class	First	Second	Third	All
sex				
female	0.968085	0.921053	0.500000	0.742038
male	0.368852	0.157407	0.135447	0.188908
All	0.629630	0.472826	0.242363	0.383838

Operações de strings vetorizadas

Um ponto forte do **Python** é sua relativa facilidade no tratamento e manipulação de dados de tipo *string*.

O **Pandas** se aproveita esta vantagem e fornece um conjunto abrangente de *operações de strings vetorizadas* que se tornam uma peça essencial do tipo de manipulação necessária ao trabalhar com dados do mundo real.

Apresentando operações de string do Pandas

Vimos nas seções anteriores como ferramentas como **NumPy** e **Pandas** generalizam operações aritméticas para que possamos executar a mesma operação de maneira fácil e rápida em muitos elementos do array.

A vetorização de operações simplifica a sintaxe de operação em matrizes de dados. Não precisamos mais nos preocupar com o tamanho ou formato da matriz, mas apenas com qual operação queremos realizar.

Para matrizes de *strings*, o **NumPy** não fornece um acesso tão simples e, portanto, você fica preso ao usar uma sintaxe de loop explícito.

In [64]:
```
data = ['peter', 'Paul', 'MARY', 'gUIDO']
[s.capitalize() for s in data]
```

Out[64]:
```
['Peter', 'Paul', 'Mary', 'Guido']
```

Talvez isso seja suficiente para trabalhar com alguns dados, mas não funciona se houver algum valor ausente.

In [66]:
```
data = ['peter', 'Paul', None, 'MARY', 'gUIDO']
try:
    [s.capitalize() for s in data]
except Exception as e:
    print(e)
```

```
'NoneType' object has no attribute 'capitalize'
```

O **Pandas** inclui recursos para atender a essa necessidade de operações de *strings* vetorizadas e para lidar corretamente com dados ausentes por meio do atributo `str` dos objetos **Pandas** `Series e Index` contendo *strings*.

In [67]:
```
names = pd.Series(data)
names
```

Out[67]:
```
0    peter
1     Paul
2     None
3     MARY
4    gUIDO
dtype: object
```

Agora podemos chamar um único método que colocará todas as entradas em maiúscula, ignorando quaisquer valores ausentes.

In [68]:
```
names.str.capitalize()
```

Out[68]:
```
0    Peter
1     Paul
2     None
3     Mary
4    Guido
dtype: object
```

Tabelas de métodos de string do Pandas

Se você tiver um bom entendimento da manipulação de *strings* em **Python**, a maior parte da sintaxe de *strings* do **Pandas** é bastante intuitiva. O suficiente para que provavelmente seja necessário apenas listar uma tabela de métodos disponíveis.

In [70]:
```
turma = pd.Series(["Thiago", "Renata", "Luciano", "Luis", "Nairan",
                   "Matheus", "Rafaela", "Tales", "Manoel", "Sérgio
                   "RICARDO", "Gabriel", "Arthur", "Everaldina", "V
                   "Girleide", "João", "Paulo", "Brenndol", "Erika"
                   "Danilo", "Raíssa", "Leane", "Vitor", "Myllena",
```

Métodos semelhantes aos métodos de string do Python

Quase todos os métodos de *string* do **Python** são espelhados por um método de *string* vetorizado do **Pandas**. Aqui está uma lista de métodos `str` do **Pandas** que espelham os métodos de string do **Python**.

len()	lower()	translate()	islower()
ljust()	upper()	startswith()	isupper()
rjust()	find()	endswith()	isnumeric()
center()	rfind()	isalnum()	isdecimal()
zfill()	index()	isalpha()	split()
strip()	rindex()	isdigit()	rsplit()
rstrip()	capitalize()	isspace()	partition()
lstrip()	swapcase()	istitle()	rpartition()

Importante destacar que estes métodos têm vários valores de retorno. Alguns, como `lower()`, retornam uma série de strings.

In [72]:
```
turma.str.lower()
```

Out[72]:
```
0         thiago
1         renata
2        luciano
3           luis
4         nairan
5         thiago
6        matheus
7        rafaela
8          tales
9         manoel
10        sérgio
11        allana
12       ricardo
13       gabriel
14        arthur
15    everaldina
16      vinicius
17      girleide
18          joão
19         paulo
20      brenndol
21         erika
22           ian
23        danilo
24        raíssa
25         leane
26         vitor
27       myllena
28          jose
29        marcos
dtype: object
```

Mas alguns outros retornam valores numéricos.

In [73]:
```
turma.str.len()
```

Out[73]:
```
0      6
1      6
2      7
3      4
4      6
5      6
6      7
7      7
8      5
9      6
10     6
11     6
12     7
13     7
14     6
15    10
16     8
17     8
18     4
19     5
20     8
21     5
22     3
23     6
24     6
25     5
26     5
27     7
28     4
29     6
dtype: int64
```

Ou valores booleanos.

In [74]:
```
turma.str.startswith('T')
```

Out[74]:
```
0      True
1     False
2     False
3     False
4     False
5      True
6     False
7     False
8      True
9     False
10    False
11    False
12    False
13    False
14    False
15    False
16    False
17    False
18    False
19    False
20    False
21    False
22    False
23    False
24    False
25    False
26    False
27    False
28    False
29    False
dtype: bool
```

Outros ainda retornam listas ou outros valores compostos para cada elemento.

In [76]:
```
turma.str.split()
```

Out[76]:
```
0         [Thiago]
1         [Renata]
2        [Luciano]
3           [Luis]
4         [Nairan]
5         [Thiago]
6        [Matheus]
7        [Rafaela]
8          [Tales]
9         [Manoel]
10        [Sérgio]
11        [Allana]
12       [RICARDO]
13       [Gabriel]
14        [Arthur]
15    [Everaldina]
16      [VINICIUS]
17      [Girleide]
18          [João]
19         [Paulo]
20      [Brenndol]
21         [Erika]
22           [Ian]
23        [Danilo]
24        [Raíssa]
25         [Leane]
26         [Vitor]
27       [Myllena]
28          [JOSE]
29        [Marcos]
dtype: object
```

Métodos usando expressões regulares

Além disso, existem vários métodos que aceitam expressões regulares para examinar o conteúdo de cada elemento de string e seguem algumas das convenções da API do módulo `re` integrado do **Python**.

Method	Description
`match()`	Chama `re.match()` em cada elemento, retornando um booleano.
`extract()`	Chama `re.match()` em cada elemento, retornando grupos correspondentes como strings.
`findall()`	Chama `re.findall()` em cada elemento
`replace()`	Substitui as ocorrências do padrão por alguma outra string
`contains()`	Chama `re.search()` em cada elemento, retornando um booleano
`count()`	Conta ocorrências de padrão
`split()`	Equivalente a `str.split()`, mas aceita regexps
`rsplit()`	Equivalente a `str.rsplit()`, mas aceita regexps

Com eles, você pode realizar uma ampla gama de operações interessantes. Por exemplo, podemos fazer algo mais complicado, como encontrar todos os nomes que começam e terminam com uma consoante, fazendo uso dos caracteres de expressão regular de início de string (^) e fim de string ($).

In [78]:
```
#turma = turma.str.capitalize()
turma.str.findall(r'^[^AEIOU].*[^aeiou]$')
```

Out[78]:
```
0             []
1             []
2             []
3         [Luis]
4       [Nairan]
5             []
6      [Matheus]
7             []
8        [Tales]
9       [Manoel]
10            []
11            []
12            []
13     [Gabriel]
14            []
15            []
16    [Vinicius]
17            []
18            []
19            []
20    [Brenndol]
21            []
22            []
23            []
24            []
25            []
26       [Vitor]
27            []
28            []
29      [Marcos]
dtype: object
```

Métodos diversos

Finalmente, existem alguns métodos variados que permitem outras operações importantes.

Métodd	O que faz
`get()`	Indexa cada elemento
`slice()`	Slice cada elemento
`slice_replace()`	Substitua a slice em cada elemento pelo valor passado
`cat()`	Concatena strings
`repeat()`	Repete valores
`normalize()`	Retorna a string em `Unicode`
`pad()`	Adicione espaços em branco à esquerda, à direita ou em ambos os lados das strings
`wrap()`	Divida strings longas em linhas com comprimento menor que uma determinada largura
`join()`	Junte strings em cada elemento da Série com separador específico
`get_dummies()`	extrair variáveis fictícias como um dataframe

Acesso e slicing de itens vetorizados

As operações `get() e slice()` , em particular, permitem o acesso a elementos vetorizados de cada array.

Por exemplo, podemos obter uma fatia dos três primeiros caracteres de cada array usando `str.slice(0, 3)` .

Observe que esse comportamento também está disponível através da sintaxe de indexação normal do **Python** – por exemplo, `df.str.slice(0, 3)` é equivalente a `df.str[0:3]` .

In [79]:
```
turma.str[0:3]
```

Out[79]:
```
0     Thi
1     Ren
2     Luc
3     Lui
4     Nai
5     Thi
6     Mat
7     Raf
8     Tal
9     Man
10    Sér
11    All
12    Ric
13    Gab
14    Art
15    Eve
16    Vin
17    Gir
18    Joã
19    Pau
20    Bre
21    Eri
22    Ian
23    Dan
24    Raí
25    Lea
26    Vit
27    Myl
28    Jos
29    Mar
dtype: object
```

In [80]:
```
turma.str.slice(0, 3)
```

Out[80]:
```
0     Thi
1     Ren
2     Luc
3     Lui
4     Nai
5     Thi
6     Mat
7     Raf
8     Tal
9     Man
10    Sér
11    All
12    Ric
13    Gab
14    Art
15    Eve
16    Vin
17    Gir
18    Joã
19    Pau
20    Bre
21    Eri
22    Ian
23    Dan
24    Raí
25    Lea
26    Vit
27    Myl
28    Jos
29    Mar
dtype: object
```

A indexação via `df.str.get(i)` e `df.str[i]` são igualmente semelhante.

Esses métodos `get()` e `slice()` também permitem acessar elementos de arrays retornados por `split()` .

In [81]:
```
turma.str.get(-1)
```

Out[81]:
```
0     o
1     a
2     o
3     s
4     n
5     o
6     s
7     a
8     s
9     l
10    o
11    a
12    o
13    l
14    r
15    a
16    s
17    e
18    o
19    o
20    l
21    a
22    n
23    o
24    a
25    e
26    r
27    a
28    e
29    s
dtype: object
```

Variáveis indicadoras

Outro método que requer um pouco de explicação extra é o método `get_dummies()`. Isso é útil quando seus dados possuem uma coluna contendo algum tipo de indicador codificado. Por exemplo, podemos ter um conjunto de dados que contém informações na forma de códigos, como A="nascido na América," B="nascido no Reino Unido," C="gosta de queijo," D="gosta de spam"

In [84]:
```
turma_plus = pd.DataFrame({'nome': turma[0:6],
                           'info': ['B|C|D', 'B|D', 'A|C',
                                    'B|D', 'B|C', 'B|C|D']})
turma_plus
```

Out[84]:

	nome	info
0	Thiago	B|C|D
1	Renata	B|D
2	Luciano	A|C
3	Luis	B|D
4	Nairan	B|C
5	Thiago	B|C|D

A rotina `get_dummies()` permite que você divida rapidamente essas variáveis indicadoras em um `DataFrame`

In [85]:
```
turma_plus['info'].str.get_dummies('|')
```

Out[85]:

	A	B	C	D
0	0	1	1	1
1	0	1	0	1
2	1	0	1	0
3	0	1	0	1
4	0	1	1	0
5	0	1	1	1

Com essas operações como blocos de construção, você pode construir uma gama infinita de procedimentos de processamento de strings para limpar seus dados.

Não vamos nos aprofundar nesses métodos aqui, mas recomendo que você leia "Trabalhando com dados de texto" (http://pandas.pydata.org/pandas-docs/stable/text.html) no Pandas documentação on-line