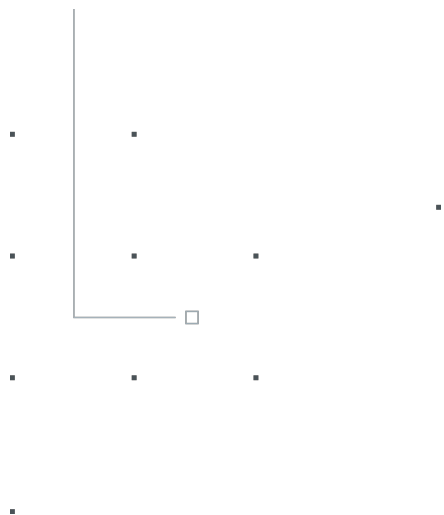


FIAP

NBA



Pandas II

Dheny R. Fernandes

1. Pandas II

1. Indexação Hierárquica
2. Reshape e Pivoting
3. Agregação de dados e operações Group

Indexação Hierárquica



Como vimos, em muitas aplicações os dados podem estar espalhados em diversos arquivos ou base de dados, ou estarem organizados de uma forma não conveniente para análise. Vimos como combinar dados e agora veremos como **transformar**, ou reorganizar, os dados.

Para entender os métodos que serão usados na transformação de dados, precisamos entender indexação hierárquica, uma importante ferramenta do pandas que permite que você tenha múltiplos níveis de índice.

Uma outra maneira de entender essa ferramenta é que ela **permite trabalhar com dados de alta dimensionalidade numa baixa dimensionalidade**.

Vamos começar com um exemplo simples no código:

A imagem abaixo mostra uma visualização de uma série com MultiIndex como seu índice. Os “espaços” no índice significam que o rótulo diretamente acima é o válido.

```
a  1    0.374540
   2    0.950714
   3    0.731994
b  1    0.598658
   3    0.156019
c  1    0.155995
   2    0.058084
d  2    0.866176
   3    0.601115
dtype: float64
```

Com o objeto hierarquicamente indexado, indexação parcial é possível, permitindo a seleção de subconjuntos de dados. Veja o código:

Indexação Hierárquica possui um importante papel em transformar os dados e em operações baseadas em Group. Por exemplo, é possível transformar os dados num DataFrame usando o método *unstack()*. A operação inversa é *stack()*:

```
a  1    0.374540
   2    0.950714
   3    0.731994
b  1    0.598658
   3    0.156019
c  1    0.155995
   2    0.058084
d  2    0.866176
   3    0.601115
dtype: float64
```

data

	1	2	3
a	0.374540	0.950714	0.731994
b	0.598658	NaN	0.156019
c	0.155995	0.058084	NaN
d	NaN	0.866176	0.601115

data.unstack()

```
a  1    0.374540
   2    0.950714
   3    0.731994
b  1    0.598658
   3    0.156019
c  1    0.155995
   2    0.058084
d  2    0.866176
   3    0.601115
dtype: float64
```

data.unstack().stack()

Com DataFrame, ambos os eixos podem ter um índice hierárquico:

		Ohio		Colorado
		Green	Red	Green
a	1	0	1	2
	2	3	4	5
b	1	6	7	8
	2	9	10	11

Os níveis hierárquicos podem ter o atributo *names* com valores preenchidos.

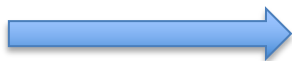
		state	Ohio		Colorado
		color	Green	Red	Green
key1	key2				
a	1		0	1	2
	2		3	4	5
b	1		6	7	8
	2		9	10	11

O método `sort_index()` funciona lexicograficamente usando todos os níveis de índice, mas é possível escolher apenas um único nível ou subconjunto de níveis para ordenação usando o parâmetro `level`.

state		Ohio		Colorado
color		Green	Red	Green
key1	key2			
a	1	0	1	2
b	1	6	7	8
a	2	3	4	5
b	2	9	10	11

É possível usar uma ou mais colunas de um DataFrame como índice através do método `set_index()`:

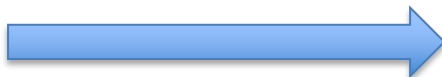
	a	b	c	d
0	0	7	one	0
1	1	6	one	1
2	2	5	one	2
3	3	4	two	0
4	4	3	two	1
5	5	2	two	2
6	6	1	two	3



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

O método *reset_index()*, por outro lado, faz o oposto de *set_index()*, ou seja, os índices hierárquicos são transformados novamente em colunas:

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



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

Reshape e Pivoting



A indexação hierárquica prove uma maneira consistente de reorganizar os dados num DataFrame. Existem dois métodos que usamos costumeiramente:

- *Stack*: rotaciona as colunas para linhas
- *Unstack*: rotaciona as linhas para colunas

Para entender esses processos, vamos usar alguns exemplos. Considere o seguinte DataFrame:

number	one	two	three
state			
Ohio	0	1	2
Colorado	3	4	5

Usar o método *stack()* nesses dados ira rotacionar (pivotar) as colunas em linhas, produzindo uma Series:

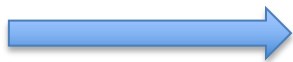
```
state    number
Ohio     one      0
         two      1
         three    2
Colorado one      3
         two      4
         three    5
dtype: int64
```

A partir de uma Series hierarquicamente indexada, podemos reorganizar os dados de volta num DataFrame usando *unstack()*:

number	one	two	three
state			
Ohio	0	1	2
Colorado	3	4	5

O método *unstack()* pode introduzir NaN se todos os valores num nível não forem encontrados em cada subgrupo. Observe:

```
one  a    0
     b    1
     c    2
     d    3
two  c    4
     d    5
     e    6
dtype: Int64
```



	a	b	c	d	e
one	0	1	2	3	<NA>
two	<NA>	<NA>	4	5	6

Stack() filtra NaN por padrão, assim a operação é facilmente invertível.

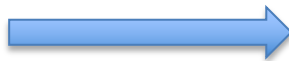
```
one  a    0
     b    1
     c    2
     d    3
two  c    4
     d    5
     e    6
dtype: Int64
```

```
one  a    0
     b    1
     c    2
     d    3
     e   <NA>
two  a   <NA>
     b   <NA>
     c    4
     d    5
     e    6
dtype: Int64
```

Dropna = False

Quando aplicado num DataFrame, o nível a ser desempilhado se torna o menor nível no resultado.

		side	left	right
state	number			
Ohio	one		0	5
	two		1	6
	three		2	7
Colorado	one		3	8
	two		4	9
	three		5	10



		side	left		right
state	number	Ohio	Colorado	Ohio	Colorado
	one	0	3	5	8
	two	1	4	6	9
	three	2	5	7	10

Quando chamamos *stack()*, podemos indicar o nome do eixo a empilhar:

		state	Colorado	Ohio
number	side			
one	left		3	0
	right		8	5
two	left		4	1
	right		9	6
three	left		5	2
	right		10	7

Uma maneira comum de armazenar múltiplas séries temporais em base de dados e arquivos CSV é o que chamamos de *formato longo ou empilhado*. Neste formato, valores individuais são representados por uma única linha na tabela ao invés de múltiplos valores por linha.

Vamos fazer a leitura de alguns dados e realizar algumas transformações neles:

	year	quarter	realgdp	infl	unemp
0	1959	1	2710.349	0.00	5.8
1	1959	2	2778.801	2.34	5.1
2	1959	3	2775.488	2.74	5.3
3	1959	4	2785.204	0.27	5.6
4	1960	1	2847.699	2.31	5.2

Vamos usar o método *PeriodIndex()* do Pandas para combinar as colunas ano e trimestre e usar como índice que consiste em valores *datetime* no fim de cada trimestre:

	realgdp	infl	unemp
date			
1959-01-01	2710.349	0.00	5.8
1959-04-01	2778.801	2.34	5.1
1959-07-01	2775.488	2.74	5.3
1959-10-01	2785.204	0.27	5.6
1960-01-01	2847.699	2.31	5.2

O método *pop()* retorna a coluna ao mesmo tempo que deleta ela do DataFrame. Então, selecionamos um subconjunto de colunas e fornecemos o nome do índice de “item”. Por fim, os dados são reorganizados com o método *stack()* , e obtemos o seguinte:

	date	item	value
0	1959-01-01	realgdp	2710.349
1	1959-01-01	infl	0.000
2	1959-01-01	unemp	5.800
3	1959-04-01	realgdp	2778.801
4	1959-04-01	infl	2.340

Este é o formato conhecido como **longo ou empilhado**, em que cada linha na tabela representa uma única amostra.

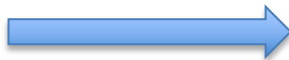
Os dados são frequentemente armazenados dessa maneira em banco de dados relacionais. No exemplo anterior, *date* e *item* seriam usualmente chaves primárias, oferecendo integridade relacional e facilidade de *joins*.

Entretanto, em alguns casos, os dados serão mais difíceis de lidar nesse formato. Pode ser que seja preferível um DataFrame que contém uma coluna por valor distinto de *item* indexado por timestamps na coluna *date*.

O método ***pivot()*** do Pandas realiza exatamente esse tipo de transformação. Veja o código:

A operação inversa do método *pivot()* para DataFrames é o *melt()*. Em vez de transformar uma coluna em várias em um novo DataFrame, ele mescla várias colunas em uma, produzindo um DataFrame maior que o de entrada. Vejamos um exemplo no código:

	key	A	B	C
0	foo	1	4	7
1	bar	2	5	8
2	baz	3	6	9



	key	variable	value
0	foo	A	1
1	bar	A	2
2	baz	A	3
3	foo	B	4
4	bar	B	5
5	baz	B	6
6	foo	C	7
7	bar	C	8
8	baz	C	9

Usando o método *pivot()*, podemos retornar ao arranjo original do DataFrame:

variable	A	B	C	
key				
bar	2	5	8	
baz	3	6	9	
foo	1	4	7	

Visto que o resultado do *pivot()* cria um índice a partir da coluna usada como rótulo das linhas, podemos usar o método *reset_index()* para mover os dados para uma coluna:

variable	key	A	B	C
0	bar	2	5	8
1	baz	3	6	9
2	foo	1	4	7

É possível especificar um subconjunto de colunas a serem usadas como valores de colunas:

	key	variable	value
0	foo	A	1
1	bar	A	2
2	baz	A	3
3	foo	B	4
4	bar	B	5
5	baz	B	6

E é possível usar o *melt()* sem qualquer identificador de grupo:

	variable	value
0	A	1
1	A	2
2	A	3
3	B	4
4	B	5
5	B	6
6	C	7
7	C	8
8	C	9

Para facilitar o entendimento, considere um exemplo prático. Suponha que eu tenha um DataFrame que contém o quanto cada uma das minhas lojas venderam num determinado dia.

	Nome	10/05/2023	10/06/2023	10/07/2023	10/08/2023
0	Loja1	141	20	95	183
1	Loja2	98	101	103	19
2	Loja3	69	64	122	186
3	Loja4	23	141	199	46

Entretanto, pode ser difícil fazer análise assim, principalmente se quero trabalhar com séries temporais. Assim, o método *melt()* permite um rearranjo dos dados:

	Nome	Data	Vendas_Total
0	Loja1	10/05/2023	141
1	Loja2	10/05/2023	98
2	Loja3	10/05/2023	69
3	Loja4	10/05/2023	23
4	Loja1	10/06/2023	20
5	Loja2	10/06/2023	101
6	Loja3	10/06/2023	64
7	Loja4	10/06/2023	141
8	Loja1	10/07/2023	95
9	Loja2	10/07/2023	103
10	Loja3	10/07/2023	122
11	Loja4	10/07/2023	199
12	Loja1	10/08/2023	183
13	Loja2	10/08/2023	19
14	Loja3	10/08/2023	186
15	Loja4	10/08/2023	46

Agregação de dados e operações Group

Agrupar um conjunto de dados e aplicar uma função a cada grupo, seja uma agregação ou transformação, pode ser um componente crítico no *workflow* de análise de dados. Depois de carregar, combinar e preparar o conjunto de dados, você pode querer calcular estatísticas ou até rotacionar tabelas para propósitos de relatórios ou visualização.

O Pandas oferece um versátil conjunto de operações *groupby* que podem ser aplicadas nesses conjuntos de dados.

Uma razão para a popularidade de bancos de dados relacionais e SQL é a facilidade com que os dados podem ser unidos, filtrados, transformados e agregados. No entanto, linguagens como SQL impõem certas limitações nos tipos de operações de grupo que podem ser executadas. Entretanto, a combinação de Python e Pandas vem pra facilitar nosso trabalho.

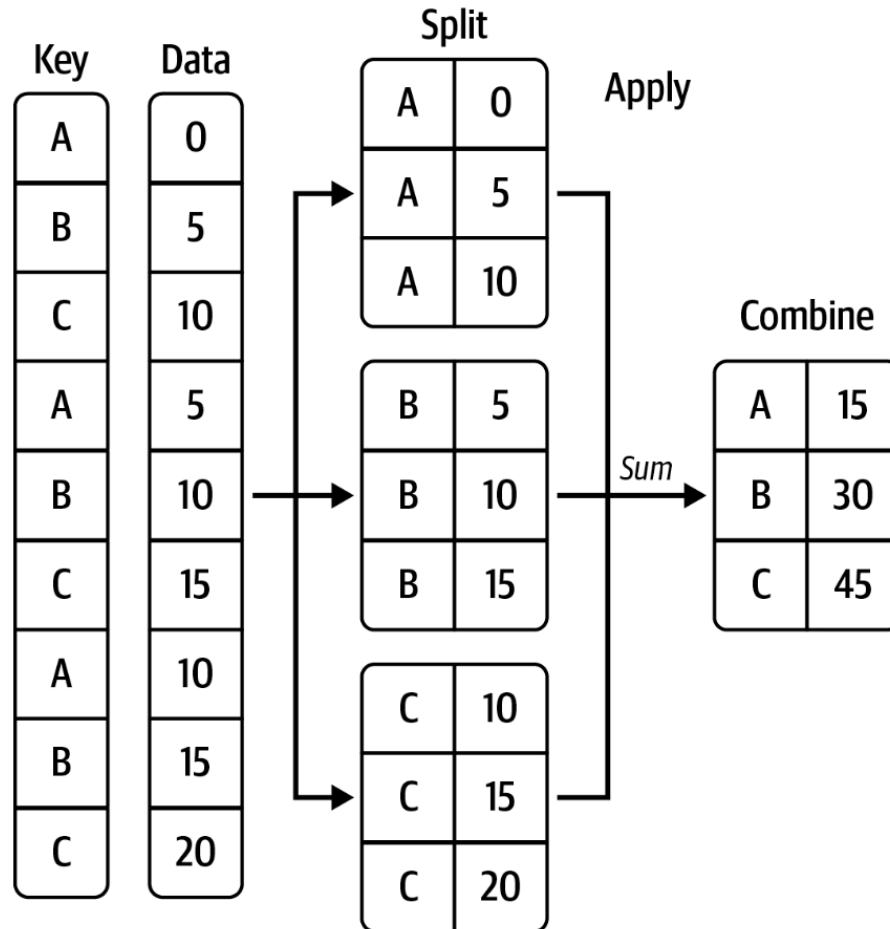
Aqui, vamos focar nos seguintes pontos:

- Dividir um objeto Pandas usando uma ou mais chaves
- Calcular um resumo estatístico de grupo, como contagem, média ou desvio padrão, além de possibilitar uma função definida pelo próprio usuário
- Aplicar transformações nos grupos
- *Pivot tables e cross-tabulations*
- Executar análises estatísticas de grupos

Primeiramente, é importante saber como pensar sobre operações em grupo. Hadley Wickham, autor de muitos pacotes do R, cunhou o termo *split-apply-combine* para descrever operações em grupo.

No primeiro estágio do processo, *os dados são divididos em grupos* baseados em uma ou mais chaves providenciadas. Feito isso, *uma função é aplicada em cada grupo*, produzindo um novo valor. Finalmente, *os resultados de todas essas aplicações de funções são combinados no objeto resultante*.

A figura a seguir ilustra esse processo:



Para começar, considere o seguinte DataFrame:

	key1	key2	data1	data2
0	a	1	-0.229450	0.881761
1	a	2	0.389349	-1.009085
2	None	1	-1.265119	-1.583294
3	b	2	1.091992	0.773700
4	b	1	2.778313	-0.538142
5	a	<NA>	1.193640	-1.346678
6	None	1	0.218638	-0.880591

Suponha que você queira calcular a média da coluna *data1* usando os rótulos de *key1*. Podemos fazer isso usando método *groupby()*:

```
key1
a    0.451179
b    1.935153
Name: data1, dtype: float64
```

Importante notar que a variável *grouped* é um objeto GroupBy. Ainda não houve nenhum cálculo exceto para alguns dados intermediários sobre o agrupamento a partir da chave *key1*. A ideia é que este objeto possui todas as informações necessárias para então aplicar alguma operação em cada grupo.

Podemos passar múltiplos argumentos como uma lista para criar os grupos. Veja o exemplo:

```
key1  key2
a      1    -0.229450
      2     0.389349
b      1     2.778313
      2     1.091992
Name: data1, dtype: float64
```

Além da média, um outro método comumente utilizado em operações Group é o *size()*, que retorna uma Series contendo o tamanho de cada grupo:

```
key1  key2
a      1      1
      2      1
b      1      1
      2      1
dtype: int64
```

Por padrão, qualquer valor faltante numa chave de grupo é excluído do resultado final. Podemos circundar esse comportamento usando o parâmetro *dropna=False*.

```
key1
a      3
b      2
NaN     2
dtype: int64
```


O objeto retornado pelo *groupby* suporta iteração, gerando uma sequência de 2 tuplas contendo o nome do grupo e os dados do grupo.

```
a
  key1  key2    data1    data2
0     a     1 -0.229450  0.881761
1     a     2  0.389349 -1.009085
5     a  <NA>  1.193640 -1.346678
b
  key1  key2    data1    data2
3     b     2  1.091992  0.773700
4     b     1  2.778313 -0.538142
```

No caso de múltiplas chaves, o primeiro elemento na tupla será uma tupla de valores de chaves.

```
('a', 1)
  key1  key2    data1    data2
0     a     1 -0.22945  0.881761
('a', 2)
  key1  key2    data1    data2
1     a     2  0.389349 -1.009085
('b', 1)
  key1  key2    data1    data2
4     b     1  2.778313 -0.538142
('b', 2)
  key1  key2    data1    data2
3     b     2  1.091992  0.7737
```

É possível criar um dicionário a partir de cada pedaço de dados usando dict comprehension:

	key1	key2	data1	data2
3	b	2	1.091992	0.773700
4	b	1	2.778313	-0.538142

Indexar um objeto GroupBy criado a partir de um DataFrame com uma coluna ou um array de colunas possui o efeito de selecionar um subconjunto. Isto significa que:

`df.groupby("key1")["data1"]` é uma conveniência para `df["data1"].groupby(df["key1"])`

Especialmente para grandes datasets, pode ser desejável agregar apenas algumas colunas. Por exemplo, no dataset visto anteriormente, para calcular a média apenas para a coluna *data2* e obter o resultado como um DataFrame, poderíamos escrever:

```
df.groupby(["key1", "key2"])["data2"].mean()
```

		data2
key1	key2	
a	1	0.881761
	2	-1.009085
b	1	-0.538142
	2	0.773700

Por padrão, a informação de agrupamento vem na forma de um array, mas é possível usar outra forma, como, por exemplo, um dicionário. Veja:

	blue	red
Joe	1.469871	-0.101791
Steve	-2.131631	1.647479
Wanda	0.281508	0.246680
Jill	0.897160	0.054327
Trey	2.451905	1.434977

Agregação se refere a qualquer transformação nos dados que produz um escalar a partir de arrays. Alguns exemplos anteriores usaram várias delas, como *mean*, *count*, *sum*. Abaixo, segue uma lista não exaustiva de funções para agregação:

Function name	Description
<code>any</code> , <code>all</code>	Return True if any (one or more values) or all non-NA values are “truthy”
<code>count</code>	Number of non-NA values
<code>cummin</code> , <code>cummax</code>	Cumulative minimum and maximum of non-NA values
<code>cumsum</code>	Cumulative sum of non-NA values
<code>cumprod</code>	Cumulative product of non-NA values
<code>first</code> , <code>last</code>	First and last non-NA values
<code>mean</code>	Mean of non-NA values
<code>median</code>	Arithmetic median of non-NA values
<code>min</code> , <code>max</code>	Minimum and maximum of non-NA values
<code>nth</code>	Retrieve value that would appear at position <code>n</code> with the data in sorted order
<code>ohlc</code>	Compute four “open-high-low-close” statistics for time series-like data
<code>prod</code>	Product of non-NA values
<code>quantile</code>	Compute sample quantile
<code>rank</code>	Ordinal ranks of non-NA values, like calling <code>Series.rank</code>
<code>size</code>	Compute group sizes, returning result as a Series
<code>sum</code>	Sum of non-NA values
<code>std</code> , <code>var</code>	Sample standard deviation and variance

É possível usar agregações e qualquer método que é definido no objeto a ser agrupado. Por exemplo, o método *nsmallest* da Series seleciona os n menores valores dos dados. Mesmo não sendo um método explicitamente implementado para o GroupBy, ainda é possível de ser utilizado, mas com uma implementação não otimizada. Veja o código:

	key1	key2	data1	data2
0	a	1	-0.229450	0.881761
1	a	2	0.389349	-1.009085
2	None	1	-1.265119	-1.583294
3	b	2	1.091992	0.773700
4	b	1	2.778313	-0.538142
5	a	<NA>	1.193640	-1.346678
6	None	1	0.218638	-0.880591



```
key1
a    0    -0.229450
    1     0.389349
b    3     1.091992
    4     2.778313
Name: data1, dtype: float64
```

Para usar sua própria função de agregação, passe como parâmetro qualquer função que agregue um array para o método *agg()*:

```
def peak_to_peak(arr):  
    return arr.max() - arr.min()  
grouped.agg(peak_to_peak)
```

key2 data1 data2

key1

Como visto, agregar uma Series ou todas as colunas de um DataFrame é questão de usar o método *agg()* com a função desejada ou chamar algum método existente, como *mean()* ou *std()*. Entretanto, pode ser que seja desejável usar funções diferentes para colunas diferentes, ou então múltiplas funções de uma única vez.

Vamos usar o dataset tips para entender o processo:

	total_bill	tip	smoker	day	time	size	tip_pct
0	16.99	1.01	No	Sun	Dinner	2	0.059447
1	10.34	1.66	No	Sun	Dinner	3	0.160542
2	21.01	3.50	No	Sun	Dinner	3	0.166587
3	23.68	3.31	No	Sun	Dinner	2	0.139780
4	24.59	3.61	No	Sun	Dinner	4	0.146808

Realizei o agrupamento por *day* e *smoker*. Com o agrupamento feito, podemos aplicar qualquer função de agregação e se passar uma lista de funções, você recebe como retorno um DataFrame com os nomes de colunas obtidos das funções:

		mean	std	peak_to_peak
day	smoker			
Fri	No	0.151650	0.028123	0.067349
	Yes	0.174783	0.051293	0.159925
Sat	No	0.158048	0.039767	0.235193
	Yes	0.147906	0.061375	0.290095
Sun	No	0.160113	0.042347	0.193226
	Yes	0.187250	0.154134	0.644685
Thur	No	0.160298	0.038774	0.193350
	Yes	0.163863	0.039389	0.151240

É possível alterar o nome das funções quando for apresentar o DataFrame resultante. Para isso, passe como parâmetro uma lista de tuplas na forma (nome, função). Veja abaixo:

		tip_pct		total_bill	
		Average	Variance	Average	Variance
day	smoker				
Fri	No	0.151650	0.000791	18.420000	25.596333
	Yes	0.174783	0.002631	16.813333	82.562438
Sat	No	0.158048	0.001581	19.661778	79.908965
	Yes	0.147906	0.003767	21.276667	101.387535
Sun	No	0.160113	0.001793	20.506667	66.099980
	Yes	0.187250	0.023757	24.120000	109.046044
Thur	No	0.160298	0.001503	17.113111	59.625081
	Yes	0.163863	0.001551	19.190588	69.808518

Agora suponha que você queira aplicar diferentes funções para uma ou mais colunas. Para fazer isso, passe um dicionário para o método *agg()* contem um mapeamento dos nomes das colunas para qualquer função especificada numa lista:

					tip_pct	size
		min	max	mean	std	sum
day	smoker					
Fri	No	0.120385	0.187735	0.151650	0.028123	9
	Yes	0.103555	0.263480	0.174783	0.051293	31
Sat	No	0.056797	0.291990	0.158048	0.039767	115
	Yes	0.035638	0.325733	0.147906	0.061375	104
Sun	No	0.059447	0.252672	0.160113	0.042347	167
	Yes	0.065660	0.710345	0.187250	0.154134	49
Thur	No	0.072961	0.266312	0.160298	0.038774	112
	Yes	0.090014	0.241255	0.163863	0.039389	40

Como vimos, o `apply` divide o objeto a ser manipulado em pedaços, chama a função passada em cada um deles e concatena cada pedaço.

Para entendermos seu correto funcionamento, vamos retornar ao dataset *tips*. Suponha que você precise selecionar os top cinco valores por grupo. Primeiro, escreva uma função que seleciona as linhas com os maiores valores numa coluna em particular. Depois, agrupe os dados por *smoker* e chame o `apply` passando essa função como parâmetro. Obteremos o seguinte:

		total_bill	tip	smoker	day	time	size	tip_pct
smoker								
No	232	11.61	3.39	No	Sat	Dinner	2	0.291990
	149	7.51	2.00	No	Thur	Lunch	2	0.266312
	51	10.29	2.60	No	Sun	Dinner	2	0.252672
	185	20.69	5.00	No	Sun	Dinner	5	0.241663
	88	24.71	5.85	No	Thur	Lunch	2	0.236746
Yes	172	7.25	5.15	Yes	Sun	Dinner	2	0.710345
	178	9.60	4.00	Yes	Sun	Dinner	2	0.416667
	67	3.07	1.00	Yes	Sat	Dinner	1	0.325733
	183	23.17	6.50	Yes	Sun	Dinner	4	0.280535
	109	14.31	4.00	Yes	Sat	Dinner	2	0.279525

O que aconteceu aqui? Primeiro, o DataFrame foi dividido em grupos baseado no valor de *smoke*. Então, a função *top* foi chamada em cada grupo e o resultado de cada chamada foi concatenado (usando `pandas.concat`), rotulando cada pedaço com o nome do grupo. O resultado, portanto, contém um índice hierárquico com um nível interno que contém valores de índice do DataFrame original.

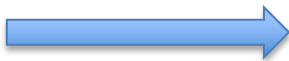
Se você passar uma função ao *apply* que recebe outros argumentos, você pode passá-los depois da função:

```
tips.groupby(["smoker", "day"]).apply(top, n=1, column="total_bill")
```

			total_bill	tip	smoker	day	time	size	tip_pct
smoker	day								
No	Fri	94	22.75	3.25	No	Fri	Dinner	2	0.142857
	Sat	212	48.33	9.00	No	Sat	Dinner	4	0.186220
	Sun	156	48.17	5.00	No	Sun	Dinner	6	0.103799
	Thur	142	41.19	5.00	No	Thur	Lunch	5	0.121389
Yes	Fri	95	40.17	4.73	Yes	Fri	Dinner	4	0.117750
	Sat	170	50.81	10.00	Yes	Sat	Dinner	3	0.196812
	Sun	182	45.35	3.50	Yes	Sun	Dinner	3	0.077178
	Thur	197	43.11	5.00	Yes	Thur	Lunch	4	0.115982

O Pandas possui algumas funções, como *cut()* e *qcut()*, para dividir os dados em compartimentos ou quantis. Tais funções podem ser combinadas com *groupby*, o que torna muito conveniente análises estatísticas no dataset. Considere o seguinte exemplo:

	data1	data2
0	-0.077372	-0.676671
1	0.015019	1.004593
2	0.538062	0.096828
3	-0.041681	-0.503847
4	2.017908	0.344384



```
0    (-0.205, 1.429]
1    (-0.205, 1.429]
2    (-0.205, 1.429]
3    (-0.205, 1.429]
4    (1.429, 3.062]
5    (-0.205, 1.429]
6    (-0.205, 1.429]
7    (-1.838, -0.205]
Name: data1, dtype: category
Categories (4, interval[float64, right]): [(-3.478, -1.838] < (-1.838, -0.205] < (-0.205, 1.429] < (1.429, 3.062]]
```

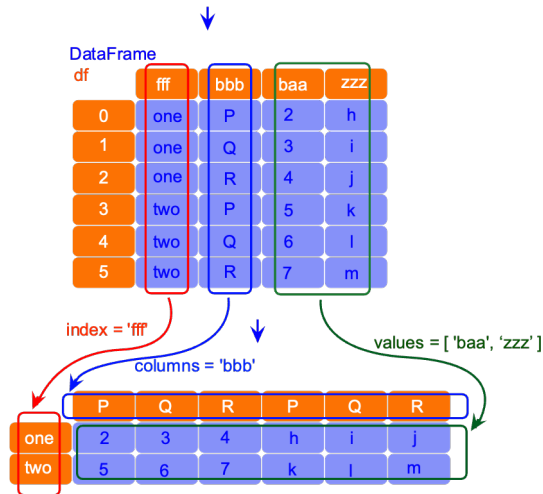
O objeto categórico retornado pelo `cut()` pode ser passado diretamente para o `groupby`, de maneira que podemos calcular um conjunto de estatísticas para os quantis:

		min	max	count	mean
data1					
(-3.478, -1.838]	data1	-3.471430	-1.850632	28	-2.256595
	data2	-1.985461	2.441504	28	-0.070586
(-1.838, -0.205]	data1	-1.835711	-0.205728	403	-0.827385
	data2	-2.722278	2.501645	403	0.033773
(-0.205, 1.429]	data1	-0.200749	1.417020	502	0.504066
	data2	-2.899894	2.765980	502	0.025218
(1.429, 3.062]	data1	1.429117	3.061947	67	1.787994
	data2	-1.932167	2.103673	67	-0.135325

Pivot table é uma ferramenta de sumarização de dados frequentemente encontrada em softwares de análises e de planilhas. Ela agrega uma tabela de dados por uma ou mais colunas, organizando os dados num retângulo com algumas das chaves de grupo nas linhas e outras nas colunas.

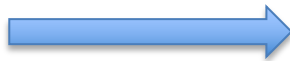
Em Python, *pivot tables* são feitas em conjunto com o Pandas através das operações *groupby* combinadas com operações de transformação usando indexação hierárquica. Além disso, Pandas possui o método *pivot_table()*

```
df.pivot ( index = 'fff' , columns = 'bbb' , values = [ 'baa' , 'zzz' ] )
```



Retornando ao dataset *tips*, suponha que você queira criar uma tabela de um grupo de médias (a agregação padrão do *pivot_table*) organizado por *day* e *smoker* nas linhas:

	total_bill	tip	smoker	day	time	size	tip_pct
0	16.99	1.01	No	Sun	Dinner	2	0.059447
1	10.34	1.66	No	Sun	Dinner	3	0.160542
2	21.01	3.50	No	Sun	Dinner	3	0.166587
3	23.68	3.31	No	Sun	Dinner	2	0.139780
4	24.59	3.61	No	Sun	Dinner	4	0.146808



		size	tip	tip_pct	total_bill
day	smoker				
Fri	No	2.250000	2.812500	0.151650	18.420000
	Yes	2.066667	2.714000	0.174783	16.813333
Sat	No	2.555556	3.102889	0.158048	19.661778
	Yes	2.476190	2.875476	0.147906	21.276667
Sun	No	2.929825	3.167895	0.160113	20.506667
	Yes	2.578947	3.516842	0.187250	24.120000
Thur	No	2.488889	2.673778	0.160298	17.113111
	Yes	2.352941	3.030000	0.163863	19.190588

Entretanto, isto poderia ser produzido diretamente com um `groupby`, usando `tips.groupby(['day', 'smoker']).mean()`

Suponha agora que você queira a média apenas de *tip_pct* e *size* e adicionalmente agrupar por *time*.

		size		tip_pct	
		No	Yes	No	Yes
time	day				
Dinner	Fri	2.000000	2.222222	0.139622	0.165347
	Sat	2.555556	2.476190	0.158048	0.147906
	Sun	2.929825	2.578947	0.160113	0.187250
	Thur	2.000000	NaN	0.159744	NaN
Lunch	Fri	3.000000	1.833333	0.187735	0.188937
	Thur	2.500000	2.352941	0.160311	0.163863

Para usar outra função de agregação além de *mean*, passa o nome dela como valor para o argumento *aggfunc*. Veja o exemplo:

```
tips.pivot_table(index=["time", "smoker"], columns="day",  
                  values="tip_pct", aggfunc=len)
```

		day	Fri	Sat	Sun	Thur
time	smoker					
Dinner	No	3.0	45.0	57.0	1.0	
	Yes	9.0	42.0	19.0	NaN	
Lunch	No	1.0	NaN	NaN	44.0	
	Yes	6.0	NaN	NaN	17.0	

Para preencher os valores faltantes, use o argumento *fill-value*:

```
tips.pivot_table(index=["time", "smoker"], columns="day",  
                  values="tip_pct", aggfunc=len, fill_value=0)
```

		day	Fri	Sat	Sun	Thur
time	smoker					
Dinner	No	3	45	57	1	
	Yes	9	42	19	0	
Lunch	No	1	0	0	44	
	Yes	6	0	0	17	

Cross-tabulation ou crosstab é um caso especial do método *pivot_table* que calcula frequência de grupo. Considere os seguintes dados como exemplo:

	Sample	Nationality	Handedness
0	1	USA	Right-handed
1	2	Japan	Left-handed
2	3	USA	Right-handed
3	4	Japan	Right-handed
4	5	Japan	Left-handed

Como parte da análise desses dados, pode ser que queiramos sumarizar os dados por nacionalidade de lateralidade. É possível usar *pivot_table*, mas o *pandas.crosstab* é mais conveniente.

Handedness	Left-handed	Right-handed	All
Nationality			
Japan	2	3	5
USA	1	4	5
All	3	7	10

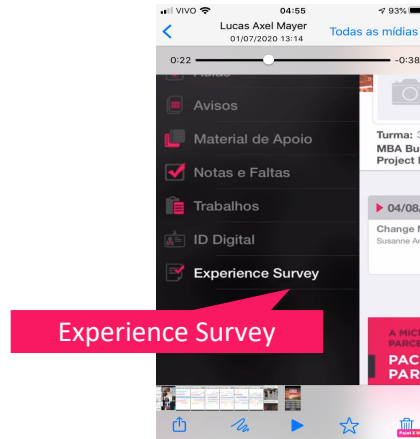
Os dois primeiros argumentos do *crosstab* podem ser um array, Series ou lista de array, como no exemplo abaixo:

		smoker	No	Yes	All
time	day				
Dinner	Fri	3	9	12	
	Sat	45	42	87	
	Sun	57	19	76	
	Thur	1	0	1	
Lunch	Fri	1	6	7	
	Thur	44	17	61	
All		151	93	244	

Resolver o exercício no notebook

O que você achou da aula de hoje?


Entrar no aplicativo FIAPP, e no menu clicar em Experience Survey



Ou pelo link: <https://fiap.me/Pesquisa-MBA>

Obrigado!

profdheny.fernandes@fiap.com.br

 /dhenyfernandes

FIAP MBA⁺

Copyright © 2018 | Professor Dheny R. Fernandes

Todos os direitos reservados. Reprodução ou divulgação total ou parcial deste documento, é expressamente proibido sem consentimento formal, por escrito, do professor/autor.

FIAP