Módulo de Programação Python

Trilha Python - Aula 14: Utilizando Pandas - Introdução

Objetivo: Trabalhar com pacotes e módulos disponíveis em **Python**: **Pandas**: Apresentar os recursos e funcionalidades básicas para operar com tabelas de dados em **Pandas**.

Tratando dados ausentes

Frequentemente são utilizados exemplos simples em que os dados são limpos e compostos por valores homogêneos. Este contexto não é facilmente encontrado quando se trata de dados extraídos do mundo real.

Resulta relativamente comum encontrar conjuntos de dados com dados faltando. A informação de que determinado dado está faltando pode ser apresentada de diferentes maneiras, dependendo da fonte de onde os dados são obtidos.

Existem várias estratégias que foram desenvolvidos para indicar a presença de dados ausentes em uma tabela ou `DataFrame`. As mais usadas se baseiam no uso de uma *máscara*, que indica globalmente valores ausentes, ou utilizam um *sentinela*, que identifica uma entrada ausente num conjunto de dados.

A abordagem de mascaramento pode utilizar, por exemplo, uma matriz *booleana* totalmente separada. Já na abordagem sentinela, o valor sentinela pode ser alguma convenção específica de dados, como indicar um valor inteiro ausente com -9999 ou pode ser uma convenção mais global, como indicar um valor de ponto flutuante ausente. com **NaN**, um valor especial que faz parte da especificação de ponto flutuante **IEEE**.

Nenhuma dessas abordagens é isenta de restrições: o uso de uma matriz de máscara separada requer a alocação de uma matriz *booleana* adicional, o que adiciona sobrecarga tanto no armazenamento quanto na computação. Um valor sentinela reduz o intervalo de valores válidos que podem ser representados e pode exigir lógica extra na aritmética da **CPU** e **GPU**. Valores especiais comuns como **NaN** não estão disponíveis para todos os tipos de dados.

Como na maioria dos casos em que não existe uma escolha universalmente ideal, diferentes linguagens e sistemas utilizam convenções diferentes.

A maneira como o **Pandas** lida com valores ausentes é limitada por sua dependência do pacote **NumPy**, que não possui uma noção integrada de valores ausentes para tipos de dados que não sejam de ponto flutuante.

NumPy tem suporte para matrizes mascaradas – ou seja, matrizes que possuem uma matriz de máscara *booleana* separada anexada para marcar os dados como “bons” ou “ruins”.

O **Pandas** poderia ter adotado uma estratégia baseada no uso deste recurso, mas a sobrecarga em armazenamento, computação e manutenção de código torna essa escolha pouco atraente.

Com essas restrições em mente, os desenvolvedores de **Pandas** optaram por usar sentinelas para dados ausentes e ainda optou por usar dois valores nulos do **Python** já existentes: o valor especial de ponto flutuante `NaN` e o objeto `None` do Python.

Utilizando `None`

O primeiro valor sentinela usado pelo **Pandas** é `None`, um objeto **Python** que é frequentemente usado para dados ausentes. Por ser um objeto Python, `None` não pode ser usado em nenhum array **NumPy**/**Pandas** arbitrário, mas apenas em arrays com tipo de dados `'object'` (ou seja, arrays de objetos **Python**):

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

```
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.
```

In [2]:
```
angulos = np.array([3.193, 3.473, 3.089, 5.811, 3.001])
print(angulos)
print(angulos.dtype)

angulos = np.array([3.193, None, 3.473, 3.089, None, 5.811, 3.001])
print(angulos)
print(angulos.dtype)
```

```
[3.193 3.473 3.089 5.811 3.001]
float64
[3.193 None 3.473 3.089 None 5.811 3.001]
object
```

Este `dtype=object` significa que a melhor representação de tipo comum que **NumPy** pode inferir, com base no conteúdo do *ndarray*, é que eles são objetos **Python**.

Embora esse tipo de array de objetos seja útil para alguns propósitos, quaisquer operações nos dados serão feitas no nível **Python**, com muito mais sobrecarga do que as operações normalmente rápidas vistas em arrays com tipos nativos.

In [3]:
```
for dtype in ['object', 'int', 'float']:
    print("dtype =", dtype)
    %timeit np.arange(1E6, dtype=dtype).sum()
    print()
```

```
dtype = object
36.3 ms ± 302 µs per loop (mean ± std. dev. of 7 runs, 10 loops ea
ch)

dtype = int
296 µs ± 2.83 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops
each)

dtype = float
519 µs ± 1.92 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops
each)
```

O uso de objetos **Python** em um array também significa que se você realizar operações de agregações como `sum()` ou `min()` em um array com valores `None`, você geralmente receberá um erro dado que a adição entre um número inteiro ou um ponto flutuante, por exemplo, e `None` não está definida.

Utilizando `NaN`

A outra representação de dados faltantes, `NaN`, é diferente. Trata-se de um valor especial de ponto flutuante reconhecido por todos os sistemas que usam a representação de ponto flutuante padrão **IEEE**.

In [4]:
```
angulos = np.array([3.193, np.nan, 3.473, 3.089, np.nan, 5.811, 3.0
print(angulos)
print(angulos.dtype)
```

```
[3.193   nan 3.473 3.089   nan 5.811 3.001]
float64
```

Reparem que op tipo do *ndarray* é, como esperado, `float64`, mesmo se os tipos presentes forem, por exemplo, inteiros. Desta forma as operações eficientes com *ndarrays*, disponíveis em **NumPy**, estarão disponíveis, ao contrário de quando utilizamos `None`. Claro que temos o custo de que, desta forma, tudo vira `float64` com suas vantagen e desvantagens.

In [5]:
```
angulos = np.array([30, np.nan, 45, np.nan, 60, 90, 120])
print(angulos)
print(angulos.dtype)
```

```
[ 30.  nan  45.  nan  60.  90. 120.]
float64
```

Outro fator a levar em consideração é que quando um dos operandos é `NaN`, independente da operação aritmética que for utilizada, o resultado será outro `NaN`.

In [6]:
```
incremento = np.ones_like(angulos)
print(angulos + incremento)
```

```
[ 31.  nan  46.  nan  61.  91. 121.]
```

Este comportamento também afeta as operações de agregação.

In [7]:
```
total = np.sum(angulos)
total
```

Out[7]:
```
nan
```

Entretanto é importante lembrar que **NumPy** fornece uma versão destas operações que permite lidar com dados `NaN`.

In [8]:
```
total = np.nansum(angulos)
total
```

Out[8]:
```
345.0
```

Utilizando `NaN` e `None` em Pandas

Pandas foi desenvolvido para lidar com `NaN` e `None` de forma quase intercambiável, convertendo entre eles quando necessário.

In [9]:
```
angulos = pd.Series([30, np.nan, 45, None, 60, 90, 120])
angulos
```

Out[9]:
```
0     30.0
1      NaN
2     45.0
3      NaN
4     60.0
5     90.0
6    120.0
dtype: float64
```

Veja que, novamente, o **Pandas** lidou com uma lista de valores inteiros contendo valores ausentes, convertendo num objeto `Series de float64` .
Mesmo para objetos já definidos com um tipo que não possui um valor sentinela disponível, o **Pandas** converte automaticamente o tipo quando valores ausentes são introduzidos.

In [10]:
```
angulos = pd.Series([30, 0, 45, 60, 90, 120])
angulos
```

Out[10]:
```
0     30
1      0
2     45
3     60
4     90
5    120
dtype: int64
```

In [11]:
```
angulos[1] = None
angulos
```

Out[11]:
```
0     30.0
1      NaN
2     45.0
3     60.0
4     90.0
5    120.0
dtype: float64
```

A conversão automática de tipos segue as seguintes regras simples.

Tipo	Conversão ao armazenar dados ausentes	Valor Sentinela
`floating`	Não muda	`np.nan`
`object`	Não muda	`None` ou `np.nan`
`integer`	Cast para `float64`	`np.nan`
`boolean`	Cast para `object`	`None` ou `np.nan`

Operando em valores ausentes

Como vimos, o **Pandas** trata `None` e `NaN` como intercambiáveis para indicar valores nulos ou ausentes. Para facilitar esta convenção, existem vários métodos úteis para detectar, remover e substituir valores nulos das estruturas de dados do **Pandas**.

Detectando valores ausentes

As estruturas de dados **Pandas** têm dois métodos úteis para detectar dados nulos:

- `isnull()` : Gera uma máscara *booleana* indicando valores faltantes;
- `notnull()` : Oposto de `isnull()` .

In [12]:
```
angulos = pd.Series([30, np.nan, 45, None, 60, 90, 120])
angulos
```

Out[12]:
```
0     30.0
1      NaN
2     45.0
3      NaN
4     60.0
5     90.0
6    120.0
dtype: float64
```

In [13]:
```
angulos.isnull()
```

Out[13]:
```
0    False
1     True
2    False
3     True
4    False
5    False
6    False
dtype: bool
```

In [14]:
```
angulos.notnull()
```

Out[14]:
```
0     True
1    False
2     True
3    False
4     True
5     True
6     True
dtype: bool
```

Podemos utilizar a mascara gerada, por exemplo, para acessar os elementos utilizando indexação com mascara.

In [15]:
```
angulos[angulos.notnull()]
```

Out[15]:
```
0     30.0
2     45.0
4     60.0
5     90.0
6    120.0
dtype: float64
```

In [16]:
```
angulos[angulos.isnull()] = 0
angulos
```

Out[16]:
```
0     30.0
1      0.0
2     45.0
3      0.0
4     60.0
5     90.0
6    120.0
dtype: float64
```

Os métodos `isnull()` e `notnull()` produzem resultados *booleanos* semelhantes em objetos de tipo `DataFrames` .

Removendo valores ausentes

Além do mascaramento usado anteriormente, existem os métodos:

- `dropna()` : Retorna uma versão filtrada dos dados sem os valores ausentes.
- `fillna()` : Retorna uma cópia dos dados com valores faltantes preenchidos.

In [17]:
```
angulos = pd.Series([30, np.nan, 45, None, 60, 90, 120])
angulos
```

Out[17]:
```
0     30.0
1      NaN
2     45.0
3      NaN
4     60.0
5     90.0
6    120.0
dtype: float64
```

In [18]:
```
angulos.dropna()
```

Out[18]:
```
0     30.0
2     45.0
4     60.0
5     90.0
6    120.0
dtype: float64
```

In [19]:
```
angulos
```

Out[19]:
```
0     30.0
1      NaN
2     45.0
3      NaN
4     60.0
5     90.0
6    120.0
dtype: float64
```

In [20]:
```
angulos.fillna(0)
```

Out[20]:
```
0     30.0
1      0.0
2     45.0
3      0.0
4     60.0
5     90.0
6    120.0
dtype: float64
```

In [21]:
```
angulos
```

Out[21]:
```
0     30.0
1      NaN
2     45.0
3      NaN
4     60.0
5     90.0
6    120.0
dtype: float64
```

No caso de `DataFrames` os métodos tem mais opções disponíveis.

In [22]:
```
imagem = pd.DataFrame([[    12, np.nan,    221, 13],
                       [    24, np.nan, np.nan, 24],
                       [    13,    123,    213, 35],
                       [np.nan,     42,    126, 35]])
imagem
```

Out[22]:

	0	1	2	3
0	12.0	NaN	221.0	13
1	24.0	NaN	NaN	24
2	13.0	123.0	213.0	35
3	NaN	42.0	126.0	35

Não podemos eliminar um valor específico de um `DataFrame` . SOmente podemos eliminar linhas ou colunas completas. Dependendo do caso, você pode querer um ou outro, então `dropna()` oferece várias opções. Por padrão, `dropna()` eliminará todas as linhas nas quais qualquer valor nulo estiver presente.

In [23]:
```
imagem.dropna() # axis=0 ou axis='index' ou axis='rows'
```

Out[23]:

	0	1	2	3
2	13.0	123.0	213.0	35

In [24]:
```
imagem.dropna(axis='columns') # axis=1
```

Out[24]:

	3
0	13
1	24
2	35
3	35

As duas alternativas anteriores descartam alguns dados bons.

Você pode estar interessado em eliminar linhas ou colunas com todos os valores ausentes ou com a maioria dos valores ausentes.

Isso pode ser especificado por meio dos parâmetros `how` ou `thresh`, que permitem um controle preciso do número de nulos permitidos.

O padrão é `how='any'`, de forma que qualquer linha ou coluna (dependendo do `axis`) contendo um valor nulo será descartada. Você também pode especificar `how='all'`, que eliminará apenas linhas/colunas que sejam todas valores nulos.

Para um controle fino, o parâmetro `thresh` permite especificar um número mínimo de valores não nulos para a linha/coluna a ser mantida.

In [25]:
```
imagem.index = ['L0', 'L1', 'L2', 'L3']
imagem.columns = ['C0', 'C1', 'C2', 'C3']
imagem
```

Out[25]:

	C0	C1	C2	C3
L0	12.0	NaN	221.0	13
L1	24.0	NaN	NaN	24
L2	13.0	123.0	213.0	35
L3	NaN	42.0	126.0	35

In [26]:
```
imagem.dropna(thresh=3)
```

Out[26]:

	C0	C1	C2	C3
L0	12.0	NaN	221.0	13
L2	13.0	123.0	213.0	35
L3	NaN	42.0	126.0	35

In [27]:
```
imagem.dropna(axis=1, thresh=3)
```

Out[27]:

	C0	C2	C3
L0	12.0	221.0	13
L1	24.0	NaN	24
L2	13.0	213.0	35
L3	NaN	126.0	35

Preenchendo valores ausentes

Às vezes, em vez de descartar os valores ausentes é preferível substituí-los por um valor válido.

O valor pode ser um número específico como zero ou pode ser algum tipo de atribuição ou interpolação dos valores bons.

Você poderia fazer isso diretamente usando o método `isnull()` como máscara, mas por ser uma operação tão comum, **Pandas** fornece o método `fillna()` , que retorna uma cópia do array com os valores nulos substituídos.

Já utilizamos este método anteriormente com `Series` , mas eles pode ser utilizado c também com `DataFrames`

In [28]:
```
imagem.fillna(0)
```

Out[28]:

	C0	C1	C2	C3
L0	12.0	0.0	221.0	13
L1	24.0	0.0	0.0	24
L2	13.0	123.0	213.0	35
L3	0.0	42.0	126.0	35

Podemos alterar ainda o método de preenchimento para ter resultados diferentes.

In [29]:
```
imagem
```

Out[29]:

	C0	C1	C2	C3
L0	12.0	NaN	221.0	13
L1	24.0	NaN	NaN	24
L2	13.0	123.0	213.0	35
L3	NaN	42.0	126.0	35

In [30]:
```
imagem.ffill() #imagem.fillna(method='ffill')
```

Out[30]:

	C0	C1	C2	C3
L0	12.0	NaN	221.0	13
L1	24.0	NaN	221.0	24
L2	13.0	123.0	213.0	35
L3	13.0	42.0	126.0	35

In [31]:
```
imagem.ffill(axis=1) #imagem.fillna(method='ffill', axis=1)
```

Out[31]:

	C0	C1	C2	C3
L0	12.0	12.0	221.0	13.0
L1	24.0	24.0	24.0	24.0
L2	13.0	123.0	213.0	35.0
L3	NaN	42.0	126.0	35.0

In [32]:
```
imagem.ffill(axis=1).ffill()
```

Out[32]:

	C0	C1	C2	C3
L0	12.0	12.0	221.0	13.0
L1	24.0	24.0	24.0	24.0
L2	13.0	123.0	213.0	35.0
L3	13.0	42.0	126.0	35.0

Indexação hierárquica

Até aqui tratamos, principalmente, de dados unidimensionais e bidimensionais, armazenados nos objetos `Series` e `DataFrame` do **Pandas**.

Muitas vezes é importante ir além disso e armazenar dados em estruturas com mais dimensões, ou seja, dados indexados por mais de uma ou duas chaves.

Embora o **Pandas** forneça objetos `Panel` e `Panel4D` que lidam nativamente com dados tridimensionais e quadridimensionais, o padrão na prática é fazer uso da *indexação hierárquica* também chamada de *multiindexação*, que incorpora vários *níveis* de índice em um único índice.

Desta forma, dados de dimensões superiores podem ser representados de forma compacta dentro dos familiares objetos `Series` e `DataFrame` .

Vamos exploraremos a criação direta de objetos `MultiIndex` .

In [33]:
```
index = [('Estação_01', 2000), ('Estação_02', 2000), ('Estação_03',
         ('Estação_01', 2010), ('Estação_02', 2010), ('Estação_03',
         ('Estação_01', 2020), ('Estação_02', 2020), ('Estação_03',
temperaturas = np.ones(9)
temperaturas[:3] = np.random.uniform(25, 30, 3)
temperaturas[3:6] = np.random.uniform(26, 31, 3)
temperaturas[6:] = np.random.uniform(27, 32, 3)
estações = pd.Series(temperaturas, index=index)
estações
```

Out[33]:
```
(Estação_01, 2000)    29.445410
(Estação_02, 2000)    28.936628
(Estação_03, 2000)    27.773272
(Estação_01, 2010)    27.150228
(Estação_02, 2010)    26.375287
(Estação_03, 2010)    29.362904
(Estação_01, 2020)    28.868791
(Estação_02, 2020)    28.587239
(Estação_03, 2020)    30.573099
dtype: float64
```

Podemos agora usar *slicing* com base nesta estrutura de indexação baseada no uso de tuplas.

In [34]:
```
estações[('Estação_03', 2000):('Estação_02', 2020)]
```

Out[34]:
```
(Estação_03, 2000)    27.773272
(Estação_01, 2010)    27.150228
(Estação_02, 2010)    26.375287
(Estação_03, 2010)    29.362904
(Estação_01, 2020)    28.868791
(Estação_02, 2020)    28.587239
dtype: float64
```

Entretanto, esta forma de indexação pode dificultar tarefas com, por exemplo, extrair as temperaturas de 2020 ou apenas as da `Estação_02` .

In [35]:
```
estações[[i for i in estações.index if i[1] == 2010]]
```

Out[35]:
```
(Estação_01, 2010)    27.150228
(Estação_02, 2010)    26.375287
(Estação_03, 2010)    29.362904
dtype: float64
```

In [36]:
```
estações[[i for i in estações.index if i[0] == 'Estação_02']]
```

Out[36]:
```
(Estação_02, 2000)    28.936628
(Estação_02, 2010)    26.375287
(Estação_02, 2020)    28.587239
dtype: float64
```

O resultado obtido é o desejado, ainda que o mecanismo de indexação seja complexo e não muito eficiente, sobre tudo para grandes conjuntos de dados, quanto a sintaxe de *slicing*.

Felizmente, o **Pandas** oferece um mecanismo melhor. A indexação baseada em tupla, utilizada no exemplo anterior, é essencialmente um multi-índice rudimentar. O tipo **Pandas** `MultiIndex` nos fornece o tipo de operações que desejamos ter.

In [37]:
```
index = pd.MultiIndex.from_tuples(index, names=['Estação', 'Ano'])
index
```

Out[37]:
```
MultiIndex([('Estação_01', 2000),
            ('Estação_02', 2000),
            ('Estação_03', 2000),
            ('Estação_01', 2010),
            ('Estação_02', 2010),
            ('Estação_03', 2010),
            ('Estação_01', 2020),
            ('Estação_02', 2020),
            ('Estação_03', 2020)],
           names=['Estação', 'Ano'])
```

In [38]:
```
estações = pd.Series(temperaturas, index=index)
estações
```

Out[38]:
```
Estação     Ano
Estação_01  2000    29.445410
Estação_02  2000    28.936628
Estação_03  2000    27.773272
Estação_01  2010    27.150228
Estação_02  2010    26.375287
Estação_03  2010    29.362904
Estação_01  2020    28.868791
Estação_02  2020    28.587239
Estação_03  2020    30.573099
dtype: float64
```

Agora, as duas primeiras colunas da representação da série mostram os vários valores do índice, enquanto a terceira coluna mostra os dados.

Agora,para extrair as temperaturas de 2020 ou apenas as da `Estação_02` podemos fazer de forma simples.

In [39]:
```
estações[:, 2020]
```

Out[39]:
```
Estação
Estação_01    28.868791
Estação_02    28.587239
Estação_03    30.573099
dtype: float64
```

In [40]:
```
estações['Estação_02', :]
```

Out[40]:
```
Ano
2000    28.936628
2010    26.375287
2020    28.587239
dtype: float64
```

Desta forma temos um array indexado individualmente com apenas as chaves nas quais estamos interessados.

Essa sintaxe é muito mais conveniente e que funciona de forma muito mais eficiente, do que a solução de multi-indexação baseada em tupla com a qual começamos.

Tratando `MultiIndex` como dimensão extra

Umm questionamento importante neste ponto poderia ser: poderíamos facilmente ter armazenado os mesmos dados usando um simples `DataFrame` com rótulos de índices e colunas.

Na verdade, o **Pandas** foi construído com essa equivalência em mente. O método `unstack()` converterá rapidamente um `Series` indexado multiplicadamente em um `DataFrame` indexado convencionalmente.

In [41]:
```
estaçõesDF = estações.unstack()
estaçõesDF
```

Out[41]:

Ano	2000	2010	2020
Estação			
Estação_01	29.445410	27.150228	28.868791
Estação_02	28.936628	26.375287	28.587239
Estação_03	27.773272	29.362904	30.573099

Como seria de se esperar, o método `stack()` fornece a operação oposta.

In [42]:
```
estaçõesS = estaçõesDF.stack()
estaçõesS
```

Out[42]:
```
Estação     Ano
Estação_01  2000    29.445410
            2010    27.150228
            2020    28.868791
Estação_02  2000    28.936628
            2010    26.375287
            2020    28.587239
Estação_03  2000    27.773272
            2010    29.362904
            2020    30.573099
dtype: float64
```

Se `Series` com `MultiIndex` podem ser convertidas em `DataFrames` , para que introduzir este recurso?

A razão é simples: assim como fomos capazes de usar multi-indexação para representar dados bidimensionais dentro de uma `Series` unidimensional, também podemos usá-la para representar dados de três ou mais dimensões em `Series` ou `DataFrames` .

Cada nível extra em um `MultiIndex` representa uma dimensão extra de dados; aproveitar essa propriedade nos dá muito mais flexibilidade nos tipos de dados que podemos representar.

No exemplo anterior poderíamos pensar que a temperatura representada em cada estação, por ano, é a temperatura média. Poderíamos querer outra coluna de dados para cada estação em cada ano com a temperatura máxima daquele ano.

In [43]:
```
tMaxima = temperaturas + np.random.uniform(3, 6, 9)
tMaxima
```

Out[43]:
```
array([35.16772677, 33.94079081, 33.40341182, 31.10828881, 29.7092
0754,
       35.35198806, 32.04233895, 32.78501128, 34.75560891])
```

In [44]:
```
estaçõesDF = pd.DataFrame({'tMed': estações,
                           'tMax': tMaxima})
estaçõesDF
```

Out[44]:

		tMed	tMax
Estação	**Ano**		
Estação_01	**2000**	29.445410	35.167727
Estação_02	**2000**	28.936628	33.940791
Estação_03	**2000**	27.773272	33.403412
Estação_01	**2010**	27.150228	31.108289
Estação_02	**2010**	26.375287	29.709208
Estação_03	**2010**	29.362904	35.351988
Estação_01	**2020**	28.868791	32.042339
Estação_02	**2020**	28.587239	32.785011
Estação_03	**2020**	30.573099	34.755609

In [45]:
```
estaçõesS = estaçõesDF.stack()
estaçõesS
```

Out[45]:
```
Estação     Ano
Estação_01  2000  tMed    29.445410
                  tMax    35.167727
Estação_02  2000  tMed    28.936628
                  tMax    33.940791
Estação_03  2000  tMed    27.773272
                  tMax    33.403412
Estação_01  2010  tMed    27.150228
                  tMax    31.108289
Estação_02  2010  tMed    26.375287
                  tMax    29.709208
Estação_03  2010  tMed    29.362904
                  tMax    35.351988
Estação_01  2020  tMed    28.868791
                  tMax    32.042339
Estação_02  2020  tMed    28.587239
                  tMax    32.785011
Estação_03  2020  tMed    30.573099
                  tMax    34.755609
dtype: float64
```

Podemos tratar estas estruturas utilizando as *unfuncs*, da mesma forma que como feito até aqui.

In [46]:
```
difMedMax = estaçõesDF['tMax'] - estaçõesDF['tMed']
difMedMax
```

Out[46]:
```
Estação     Ano
Estação_01  2000    5.722317
Estação_02  2000    5.004162
Estação_03  2000    5.630140
Estação_01  2010    3.958061
Estação_02  2010    3.333920
Estação_03  2010    5.989084
Estação_01  2020    3.173548
Estação_02  2020    4.197772
Estação_03  2020    4.182510
dtype: float64
```

Desta forma podemos manipular e tratar de maneira fácil e rápida até mesmo dados com muitas dimensões.

Como criar `MultiIndex`

A maneira mais direta de construir um `Series` ou `DataFrame` com indexação múltipla é simplesmente passar uma lista de dois ou mais arrays de índice para o construtor.

In [47]:
```
multiIndexDF = pd.DataFrame(np.random.rand(6, 3),
                   index=[['alpha', 'beta']*3,
                          ['A', 'B', 'C']*2],
                   columns=['set1', 'set2', 'set3'])
multiIndexDF
```

Out[47]:

		set1	set2	set3
alpha	**A**	0.976129	0.328074	0.451698
beta	**B**	0.793604	0.650422	0.490797
alpha	**C**	0.083682	0.774549	0.625333
beta	**A**	0.535920	0.176321	0.377157
alpha	**B**	0.461891	0.886607	0.121380
beta	**C**	0.538530	0.034229	0.814146

In [48]:
```
multiIndexS = multiIndexDF.unstack()
multiIndexS
```

Out[48]:

	set1			set2				
	A	B	C	A	B	C	A	B
alpha	0.976129	0.461891	0.083682	0.328074	0.886607	0.774549	0.451698	0.121380	0.62
beta	0.535920	0.793604	0.538530	0.176321	0.650422	0.034229	0.377157	0.490797	0.81

In [49]:
```
multiIndexDF = multiIndexS.stack()
multiIndexDF
```

Out[49]:

		set1	set2	set3
alpha	A	0.976129	0.328074	0.451698
	B	0.461891	0.886607	0.121380
	C	0.083682	0.774549	0.625333
beta	A	0.535920	0.176321	0.377157
	B	0.793604	0.650422	0.490797
	C	0.538530	0.034229	0.814146

Da mesma forma, se você passar um dicionário com as tuplas como chaves, **Pandas** reconhecerá isso automaticamente e usará um `MultiIndex` .

In [50]:
```
dataDic = {('alpha', 'A'):0.629482, ('alpha', 'B'):0.849897, ('alph
           ('beta', 'A'):0.156, ('beta', 'B'):0.123, ('beta', 'C'):
multiIndexS = pd.Series(dataDic)
multiIndexS
```

Out[50]:
```
alpha  A    0.629482
       B    0.849897
       C    0.001000
beta   A    0.156000
       B    0.123000
       C    0.987000
dtype: float64
```

No entanto, às vezes é útil criar explicitamente um `MultiIndex` .

Podemos construir o `MultiIndex` a partir de uma lista simples de arrays fornecendo os valores do índice em de cada nível.

In [51]:
```
pd.MultiIndex.from_arrays([['alpha']*3 + ['beta']*3, ['A', 'B', 'C'
```

Out[51]:
```
MultiIndex([('alpha', 'A'),
            ('alpha', 'B'),
            ('alpha', 'C'),
            ( 'beta', 'A'),
            ( 'beta', 'B'),
            ( 'beta', 'C')],
           )
```

Podemos construí-lo também a partir de uma lista de tuplas fornecendo os múltiplos valores de índice de cada ponto.

In [52]:
```
i1 = ['alpha', 'beta']
i2 = ['A', 'B', 'C']
tuplas = [(x, y) for x in i1 for y in i2]
tuplas
```

Out[52]:
```
[('alpha', 'A'),
 ('alpha', 'B'),
 ('alpha', 'C'),
 ('beta', 'A'),
 ('beta', 'B'),
 ('beta', 'C')]
```

In [53]:
```
pd.MultiIndex.from_tuples(tuplas)
```

Out[53]:
```
MultiIndex([('alpha', 'A'),
            ('alpha', 'B'),
            ('alpha', 'C'),
            ( 'beta', 'A'),
            ( 'beta', 'B'),
            ( 'beta', 'C')],
           )
```

Podemos até construí-lo a partir do produto cartesiano dos índices.

In [54]:
```
pd.MultiIndex.from_product([i1, i2])
```

Out[54]:
```
MultiIndex([('alpha', 'A'),
            ('alpha', 'B'),
            ('alpha', 'C'),
            ( 'beta', 'A'),
            ( 'beta', 'B'),
            ( 'beta', 'C')],
           )
```

Estes objetos podem ser passados como argumento de `index` ao criar uma `Series` ou um `Dataframe`, ou ser passado para o método `reindex` de uma `Series` ou `DataFrame` já criado.

Às vezes é conveniente atribuir nomes os níveis do `MultiIndex`. Já fizemos isso num dos exemplos anteriores. Isso pode ser feito passando o argumento de `names` para qualquer um dos construtores `MultiIndex` acima ou definindo o atributo de `names` do `index` posteriormente.

In [55]:
```
multiIndexDF.index.names = ['Nível 1', 'Nível 2']
multiIndexDF
```

Out[55]:

		set1	set2	set3
Nível 1	**Nível 2**			
alpha	**A**	0.976129	0.328074	0.451698
	B	0.461891	0.886607	0.121380
	C	0.083682	0.774549	0.625333
beta	**A**	0.535920	0.176321	0.377157
	B	0.793604	0.650422	0.490797
	C	0.538530	0.034229	0.814146

Com conjuntos de dados mais complexos, esta pode ser uma forma útil de acompanhar o significado dos vários valores dos índices.

`MultiIndex` para colunas

Em um `DataFrame`, as linhas e colunas são completamente simétricas, e assim como as linhas podem ter múltiplos níveis de índices, as colunas também podem ter múltiplos níveis.

In [56]:
```
# hierarchical indices and columns
index = pd.MultiIndex.from_product([[2000, 2010, 2020],
                                    ['Estação-01', 'Estação-02', 'E
                                   names=['Ano', 'Estação'])
columns = pd.MultiIndex.from_product([['Diurno', 'Noturno'], ['Temp
                                     names=['Período', 'type'])

# mock some data
data = np.random.random((9, 4))
data[:, ::2] += 30
data[:, 1::2] += 90

# create the DataFrame
estações = pd.DataFrame(data, index=index, columns=columns)
estações
```

Out[56]:

	Período	Diurno		Noturno	
	type	Temp	Umidade	Temp	Umidade
Ano	Estação				
2000	Estação-01	30.362160	90.009664	30.022273	90.709034
	Estação-02	30.999859	90.531018	30.926456	90.498721
	Estação-03	30.497276	90.267606	30.348463	90.195027
2010	Estação-01	30.153747	90.475547	30.070615	90.889947
	Estação-02	30.581456	90.921524	30.997409	90.994051
	Estação-03	30.005718	90.412853	30.125193	90.227360
2020	Estação-01	30.884763	90.495669	30.309056	90.677228
	Estação-02	30.009332	90.375780	30.501290	90.619149
	Estação-03	30.172606	90.471008	30.416142	90.342209

O exemplo anterior mostra como a indexação múltipla para linhas e colunas pode ser muito útil. Trata-se fundamentalmente de dados quadridimensionais, onde as dimensões são o ano da medição, a estação onde foi feita, o período em que foi feita a medição e o parâmetro que foi monitorado. Com isto podemos, por exemplo, pegar todos os dados de temperatura de uma estação específica.

In [57]:
```
estações.loc[:, ('Diurno', 'Temp')][:, 'Estação-01']
```

Out[57]:
```
Ano
2000    30.362160
2010    30.153747
2020    30.884763
Name: (Diurno, Temp), dtype: float64
```

In [58]:
```
# ou de um determinado ano
estações.loc[:, ('Diurno', 'Temp')][2010, :]
```

Out[58]:
```
Estação
Estação-01    30.153747
Estação-02    30.581456
Estação-03    30.005718
Name: (Diurno, Temp), dtype: float64
```

Indexação e slicing com MultiIndex

Indexar e fatiar em um `MultiIndex` foi projetado para ser intuitivo e ajuda se você pensar nos índices como dimensões adicionadas.

Veremos primeiro a indexação de `Series` com indexação múltipla.

In [59]:
```
listaPaises = [('Brasil','America'), ('Alemanha', 'Europa'), ('Itál
               ('Argentina', 'America'), ('França', 'Europa'), ('Ur
               ('Espanha', 'Europa'), ('Inglaterra', 'Europa')]
listaTítulos = [5, 4, 4, 3, 2, 2, 1, 1]

index = pd.MultiIndex.from_tuples(listaPaises, names=['Continente',
index
```

Out[59]:
```
MultiIndex([(    'Brasil', 'America'),
            (  'Alemanha',  'Europa'),
            (    'Itália',  'Europa'),
            ( 'Argentina', 'America'),
            (    'França',  'Europa'),
            (   'Uruguai', 'America'),
            (   'Espanha',  'Europa'),
            ('Inglaterra',  'Europa')],
           names=['Continente', 'País'])
```

In [60]:
```
campMundiais = pd.Series(listaTítulos, index=index)
campMundiais = campMundiais.sort_index()
campMundiais
```

Out[60]:
```
Continente  País
Alemanha    Europa     4
Argentina   America    3
Brasil      America    5
Espanha     Europa     1
França      Europa     2
Inglaterra  Europa     1
Itália      Europa     4
Uruguai     America    2
dtype: int64
```

Podemos acessar um elemento específico, como por exemplo a quantidade de títulos do Brasil.

In [61]:
```
campMundiais[('Brasil', 'America')]
```

Out[61]:
```
5
```

O `MultiIndex` também suporta indexação de apenas um dos níveis do índice.

O resultado é outra `Series`, com os índices de nível inferior mantidos.

In [62]:
```
campMundiais['Brasil']
```

Out[62]:
```
País
America    5
dtype: int64
```

In [63]:
```
campMundiais[:,'America']
```

Out[63]:
```
Continente
Argentina    3
Brasil       5
Uruguai      2
dtype: int64
```

O slicing parcial também está disponível, desde que o `MultiIndex` esteja ordenado

In [64]:
```
campMundiais['Argentina':'França']
```

Out[64]:
```
Continente  País
Argentina   America    3
Brasil      America    5
Espanha     Europa     1
França      Europa     2
dtype: int64
```

Outros tipos de indexação e seleção também funcionam como, por exemplo, seleção baseada em máscaras booleanas.

In [65]:
```
campMundiais[campMundiais > 2]
```

Out[65]:
```
Continente  País
Alemanha    Europa     4
Argentina   America    3
Brasil      America    5
Itália      Europa     4
dtype: int64
```

In [66]:
```
campMundiais[['Brasil', 'Argentina', 'Uruguai']]
```

Out[66]:
```
Continente  País
Brasil      America    5
Argentina   America    3
Uruguai     America    2
dtype: int64
```

Um `DataFrame` com indexação múltipla se comporta de maneira semelhante.

In [67]:
```
estações
```

Out[67]:

	Período	Diurno		Noturno	
	type	Temp	Umidade	Temp	Umidade
Ano	Estação				
	Estação-01	30.362160	90.009664	30.022273	90.709034
2000	**Estação-02**	30.999859	90.531018	30.926456	90.498721
	Estação-03	30.497276	90.267606	30.348463	90.195027
	Estação-01	30.153747	90.475547	30.070615	90.889947
2010	**Estação-02**	30.581456	90.921524	30.997409	90.994051
	Estação-03	30.005718	90.412853	30.125193	90.227360
	Estação-01	30.884763	90.495669	30.309056	90.677228
2020	**Estação-02**	30.009332	90.375780	30.501290	90.619149
	Estação-03	30.172606	90.471008	30.416142	90.342209

Apenas é importante lembrar que as colunas são consideradas antes que as linhas em um `DataFrame` , e por tanto a sintaxe usada para `Series` com indexação múltipla se aplica às colunas.

In [68]:
```
estações['Diurno', 'Temp']
```

Out[68]:
```
Ano   Estação
2000  Estação-01    30.362160
      Estação-02    30.999859
      Estação-03    30.497276
2010  Estação-01    30.153747
      Estação-02    30.581456
      Estação-03    30.005718
2020  Estação-01    30.884763
      Estação-02    30.009332
      Estação-03    30.172606
Name: (Diurno, Temp), dtype: float64
```

In [69]:
```
estações['Noturno']
```

Out[69]:

		type	Temp	Umidade
Ano	**Estação**			
2000	**Estação-01**	30.022273	90.709034
	Estação-02	30.926456	90.498721
	Estação-03	30.348463	90.195027
2010	**Estação-01**	30.070615	90.889947
	Estação-02	30.997409	90.994051
	Estação-03	30.125193	90.227360
2020	**Estação-01**	30.309056	90.677228
	Estação-02	30.501290	90.619149
	Estação-03	30.416142	90.342209

In [70]:
```
estações['Diurno', 'Temp'][2010, 'Estação-01']
```

Out[70]:
```
30.15374650280863
```

In [71]:
```
estações['Diurno', 'Temp'][2010]
```

Out[71]:
```
Estação
Estação-01    30.153747
Estação-02    30.581456
Estação-03    30.005718
Name: (Diurno, Temp), dtype: float64
```

In [72]:
```
estações['Diurno', 'Temp'][:, 'Estação-01']
```

Out[72]:
```
Ano
2000    30.362160
2010    30.153747
2020    30.884763
Name: (Diurno, Temp), dtype: float64
```

Além disso podemos usar o `loc` , `iloc` .

In [73]:
```
estações.iloc[:1, :2]
```

Out[73]:

		Período type	Diurno Temp	Diurno Umidade
Ano	**Estação**			
2000	**Estação-01**	30.36216	90.009664	

O `iloc` acessa o índice implícito, ou seja, trata os dados como uma matriz bidimensional.

In [82]:
```
estações.values
```

Out[82]:
```
array([[30.36216039, 90.00966447, 30.02227261, 90.70903403],
       [30.99985855, 90.53101816, 30.92645588, 90.49872061],
       [30.49727577, 90.26760579, 30.3484632 , 90.19502717],
       [30.1537465 , 90.47554715, 30.07061486, 90.88994701],
       [30.58145626, 90.92152376, 30.99740931, 90.99405103],
       [30.0057182 , 90.41285293, 30.12519316, 90.22736032],
       [30.88476331, 90.49566875, 30.30905603, 90.67722775],
       [30.00933233, 90.37577963, 30.50129003, 90.61914935],
       [30.17260555, 90.47100805, 30.41614161, 90.34220944]])
```

Acessar os dados desta forma pode ser mais complicado já que:

- As primeiras três linhas correspondem à chave `2000` , as próximas três a `2010` e as últimas três a `2020` .
- As linhas 0, 3 e 6 correspondem à chave `Estação-01` , as linhas 1, 4 e 7 a `Estação-02` e as linhas 2, 5 e 8 à chave `Estação-03` .
- As primeiras duas colunas correspondem à chave `Diurno` e as próximas duas à chave `Noturno` .
- As colunas 0 e 2 correspondem à have `Temp` e as colunas 1 e 3 à chave `Umidade` .

In [86]:
```
# Para conseguir o mesmo resultado que
#estações['Diurno', 'Temp'][:, 'Estação-01']
estações.iloc[0::3, 0]
```

Out[86]:
```
Ano   Estação
2000  Estação-01    30.362160
2010  Estação-01    30.153747
2020  Estação-01    30.884763
Name: (Diurno, Temp), dtype: float64
```

Esses indexadores fornecem uma visão semelhante a uma matriz dos dados bidimensionais subjacentes, mas cada índice individual em `loc` ou `iloc` pode receber uma tupla de vários índices.

In [74]:
```
estações.loc[:, ('Diurno', 'Temp')]
```

Out[74]:
```
Ano   Estação
2000  Estação-01    30.362160
      Estação-02    30.999859
      Estação-03    30.497276
2010  Estação-01    30.153747
      Estação-02    30.581456
      Estação-03    30.005718
2020  Estação-01    30.884763
      Estação-02    30.009332
      Estação-03    30.172606
Name: (Diurno, Temp), dtype: float64
```

In [75]:
```
estações.loc[(2020, 'Estação-03'), ('Diurno', 'Temp')]
```

Out[75]:
```
30.172605553889188
```

In [79]:
```
estações
```

Out[79]:

	Período type	Diurno		Noturno	
Ano	Estação	Temp	Umidade	Temp	Umidade
2000	Estação-01	30.362160	90.009664	30.022273	90.709034
	Estação-02	30.999859	90.531018	30.926456	90.498721
	Estação-03	30.497276	90.267606	30.348463	90.195027
2010	Estação-01	30.153747	90.475547	30.070615	90.889947
	Estação-02	30.581456	90.921524	30.997409	90.994051
	Estação-03	30.005718	90.412853	30.125193	90.227360
2020	Estação-01	30.884763	90.495669	30.309056	90.677228
	Estação-02	30.009332	90.375780	30.501290	90.619149
	Estação-03	30.172606	90.471008	30.416142	90.342209

In [81]:
```
estações.iloc[(3, 0)]
```

Out[81]:
```
30.15374650280863
```

Reorganizando `MultiIndex`

Um dos segredos para trabalhar com dados indexados de forma múltipla é saber como transformá-los de maneira eficaz. Há uma série de operações que preservam todas as informações do conjunto de dados, mas as reorganizam para facilitar a manipulação dos mesmos. Vimos um breve exemplo disso nos métodos `stack()` e `unstack()`, mas existem muitas outras maneiras de controlar com precisão o rearranjo de dados entre índices hierárquicos e colunas.

Índices ordenados e não ordenados

Muitas das operações de slicing com `MultiIndex` n~áo funciona se o índice não estiver ordenado.

Começaremos criando alguns dados indexados simples com índices não ordenados.

In [105]:
```
multiIndexS = pd.Series(np.random.rand(9),
                        index=[['gamma']*3 + ['beta']*3 + ['alpha']*3,
                               ['A', 'C', 'B']*3])
multiIndexS
```

Out[105]:
```
gamma  A    0.651887
       C    0.084866
       B    0.029783
beta   A    0.508477
       C    0.849120
       B    0.865123
alpha  A    0.080406
       C    0.827525
       B    0.121062
dtype: float64
```

Se tentarmos obter um slicing destes índices teremos um erro.

In [106]:
```
try:
    #multiIndexS['gamma': 'alpha']
    multiIndexS[:, 'A':'B']
except Exception as e:
    print(e)
```

```
Expected label or tuple of labels, got (slice(None, None, None), s
lice('A', 'B', None))
```

Embora não esteja totalmente claro na mensagem de erro, este é o resultado da não ordenação do `MultiIndex` .

Por vários motivos, slicing e outras operações semelhantes exigem que os níveis no `MultiIndex` estejam em ordenados.

O **Pandas** fornece uma série de rotinas para realizar esse tipo de classificação, como por exemplos os métodos `sort_index()` e `sortlevel()` do `DataFrame` .

In [107]:
```
multiIndexS = multiIndexS.sort_index()
multiIndexS
```

Out[107]:
```
alpha  A    0.080406
       B    0.121062
       C    0.827525
beta   A    0.508477
       B    0.865123
       C    0.849120
gamma  A    0.651887
       B    0.029783
       C    0.084866
dtype: float64
```

In [116]:
```
multiIndexS.loc['alpha': 'beta']
```

Out[116]:
```
alpha  A    0.080406
       B    0.121062
       C    0.827525
beta   A    0.508477
       B    0.865123
       C    0.849120
dtype: float64
```

In [117]:
```
multiIndexS.loc[:, 'A':'B']
```

Out[117]:
```
alpha  A    0.080406
       B    0.121062
beta   A    0.508477
       B    0.865123
gamma  A    0.651887
       B    0.029783
dtype: float64
```

Empilhamento e desempilhamento de índices

Como vimos brevemente antes, é possível converter um conjunto de dados de um `MultiIndex` empilhado para uma representação bidimensional simples, especificando opcionalmente o nível a ser usado.

In [118]:
```
multiIndexS.unstack()
```

Out[118]:

	A	B	C
alpha	0.080406	0.121062	0.827525
beta	0.508477	0.865123	0.849120
gamma	0.651887	0.029783	0.084866

In [119]:
```
multiIndexS.unstack(level=0)
```

Out[119]:

	alpha	beta	gamma
A	0.080406	0.508477	0.651887
B	0.121062	0.865123	0.029783
C	0.827525	0.849120	0.084866

In [120]:
```
multiIndexS.unstack(level=1)
```

Out[120]:

	A	B	C
alpha	0.080406	0.121062	0.827525
beta	0.508477	0.865123	0.849120
gamma	0.651887	0.029783	0.084866

A operação inversa de `unstack()` é `stack()`, que pode ser usado para recuperar a série original.

In [121]:
```
multiIndexS.unstack().stack()
```

Out[121]:
```
alpha  A    0.080406
       B    0.121062
       C    0.827525
beta   A    0.508477
       B    0.865123
       C    0.849120
gamma  A    0.651887
       B    0.029783
       C    0.084866
dtype: float64
```

Definindo e redefinindo os índices

Outra forma de reorganizar os dados hierárquicos é transformar os rótulos do índice em colunas; isso pode ser feito com o método `reset_index`.

In [122]:
```
multiIndexS.reset_index(name='Valor')
```

Out[122]:

	level_0	level_1	Valor
0	alpha	A	0.080406
1	alpha	B	0.121062
2	alpha	C	0.827525
3	beta	A	0.508477
4	beta	B	0.865123
5	beta	C	0.849120
6	gamma	A	0.651887
7	gamma	B	0.029783
8	gamma	C	0.084866

Muitas vezes, ao trabalhar com dados no mundo real, os dados brutos de entrada têm esta aparência e é útil construir um `MultiIndex` a partir dos valores da coluna. Isso pode ser feito com o método `set_index` do `DataFrame`, que retorna um `DataFrame` com indexação múltipla.

In [123]:
```
tabela = multiIndexS.reset_index(name='Valor')
tabela.set_index(['level_0', 'level_1'])
```

Out[123]:

		Valor
level_0	**level_1**	
alpha	**A**	0.080406
	B	0.121062
	C	0.827525
beta	**A**	0.508477
	B	0.865123
	C	0.849120
gamma	**A**	0.651887
	B	0.029783
	C	0.084866