Analisando Dados com Pandas

Asimov Academy

ASIMOV

Conteúdo

01. Conceitos básicos de Pandas

Pandas é uma biblioteca do Python amplamente utilizada na análise de dados. Com o Pandas, é possível explorar, limpar e manipular conjuntos de dados, permitindo a análise de pequenos até grandes volumes de informações. Essa versatilidade torna o Pandas uma ferramenta poderosa e indispensável para profissionais de análise de dados.

Como estudar Pandas?

Assim como em qualquer área de programação, a prática é fundamental para dominar o Pandas. A melhor maneira de estudar esta biblioteca é através da prática ativa. Ao enfrentar problemas do cotidiano, procure aplicar os conceitos apresentados utilizando o Pandas. Uma excelente forma de exercitar isso é buscar conjuntos de dados abertos e realizar análises utilizando as funcionalidades do Pandas.

02. Series

O elemento central do Pandas são as tabelas, que são composições de colunas. No Pandas, as tabelas são o que chamamos de DataFrame e cada coluna, por sua vez, chamamos de Series. Neste tópico, iremos falar sobre Series, como elas se comportam, suas propriedades e o que podemos fazer com esses elementos no Pandas.

Então, vamos começar!

Por convenção, importamos Pandas como pd e Numpy como np. Isso significa que sempre que for usar essas bibliotecas no código, ao invés de usar o nome, chamaremos como pd e np.

```
#Importando bibliotecas
import pandas as pd
import numpy as np
```

Agora, serão definidas algumas variáveis com estruturas de dados do Python já conhecidas: listas, dicionários e um array. Nesse último caso, usaremos o NumPy para definir essa estrutura e a partir delas criaremos algumas Series.

```
labels = ['a', 'b', 'c']
minha_lista = [10, 20, 30]
arr = np.array([10, 20, 30])
d = {'a':10, 'b':20, 'c':30}
```

Vamos começar criando uma Series usando listas. Para isso, executamos o comando a seguir usando o método Series do Pandas.

```
pd.Series(labels)
```

saída

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

Os itens da lista 'labels', definidos anteriormente, agora estão organizados como uma coluna. A primeira coluna são os índices de cada dado, é o elemento que informa a posição daquele dado. Pensando em uma tabela, cada índice da primeira coluna identifica uma linha na tabela, o que é muito importante para a manipulação desses dados.

Podemos dizer ao Pandas como criar esses índices.

```
pd.Series(data=labels, index=minha_lista)
```

Nesse caso, ele vai usar os valores de ‘minha_lista’ como índices, enquanto os valores de ‘labels’ são o conjunto de dados. Uma característica importante é que os índices podem ser strings ou números, então podemos defini-los da maneira que corresponder melhor ao nosso conjunto de dados.

```
10    a
20    b
30    c
dtype: object
```

Também podemos observar que o tipo dessa série é identificado como objeto. Isso acontece porque o Pandas não faz inferência de tipo e sempre que passamos uma lista como essa, ele trata como objetos Python genéricos.

Quando instanciamos uma série, por padrão, os parâmetros ‘data’ e ‘index’ são os primeiros a serem passados. Dessa forma, se reescrevermos o comando assim,

```
pd.Series(labels, minha_lista)
```

Ele entende que ‘labels’ é o parâmetro data e ‘minha_lista’ é o parâmetro index."

Podemos criar uma série a partir de um dicionário. A estrutura de um dicionário é um conjunto de chaves e valores. Usaremos a estrutura definida no início, onde cada letra está como chave de um valor numérico: ‘a’ é chave do 10, ‘b’ é chave do 20 e ‘c’ é chave do 30.

```
d = {'a':10, 'b':20, 'c':30}
```

```
pd.Series(d)
```

Nesse caso, o Pandas entende que as chaves dos valores serão, respectivamente, os valores para a coluna de índice.

```
a    10
b    20
c    30
dtype: int64
```

Agora que vimos como criar uma Series, vamos ver algumas manipulações e para isso, criaremos duas novas estruturas.

```
ser1 = pd.Series([1,2,3,4], index = ['EUA', 'Alemanha', 'Rússia', 'Japão'])
```

Essa é apenas uma maneira sucinta de escrever o comando, sem definir previamente variáveis que forneçam valores para ‘data’ e ‘index’. Podemos definir diretamente quando chamamos o método Series. Nesse caso, o primeiro parâmetro é uma lista de valores inteiros, este é o nosso conjunto de dados, e o segundo parâmetro é uma lista de países, que serão os identificadores dos nossos dados, como podemos ver a seguir.

```
EUA         1
Alemanha    2
Rússia      3
Japão       4
dtype: int64
```

Seguindo, vamos definir outra série e nesta, a única diferença da anterior é que, na posição da Rússia, colocaremos a Itália como identificador.

```
ser2 = pd.Series([1,2,3,4], index = ['EUA', 'Alemanha', 'Italia', 'Japão'])
```

```
EUA         1
Alemanha    2
Italia      3
Japão       4
dtype: int64
```

Acessaremos valores de uma série usando índices, com a mesma notação que usamos para acessar valores em strings e listas, com colchetes.

```
ser1['EUA']
```

```
1
```

Podemos passar uma lista de índices; ele retornará apenas os valores relacionados.

```
ser1[['Alemanha', 'Rússia']]
```

```
Alemanha    2
Rússia      3
dtype: int64
```

Então, para acessar um valor, passamos o índice como string e para múltiplos valores, passamos uma lista de índices.

No Pandas, podemos somar duas séries; ele realiza a operação orientado pelo índice, percorrendo um a um.

```
ser1 + ser2
```

```
Alemanha    4.0
EUA         2.0
Italia      NaN
Japão       8.0
Rússia      NaN
dtype: float64
```

Somando quem tem índices iguais e quando encontra um índice presente em apenas uma das séries, o resultado retornado será NaN (Not a Number), que indica um valor ausente. No nosso exemplo, a Itália está associada ao valor 5 na ser2, mas em ser1 ele não encontra valor definido que corresponda a isso e preenche com NaN.

03. DataFrames e Manipulação de Colunas

Vamos estudar DataFrames, o que são e como manipular essas estruturas. Aqui também veremos as semelhanças e diferenças entre DataFrames e Series. Os DataFrames são tabelas criadas com o Pandas; podemos entendê-las também como composições de Series. Ao longo deste tópico, esse conceito de composição vai ficar mais claro.

Neste ponto, além de importar o Pandas, será necessário importar o método random da biblioteca NumPy, que usaremos para gerar números aleatórios em um formato que nos possibilite preencher uma tabela.

```
import pandas as pd
from numpy.random import randn
```

Com o método `DataFrame`, vamos criar um DataFrame. Para isso, também precisamos de uma tabela ou algum conjunto de dados bidimensional que caiba no formato do DataFrame.

```
df = pd.DataFrame(randn(5,4), index=["A", "B", "C", "D", "E"], columns="W X Y Z".split())
```

Aqui usamos o método randn(5,4) para criar uma tabela de números aleatórios de cinco linhas por quatro colunas, além de definir o índice com a sequência de letras A, B, C, D, E e a identificação para cada coluna através da string columns="W X Y Z".split(). O método split() quebra essa string em uma lista de strings. Executando, obtemos o que vem a seguir.

	W	X	Y	Z
A	-0.229117	0.087277	0.337653	-0.309933
B	-0.489985	0.907704	-0.196438	-0.199404
C	0.219143	-1.973148	-0.691878	0.498034
D	-1.208147	0.550798	0.836310	-0.678003
E	-0.028455	1.507470	-1.873436	0.642079

Se verificarmos qual o tipo desse elemento com o comando `type(df)`, ele retorna o tipo `pandas.core.frame.DataFrame`, que é um conjunto específico do Pandas e estudaremos ao longo deste material. Em um DataFrame, cada valor é a composição de uma linha e uma coluna; é com essas informações que vamos fazer seleções e manipulação desses dados.

Quando trabalhamos com uma série na seleção por colchetes, nós definimos o índice. Lembre-se de quando estudamos séries no tópico anterior. Com DataFrame, essa seleção é feita definindo uma coluna nos colchetes.

```
df['W']
```

Nessa seleção, ele nos devolverá todos os valores referentes à coluna W para todos os índices.

```
A   -0.229117
B   -0.489985
C    0.219143
D   -1.208147
E   -0.028455
Name: W, dtype: float64
```

Quando fazemos uma seleção `df['W']`, por padrão, o Python nos retorna uma série. Podemos verificar isso checando o tipo do DataFrame com `type(df['W'])`. A saída deve ser `pandas.core.series.Series`.

Se quisermos obter um DataFrame com apenas uma coluna, a busca deve ser feita passando uma lista com o nome da coluna que desejamos buscar.

```
df[['W']]
```

	W
A	-0.229117
B	-0.489985
C	0.219143
D	-1.208147
E	-0.028455

Dessa maneira, também podemos passar mais de um índice e obter quantas colunas desejarmos. Aqui, vamos obter um DataFrame com duas colunas.

```
df[['W', 'X']]
```

	W	X
A	-0.229117	0.087277
B	-0.489985	0.907704
C	0.219143	-1.973148
D	-1.208147	0.550798

	W	X
E	-0.028455	1.507470

Uma das manipulações que podemos realizar em um DataFrame é a criação de novas colunas atribuindo valores ao rótulo. Vamos fazer isso para entender melhor. Seguindo a notação que usamos para fazer uma busca, passamos um nome que desejamos dar para a nova coluna, `df['new']`. Se executarmos esse comando dessa maneira, o interpretador vai indicar um `KeyError: new`. Isso acontece porque essa chave não existe; ainda não temos uma coluna com esse nome no DataFrame. Podemos definir essa coluna atribuindo valores a ela, como por exemplo assim:

```
df['new'] = df['W']
```

	W	X	Y	Z	new
A	-0.229117	0.087277	0.337653	-0.309933	-0.229117
B	-0.489985	0.907704	-0.196438	-0.199404	-0.489985
C	0.219143	-1.973148	-0.691878	0.498034	0.219143
D	-1.208147	0.550798	0.836310	-0.678003	-1.208147
E	-0.028455	1.507470	-1.873436	0.642079	-0.028455

Dessa forma, definimos a coluna new como uma cópia da coluna W; ela contém todos os valores dessa coluna. Essa é uma das formas de inserir dados em um DataFrame. Ao longo deste material, abordaremos outras maneiras de fazer isso. Podemos somar duas séries e atribuir a uma nova coluna; vamos usar ainda a coluna new para demonstrar essa situação.

```
df['new'] = df['W'] + df['Y']
```

Lembrando do conteúdo de Series, elas são orientadas a índices. Portanto, essa operação aritmética será feita somando índice a índice das duas séries definidas.

	W	X	Y	Z	new
A	-0.229117	0.087277	0.337653	-0.309933	0.108536
B	-0.489985	0.907704	-0.196438	-0.199404	-0.686423
C	0.219143	-1.973148	-0.691878	0.498034	-0.472735

	W	X	Y	Z	new
D	-1.208147	0.550798	0.836310	-0.678003	-0.371837
E	-0.028455	1.507470	-1.873436	0.642079	-1.901891

Agora, a coluna new contém os valores da soma entre as colunas W e Y.

Falamos sobre criação e inserção de valores em uma coluna. Agora, vamos ver algumas formas de deletar uma coluna em um DataFrame. Poderíamos usar o método `del`, que exclui automaticamente a coluna selecionada.

```
del df['new']
```

retornaria a nossa tabela inicial

	W	X	Y	Z
A	-0.229117	0.087277	0.337653	-0.309933
B	-0.489985	0.907704	-0.196438	-0.199404
C	0.219143	-1.973148	-0.691878	0.498034
D	-1.208147	0.550798	0.836310	-0.678003
E	-0.028455	1.507470	-1.873436	0.642079

Também podemos usar um método do DataFrame chamado drop. Com ele, podemos remover linhas ou colunas, mas por padrão ele afeta as colunas. Vamos ver isso. Usaremos aqui a tabela com a coluna new.

```
df.drop('new', axis=1)
```

Axis é o eixo onde o método vai atuar. Eixo 1 corresponde a colunas e eixo 0 corresponde a linhas. Executando o comando, ele retorna a tabela sem a coluna new.

	W	X	Y	Z
A	-0.229117	0.087277	0.337653	-0.309933
B	-0.489985	0.907704	-0.196438	-0.199404
C	0.219143	-1.973148	-0.691878	0.498034

	W	X	Y	Z
D	-1.208147	0.550798	0.836310	-0.678003
E	-0.028455	1.507470	-1.873436	0.642079

O peculiar desse método é que ele não altera a estrutura original do nosso DataFrame; ele atua em cima de uma cópia e nos devolve instantaneamente a estrutura sem a coluna indicada, para que possamos manipular esse novo arranjo naquele momento. Entretanto, se acessarmos novamente o DataFrame, ele retornará à estrutura original com todas as colunas.

```
df
```

	W	X	Y	Z	new
A	-0.229117	0.087277	0.337653	-0.309933	0.108536
B	-0.489985	0.907704	-0.196438	-0.199404	-0.686423
C	0.219143	-1.973148	-0.691878	0.498034	-0.472735
D	-1.208147	0.550798	0.836310	-0.678003	-0.371837
E	-0.028455	1.507470	-1.873436	0.642079	-1.901891

Para trabalhar com a estrutura após a remoção da coluna pelo método drop, podemos salvar em uma variável e acessar depois: `df2 = df.drop('new', axis=1)`.

A princípio, essa ideia de remoção pode parecer confusa, mas esse conceito é muito importante. A maioria das funções do Pandas não altera a estrutura original do DataFrame; elas fazem a manipulação em uma cópia e nos entregam essa cópia com as alterações que definimos, para que possamos trabalhar com isso. No entanto, há uma maneira de aplicar essas alterações na estrutura original.

```
df.drop('new', axis=1, inplace=True) #O inplace informa que a função será aplicada na
↪  estrutura original
```

Quando executamos esse comando, não obtemos um retorno instantâneo da estrutura na tela porque ele trabalhou originalmente no DataFrame e não com uma cópia.

Podemos fazer seleção nas linhas usando `df.loc` e `df.iloc`. Com `iloc`, passamos os índices que ele procura para essa linha.

```
df.loc['A']
```

```
W    -0.229117
X     0.087277
```

```
Y      0.337653
Z     -0.309933
new    0.108536
Name: A, dtype: float64
```

Observe que ele faz uma transposição; transformou os rótulos das colunas em índices e devolveu uma série dessa transposição. Também podemos passar mais valores.

```
df.loc[['A', 'B']]
```

Ele nos devolve como dataframe:

	W	X	Y	Z	new
A	-0.229117	0.087277	0.337653	-0.309933	0.108536
B	-0.489985	0.907704	-0.196438	-0.199404	-0.686423

Com `loc`, também podemos, além de passar a lista dos índices, passar uma vírgula e informar uma coluna. Assim, obteríamos os valores na coluna definida para cada índice.

```
df.loc[['A', 'B'], 'W']
```

nesse caso a resposta vai ser uma serie

```
A   -0.229117
B   -0.489985
Name: W, dtype: float64
```

Se quisermos que essa seleção retorne um DataFrame, passamos as colunas em uma lista também.

```
df.loc[['A', 'B'], ['W']]
```

	W
A	-0.229117
B	-0.489985

Por fim, vamos usar `df.iloc`. Com ele, acessamos os valores de forma numérica, passando o índice da coluna e o índice da linha. Olhando para nossa tabela,

	W	X	Y	Z	new
A	-0.229117	0.087277	0.337653	-0.309933	0.108536
B	-0.489985	0.907704	-0.196438	-0.199404	-0.686423
C	0.219143	-1.973148	-0.691878	0.498034	-0.472735
D	-1.208147	0.550798	0.836310	-0.678003	-0.371837
E	-0.028455	1.507470	-1.873436	0.642079	-1.901891

Se quisermos acessar o primeiro valor da coluna Y com iloc, passaríamos os índices da seguinte forma:

```
df.iloc[0, 2]
```

```
0.337653
```

04. Iloc e filtros

Até agora, estudamos como fazer seleções em DataFrames. Quando usamos colchetes em uma Series, são selecionados os índices. Nos DataFrames, essa seleção é feita com os rótulos de uma coluna e podemos passar apenas um valor por string ou uma lista de valores, que nos retornaria um DataFrame específico. Também vimos como fazer seleções com `loc`, passando uma lista de índices e até mesmo passando duas listas, uma com os índices e outra com as colunas.

Iloc

E vimos um pouco sobre como fazer essas seleções com `iloc`. Com ele, podemos fazer seleção numérica passando alguma informação referente à linha e alguma informação referente à coluna.

```
df.iloc[0, 2]
```

A manipulação com esse método é bem parecida com o que fazemos em strings e listas. Vamos ver isso na prática.

	W	X	Y	Z
A	-0.229117	0.087277	0.337653	-0.309933
B	-0.489985	0.907704	-0.196438	-0.199404
C	0.219143	-1.973148	-0.691878	0.498034
D	-1.208147	0.550798	0.836310	-0.678003
E	-0.028455	1.507470	-1.873436	0.642079

Se quisermos selecionar os valores até o índice D para todas as colunas, podemos fazer isso da seguinte maneira:

```
df.iloc[:-1, :]
```

Essa notação é a mesma que usamos para fatiar strings; os dois pontos indicam que queremos todos os valores e o -1 limita até a penúltima linha. Ele basicamente está dizendo que queremos todos os valores, exceto os da última linha, em todas as colunas do nosso DataFrame.

	W	X	Y	Z
A	-0.229117	0.087277	0.337653	-0.309933

	W	X	Y	Z
B	-0.489985	0.907704	-0.196438	-0.199404
C	0.219143	-1.973148	-0.691878	0.498034
D	-1.208147	0.550798	0.836310	-0.678003

Poderíamos ter definido em qual coluna iniciar essa seleção.

```
df.iloc[:-1, 1:]
```

Aqui ele selecionaria os valores a partir da coluna X.

	X	Y	Z
A	0.087277	0.337653	-0.309933
B	0.907704	-0.196438	-0.199404
C	-1.973148	-0.691878	0.498034
D	0.550798	0.836310	-0.678003

E também podemos fazer para valores específicos. Vamos supor que desejamos selecionar os valores correspondentes às linhas B, C, D e as colunas X e Y. Definiríamos a seleção a partir da linha 1 até a linha 3, índice D. Como comentado anteriormente, essa regra de seleção é muito semelhante ao que fazemos em listas, então o que passamos antes dos dois pontos entra na seleção e o que está depois não entra. Na nossa seleção, a linha de índice 1 será incluída, enquanto a linha de índice 4 será excluída. Com as colunas, a regra é a mesma: como queremos os valores referentes às colunas X e Y, definimos a seleção a partir de 1 e paramos no 3 sem incluí-lo.

```
df.iloc[1:4, 1:3]
```

	X	Y
A	0.087277	0.337653
B	0.907704	-0.196438
C	-1.973148	-0.691878
D	0.550798	0.836310

Essa metodologia é útil quando precisamos lidar com matrizes muito grandes e os índices e colunas não fazem muito sentido. Podemos manipular usando valores numéricos com o método iloc.

Filtros

Vamos aprender sobre seleção condicional em DataFrames. Com operações aritméticas, o DataFrame se orienta pelos índices e pelas colunas, buscando valores correspondentes em cada posição e realizando a operação. Também podemos fazer operações condicionais com DataFrames, e é isso que vamos aprender agora, usando o DataFrame original.

	W	X	Y	Z
A	-0.229117	0.087277	0.337653	-0.309933
B	-0.489985	0.907704	-0.196438	-0.199404
C	0.219143	-1.973148	-0.691878	0.498034
D	-1.208147	0.550798	0.836310	-0.678003
E	-0.028455	1.507470	-1.873436	0.642079

No comando abaixo, estou perguntando se o DataFrame anterior, df, é maior que zero, o que parece não fazer sentido, já que estou comparando um DataFrame com um inteiro.

```
df > 0
```

Ele comparou todos os valores e, quando eram maiores que zero, ele colocou True, e quando eram menores, colocou False.

	W	X	Y	Z
A	False	True	True	False
B	False	True	False	False
C	True	Fase	False	True
D	False	True	True	False
E	False	True	False	True

A princípio, isso não nos diz muita coisa, mas podemos usar uma condicional como essa dentro de colchetes. Aqui, pedimos o DataFrame nos casos em que ele for maior do que zero.

```
df[df > 0]
```

O resultado é um pouco confuso; agora, em alguns casos, ele preencheu com NaN (Not a Number) e retornou apenas os valores que são maiores que zero.

	W	X	Y	Z
A	NaN	0.087277	0.337653	NaN
B	NaN	0.907704	NaN	NaN
C	0.219143	NaN	NaN	0.498034
D	NaN	0.550798	0.836310	NaN
E	NaN	1.507470	NaN	0.642079

Mas o importante é entender que essa será uma forma de fazermos filtros nos nossos dados. Esse método é muito importante e se assemelha aos filtros que podemos fazer em tabelas do Excel. Vamos exercitar um pouco mais, imaginando agora se é possível selecionar os dados apenas quando os valores na coluna forem maiores que zero. Para isso, vamos modificar a condição.

```
df['Y'] > 0
```

ele nos retorna uma serie

```
A     True
B    False
C    False
D     True
E    False
Name: Y, dtype: bool
```

Vamos filtrar para quando os valores da coluna Y forem maiores do que zero, mas queremos um DataFrame com as respostas.

```
df[df['Y'] > 0]
```

Ele retorna um DataFrame sem os índices que não contemplam a condição que definimos. Isso acontece porque aqui ele está se baseando nos índices e não mais no DataFrame como um todo, como anteriormente. Então, pedimos para ele retornar o DataFrame quando o valor na coluna Y for maior do que zero.

	W	X	Y	Z
A	-0.229117	0.087277	0.337653	-0.309933

	W	X	Y	Z
D	-1.208147	0.550798	0.836310	-0.678003

Também é possível passar múltiplas condições como parâmetro. Vamos observar o comportamento disso quando não as colocamos entre colchetes.

```
(df['X'] > 0) & (df['Y'] > 0)
```

O Pandas faz a seleção linha a linha nesse caso. Quando uma linha atender à primeira condição e, ao mesmo tempo, à segunda, retornará True, o que ocorrerá para todos os valores em ambas as colunas. Basta verificar na nossa tabela e podemos constatar que isso está correto.

```
A     True
B    False
C    False
D     True
E    False
dtype: bool
```

Agora, usando colchetes, ele retornará apenas os índices, com todos os seus valores, que cumprem essa regra.

```
df[(df['X'] > 0) & (df['Y'] > 0)]
```

	W	X	Y	Z
A	-0.229117	0.087277	0.337653	-0.309933
D	-1.208147	0.55079	0.836310	-0.678003

05. Operações com índices

Trabalharemos agora um pouco mais com índices e entenderemos suas particularidades. Vou apresentar algumas funções que nos permitirão substituir, pesquisar e manipular índices em geral.

Voltando ao nosso DataFrame gerado anteriormente, vamos ver como podemos pesquisar os índices com o Pandas.

	W	X	Y	Z
A	-0.229117	0.087277	0.337653	-0.309933
B	-0.489985	0.907704	-0.196438	-0.199404
C	0.219143	-1.973148	-0.691878	0.498034
D	-1.208147	0.550798	0.836310	-0.678003
E	-0.028455	1.507470	-1.873436	0.642079

Executando o comando abaixo, ele nos retorna os índices do nosso DataFrame dentro de uma lista.

```
df.index
```

```
Index(['A', 'B', 'C', 'D', 'E'], dtype='object')
```

também podemos buscar pelas colunas com o comando

```
df.columns
```

Aqui temos como repostas os identificadores de cada coluna.

```
Index(['W', 'X', 'Y', 'Z'], dtype='object')
```

Existe um método chamado `reset_index` que podemos usar para redefinir os índices. Vamos visualizar como ele funciona primeiro e depois tentar entender.

```
df.reset_index()
```

	index	W	X	Y	Z
0	A	-0.229117	0.087277	0.337653	-0.309933
1	B	-0.489985	0.907704	-0.196438	-0.199404
2	C	0.219143	-1.973148	-0.691878	0.498034
3	D	-1.208147	0.550798	0.836310	-0.678003

	index	W	X	Y	Z
4	E	-0.028455	1.507470	-1.873436	0.642079

Observe que agora nossa tabela tem uma coluna "index" que não existia antes. Os valores dessa coluna são justamente os valores dos índices que definimos quando criamos o nosso DataFrame há alguns tópicos atrás. Então, o método estudado cria uma nova coluna no DataFrame chamada "index", contendo os índices anteriores como dados, e recria os índices como uma sequência numérica simples.

Quando executamos um método do Pandas e obtemos um DataFrame de retorno, é provável que as alterações realizadas por essa função não foram aplicadas na estrutura original, mas sim em uma cópia que ele cria e nos retorna com as alterações solicitadas. Para alterar na estrutura original, é necessário usar o `inplace=True` como parâmetro.

```
df.reset_index(inplace=True)
```

Quando executado, ele não retorna imediatamente o DataFrame. Para visualizá-lo, executamos um `df`, e o que obtemos agora é a estrutura do DataFrame original alterado.

Podemos reverter para a condição original usando outro método.

```
df.set_index("index",inplace=True)
```

Precisamos passar a coluna "index" como primeiro parâmetro para especificar qual era a condição original e usar o `inplace` para que as alterações sejam feitas na estrutura original.

index	W	X	Y	Z
A	-0.229117	0.087277	0.337653	-0.309933
B	-0.489985	0.907704	-0.196438	-0.199404
C	0.219143	-1.973148	-0.691878	0.498034
D	-1.208147	0.550798	0.836310	-0.678003
E	-0.028455	1.507470	-1.873436	0.642079

Podemos substituir os índices por uma das colunas do DataFrame e até mesmo por uma nova lista. Vamos criar uma lista.

```
novoind = 'CA NY WY OR CO'.split()
```

```
['CA', 'NY', 'WY', 'OR', 'CO']
```

E agora vamos definir esses valores como os novos índices. Para isso, precisamos nos certificar de que a lista `novoind` tenha o mesmo tamanho da coluna no DataFrame. Caso contrário, o Pandas não conseguirá associar cada valor.

```
df["novo_index"] = novoind
```

	W	X	Y	Z	novo_index
index					
A	-0.229117	0.087277	0.337653	-0.309933	CA
B	-0.489985	0.907704	-0.196438	-0.199404	NY
C	0.219143	-1.973148	-0.691878	0.498034	WY
D	-1.208147	0.550798	0.836310	-0.678003	OR
E	-0.028455	1.507470	-1.873436	0.642079	CO

Perceba que o Pandas cria uma nova coluna com a lista. Precisamos configurá-la para ocupar a posição dos índices. Para isso, podemos usar o método `set_index`.

```
df.set_index("novo_index")
```

	W	X	Y	Z
novo_index				
CA	-0.229117	0.087277	0.337653	-0.309933
NY	-0.489985	0.907704	-0.196438	-0.199404
WY	0.219143	-1.973148	-0.691878	0.498034
OR	-1.208147	0.550798	0.836310	-0.678003
CO	-0.028455	1.507470	-1.873436	0.642079

Como podemos observar, o `set_index` coloca os valores de `novo_index` para fora, mas está deletando a coluna "index" e não é isso que desejamos. Para resolver isso, vamos combinar dois métodos.

```
df.reset_index().set_index("novo_index")
```

Até `df.reset_index()`, ele pega o índice, cria uma coluna e coloca esses valores, salvando no DataFrame. Só depois ele define o novo índice. Isso preserva a estrutura do nosso DataFrame, e essas configurações você realiza de acordo com a maneira que deseja manipular o DataFrame.

06. Índices multiníveis

Hierarquia de índices e índices múltiplos

Níveis de Índice

```
outside = ['G1', 'G1', 'G1', 'G2', 'G2', 'G2']
inside = [1,2,3,1,2,3]

hier_index = list(zip(outside, inside))
```

Se executarmos hier_index, recebemos uma lista contendo os valores das nossas estruturas de dados combinados e agrupados em pares dentro de tuplas.

```
[('G1', 1), ('G1', 2), ('G1', 3), ('G2', 1), ('G2', 2), ('G2', 3)]
```

É isso que o método zip faz: ele encaixa as duas listas e nos devolve em uma tupla.

Vamos usar essa lista de tuplas para criar um elemento de MultiIndex com Pandas.

```
hier_index = pd.MultiIndex.from_tuples(hier_index)
```

Agora, quando executamos `hier_index`, podemos observar que o índice múltiplo é uma nova disposição da lista de tuplas.

```
MultiIndex([('G1', 1),
            ('G1', 2),
            ('G1', 3),
            ('G2', 1),
            ('G2', 2),
            ('G2', 3)],
           )
```

Vamos criar um Dataframe usando esses índices.

```
df = pd.DataFrame(np.random.randn(6,2), index=hier_index,columns=['A', 'B'])
```

Com `np.random.randn(6,2)`, estamos criando um DataFrame com seis linhas e duas colunas que serão preenchidas com valores numéricos aleatórios. O índice será o MultiIndex que criamos com as duas colunas A e B.

		A	B
G1	1	-0.190968	-1.363914
	2	0.239147	0.949675
	3	1.152059	-1.236651

		A	B
G2	1	1.481911	-0.368843
	2	-0.319206	-0.716082
	3	0.719545	-0.226778

O DataFrame gerado tem duas camadas de índice. Isso pode ser interpretado pensando no exemplo de uma análise feita sobre compras em filiais de uma rede de supermercado para um público específico. Então, a primeira camada de índice indicaria os dados de compras por mulheres, G1 e G2, enquanto a segunda camada de índice identificaria cada filial.

Isso é bastante útil em situações em que faz sentido ter duas camadas de índices, como no exemplo simples citado acima. É necessário se familiarizar com mais essa funcionalidade que o Pandas nos oferece. Nós podemos manipular esse DataFrame assim como vimos anteriormente.

```
df.loc["G1"]
```

Selecionamos apenas os dados que correspondem ao índice G1 com `loc`.

	A	B
1	-0.190968	-1.363914
2	0.239147	0.949675
3	1.152059	-1.236651

Por ele buscar pelo primeiro índice, dessa forma, se na nossa busca passamos um dos índices interno `df.loc[1]`, resultaria em um erro. Para fazer essa busca dar certo, precisamos fazer uma busca combinada, passando primeiro um filtro com o primeiro índice e depois um com o segundo índice.

```
df.loc["G1"].loc[1]
```

Ele faz a busca com `loc[1]` no DataFrame resultado do primeiro filtro `df.loc["G1"]`.

```
A   -0.190968
B   -1.363914
Name: 1, dtype: float64
```

Pensando nisso, podemos aplicar outros métodos a esse DataFrame.

Podemos pesquisar os índices no DataFrame, e ele mostrará o mesmo MultiIndex que criamos antes.

```
MultiIndex([('G1', 1),
            ('G1', 2),
            ('G1', 3),
            ('G2', 1),
            ('G2', 2),
            ('G2', 3)],
           )
```

Observando nosso DataFrame de índice múltiplo, sabemos que as colunas de índices não possuem nome.

		A	B
G1	1	-0.190968	-1.363914
	2	0.239147	0.949675
	3	1.152059	-1.236651
G2	1	1.481911	-0.368843
	2	-0.319206	-0.716082
	3	0.719545	-0.226778

É possível dar nome a elas da seguinte maneira:

```
df.index.names = ['Grupo', 'Número']
```

Imprimindo o DataFrame agora, temos ambas as colunas com os nomes que passamos, respectivamente.

		A	B
Grupo	Número		
G1	1	-0.190968	-1.363914
	2	0.239147	0.949675
	3	1.152059	-1.236651
G2	1	1.481911	-0.368843
	2	-0.319206	-0.716082
	3	0.719545	-0.226778

E agora vamos entender a importância de dar um nome para a coluna de cada índice.

Para trabalhar com essa hierarquia de índices, vamos usar o método `xs`, que permite fazer seleções semelhantes às que fizemos com `loc`. A diferença é que com ele podemos acessar diretamente os índices do segundo nível no DataFrame original, se especificarmos este. Especificamos o nível passando o nome da coluna, por isso é importante nomear cada nível de índice.

```
df.xs(1, level='Número') #procura todos os dados de índice 1 no level número
```

Pega todos os elementos com índice 1 para os dois níveis, sem descartar nenhuma informação.

	A	B
Grupo		
G1	-0.190968	-1.363914
G2	1.481911	-0.368843

Pegando informações com índices do primeiro nível.

```
df.xs('G1')
```

	A	B
Número		
1	-0.190968	-1.363914
2	0.239147	0.949675
3	1.152059	-1.236651

07. Tratamento de dados ausentes

Quando trabalhamos com análise de dados, acabamos nos deparando com a necessidade de lidar com a ausência de dados. Isso ocorre porque algumas informações podem não estar disponíveis na base de dados, e é necessário aprender como contornar essa situação para conseguir realizar manipulações.

Neste capítulo, vamos conhecer algumas ferramentas que o Python oferece para lidar com esse tipo de situação. Para começar, vamos recapitular a importação do pandas e numpy.

```
import numpy as np
import pandas as pd
```

E vamos usar um DataFrame simples que tenha alguns valores ausentes. Com o método `np.nan` do NumPy, conseguiremos construir essa estrutura.

```
df = pd.DataFrame({'A': [1,2,np.nan],
                   'B': [5,np.nan,np.nan],
                   'C': [1,2,3]})
```

	A	B	C
0	1.0	5.0	1
1	2.0	NaN	2
2	NaN	NaN	3

Método dropna()

O pandas permite fazer a remoção desses campos e até mesmo preenchê-los. A primeira maneira de realizar isso é com a função `dropna()`. Ele descarta linhas ou colunas que possuem valores ausentes NaN.

```
df.dropna()
```

Dessa forma, ele nos devolve apenas o DataFrame onde linhas e colunas não tenham valores NaN.

	A	B	C
0	1.0	5.0	1

Por padrão, essa função vem com um parâmetro `axis=0`. Isso significa que ela está descartando no eixo zero, ou seja, as linhas. Podemos definir esse parâmetro para que a função aplique isso nas colunas.

```
df.dropna(axis=1)
```

E aqui o resultado será diferente. Ele remove as colunas e devolve apenas aquelas que não possuem valores ausentes.

	C
0	1
1	2
2	3

Outro parâmetro que essa função recebe é o `thresh`, que representa um limite de valores ausentes que o DataFrame deve conter para a função decidir se vai descartar aquela coluna ou linha.

```
df.dropna(axis=1, thresh=2)
```

Com `axis=1` ainda definido para o eixo das colunas e agora passando `thresh=2`, serão eliminadas aquelas colunas que possuem dois ou mais valores ausentes.

	A	C
0	1.0	1
1	2.0	2
2	NaN	3

Para o dataframe com o qual estamos trabalhando, serão retornadas apenas as colunas A e C, que possuem dois ou menos valores ausentes, respectivamente. Esse método é útil quando a ausência de certos dados nos impede de trabalhar. Assim como em outros métodos, o `dropna()` também recebe o parâmetro inplace quando desejamos aplicar modificações na estrutura original.

Método fillna()

O método `fillna` preenche os campos de dados ausentes `NaN`. Podemos passar um valor que desejamos usar para substituir todos os casos de valores ausentes.

```
df.fillna(0)
```

Aqui ele nos retorna o dataframe com os valores `NaN` preenchidos com zero.

	A	B	C
0	1.0	5.0	1
1	2.0	0.0	2
2	0.0	0.0	3

Poderíamos preencher com uma string.

```
df.fillna('Conteúdo')
```

	A	B	C
0	1.0	5.0	1
1	2.0	Conteúdo	2
2	Conteúdo	Conteúdo	3

Com esse método do Pandas, preenchemos esses valores de forma que façam sentido para trabalhar com eles, sem atrapalhar nossas operações. Uma maneira muito importante de substituir esses valores sem afetar nosso processo é usando a média dos valores em uma coluna, por exemplo. Fazendo isso para a coluna A, vamos calcular a média com a função `mean`.

```
df['A'].mean()

#Retorna a média dos valores na coluna A 1 e 2
1.5
```

e podemos preencher com esse valor

```
df['A'].fillna(value=df['A'].mean())

0    1.0
1    2.0
2    1.5
Name: A, dtype: float64
```

Método ffill() Quando trabalhamos com séries temporais, conjuntos de dados ao longo de um índice, é muito usual fazer o preenchimento de valores ausentes com o último valor rastreado na coluna. Este é o dataframe com o qual desejamos trabalhar.

	A	B	C
0	1.0	5.0	1
1	2.0	NaN	2
2	NaN	NaN	3

```
df.ffill()
```

Aqui está o resultado após aplicarmos o método:

	A	B	C
0	1.0	5.0	1
1	2.0	5.0	2
2	2.0	5.0	3

Na coluna A, 2.0 foi o último valor encontrado, e o valor desconhecido a seguir foi preenchido com ele. O mesmo ocorreu para a coluna B, onde 5.0 foi o último valor reconhecido e preencheu o valor seguinte. Em seguida, o mesmo procedimento foi realizado para o último NaN na sequência da coluna.

08. Groupby

Usamos o métdodo Groupby para agrupar dados de acordo com categorias específicas e fazer operações matemáticas, ele vem do SQL que é uma linguagem de manipulação de banco de dados. Nessa seção vamos aprender a usar esse método com pandas.

Vamos começar criando o Dataframe que usaremos para realiazar algumas demonstrações.

```
import pandas as pd
#cria um DataFrame
data = {'Classe': ['Júnior', 'Júnior', 'Pleno', 'Pleno', 'Sênior', 'Sênior'],
'Nome': ['Jorge', 'Carlos', 'Roberta', 'Patrícia', 'Bruno', 'Vera'],
'Venda': [200, 120, 340, 124, 243, 350]}
```

nossa tabela possui três colunas, a coluna classe para o nível de cada vendedor: júnior, pleno e sênior. Nome,com o nome de cada funcionário e a coluna venda, que corresponde ao quanto cada um vendeu ao longo de uma data específica.

	Classe	Nome	Venda
0	Júnior	Jorge	200
1	Júnior	Carlos	120
2	Pleno	Roberta	340
3	Pleno	Patrícia	124
4	Sênior	Bruno	243
5	Sênior	Vera	350

Podemos pensar em algumas análises que podem ser feitas com base nessa tabela, como, por exemplo, qual foi o total de vendas de cada um, qual foi o total de vendas por classe, e isso poderia ser facilmente feito com o Groupby. Quando lidamos com um dataframe muito grande, com a ocorrência de valores repetidos, como um nome que aparece várias vezes ou muitas aparições da mesma classe, seria necessário usar alguma estratégia para agrupar essas informações. No Excel, isso se assemelha a SOMASE, CONT.SE ou MÉDIA.SE, todas as operações que agrupamos e realizamos sempre que uma condição é verificada.

Supomos que desejamos saber qual a média de venda para cada nível de vendedor, para isso vamos aplicar o Groupby,

```
df.groupby("Classe")
```

e a saída pra esse comando é a seguinte informação

```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x7e5262313df0>
```

Ele informa que fez o agrupamento, isso acontece porque o groupby primeiro agrupa, depois aplicamos alguma operação e depois ele expande é por isso que podemos considerar essa uma função custosa.

sum( )

Agora é só usar a função sum a todo o DataFrame agrupado

```
df.groupby("Classe").sum(numeric_only=True)
```

é necessário passar o parâmetro `numeric_only=True` para garantir que apenas colunas numéricas sejam consideradas na operação do grupo, ignorando as colunas não numéricas. Ele pode ser passado diretamente na função sum() ou como segundo parametro no groupby.

Classe	
Júnior	320
Pleno	464
Sênior	59

Como resultado ele nos dá o total de vendas para cada classe de vendedor.

mean( )

Podemos calcular a média de valores da mesma maneira, só que agora usando a função mean( ).

```
group.mean(numeric_only=True)
```

Classe	
Júnior	160.0
Pleno	232.0
Sênior	296.5

Calculando a média de vendas para cada classe de vendedor, podemos ver que os vendedores júnior têm uma média de 160.0 vendas, enquanto vendedores sênior tem uma média de 296.5.

min( )

Com o método `min` é possível obter quem fez o menor valor em vendas.

```
df.groupby("Classe").min()
```

ele retorna um DataFrame

	Nome	Venda
Classe		
Júnior	Carlos	120
Pleno	Patrícia	124
Sênior	Bruno	243

Identificando os vendedores que tiveram o desempenho mais baixo em vendas dentro de cada categoria.

max( )

Dessa maneira podemos aplicar o método `max` que nos retornará apenas os que fizeram os maiores valores em vendas de cada classe.

```
df.groupby("Classe").max()
```

	Nome	Venda
Classe		
Júnior	Jorge	200
Pleno	Roberta	340
Sênior	Vera	350

Olhando para o Dataframe podemos ver que, assim como `min`, ele pega apenas os vendedores que alcançaram o maior desempenho de vendas em suas respectivas categorias, encontrando os valores máximos de vendas para cada classe de vendedor.

Esses métodos de agregação, fazem uma seleção nos valores agrupados por classe e nos devolvem apenas aqueles que correspondem à opreação realizada.

Usando o DataFrame que criamos no ínicou, vamos fazer uma copia e redefinir os valores na coluna venda e a seguir realizaremos umas operações mais avançadas.

	Classe	Nome	Venda
0	Júnior	Jorge	200
1	Júnior	Carlos	120
2	Pleno	Roberta	340
3	Pleno	Patrícia	124
4	Sênior	Bruno	243
5	Sênior	Vera	350

```
df2 = df.copy() #df2 é uma copia de df
```

reatribuindo os valores na coluna Venda,

```
df2["Venda"] = [150, 432, 190, 230, 410, 155]
```

	Classe	Nome	Venda
0	Júnior	Jorge	150
1	Júnior	Carlos	432
2	Pleno	Roberta	190
3	Pleno	Patrícia	230
4	Sênior	Bruno	410
5	Sênior	Vera	155

vamos concatenar os dois DataFrame em um df3 usando o método `concat( )`, estudaremos essa função com mais detalhes em breve.

```
df3 = pd.concat([df, df2])
```

	Classe	Nome	Venda
0	Júnior	Jorge	200
1	Júnior	Carlos	120

	Classe	Nome	Venda
2	Pleno	Roberta	340
3	Pleno	Patrícia	124
4	Sênior	Bruno	243
5	Sênior	Vera	350
0	Júnior	Jorge	150
1	Júnior	Carlos	432
2	Pleno	Roberta	190
3	Pleno	Patrícia	230
4	Sênior	Bruno	410
5	Sênior	Vera	155

Após a combinação dos DataFrames `df3`, nos deparamos com linhas duplicadas e uma visão abrangente das vendas, o que possibilita novas análises. Utilizamos o método `groupby()` para agrupar essas informações, passando não apenas uma, mas duas colunas. Isso faz sentido aqui, visto que temos duas ocorrências para um mesmo vendedor.

```
df3.groupby(["Classe", "Nome"]).sum()
```

		Venda
Classe	Nome	
Júnior	Carlos	552
	Jorge	350
Pleno	Patrícia	354
	Roberta	530
Sênior	Bruno	653
	Vera	505

09. Merge, Concat, Join

Agora, vamos conhecer alguns métodos uteis para combinar DataFrames e para isso definiremos três estruturas.

```
df1 = pd.DataFrame({'A': ['A0', 'A1','A2', 'A3'],
                    'B': ['B0', 'B1', 'B2', 'B3'],
                    'C': ['C0', 'C1', 'C2', 'C3'],
                    'D': ['D0', 'D1', 'D2', 'D3']},
                    index=[0,1,2,3])
```

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3

```
df2 = pd.DataFrame({'A': ['A4', 'A5','A6', 'A7'],
                    'B': ['B4', 'B5', 'B6', 'B7'],
                    'C': ['C4', 'C5', 'C6', 'C7'],
                    'D': ['D4', 'D5', 'D6', 'D7']},
                    index=[4,5,6,7])
```

	A	B	C	D
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7

```
df3 = pd.DataFrame({'A': ['A8', 'A9','A10', 'A11'],
                    'B': ['B8', 'B9', 'B10', 'B11'],
                    'C': ['C8', 'C9', 'C10', 'C11'],
                    'D': ['D8', 'D9', 'D10', 'D11']},
                    index=[8,9,10,11])
```

	A	B	C	D
8	A8	B8	C8	D8

	A	B	C	D
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

Iremos usar três métodos diferentes para combinar esses DataFrames e observar o que acontece em cada caso.

Concat( )

Esse método permite concatenar objetos ao longo de um eixo específico, por padrão, isso é feito ao longo do eixo 0, ou seja, ao longo das linhas, o que significa que os objetos serão empilhados verticalmente.

```
pd.concat([df1, df2, df3])
```

como podemos observar a seguir

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1
2	A2	B2	C2	D2
3	A3	B3	C3	D3
4	A4	B4	C4	D4
5	A5	B5	C5	D5
6	A6	B6	C6	D6
7	A7	B7	C7	D7
8	A8	B8	C8	D8
9	A9	B9	C9	D9
10	A10	B10	C10	D10
11	A11	B11	C11	D11

é bastante comum o uso desse módulo, porque em muitos casos o agrupamento necessário é um simples agrupamento dos dados um após o outro. Definindo parâmetro padrão `axis` para 1 teremos o agrupamento horizontal.

```
pd.concat([df1, df2, df3], axis=1)
```

	A	B	C	D	A	B	C	D	A	B	C	D
0	A0	B0	C0	D0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1	A1	B1	C1	D1	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2	A2	B2	C2	D2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
3	A3	B3	C3	D3	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
4	NaN	NaN	NaN	NaN	A4	B4	C4	D4	NaN	NaN	NaN	NaN
5	NaN	NaN	NaN	NaN	A5	B5	C5	D5	NaN	NaN	NaN	NaN
6	NaN	NaN	NaN	NaN	A6	B6	C6	D6	NaN	NaN	NaN	NaN
7	NaN	NaN	NaN	NaN	A7	B7	C7	D7	NaN	NaN	NaN	NaN
8	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	A8	B8	C8	D8
9	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	A9	B9	C9	D9
10	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	A10	B10	C10	D10
11	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	A11	B11	C11	D11

eles são concatenados com as colunas sendo adicionadas ao lado, na horizontal. Isso acontece porque o Pandas tenta alinhar as linhas dos DataFrames de acordo com seus índices antes de concatená-las, dessa maneira os DataFrames têm diferentes índices e não correspondem completamente, e ele usa NaN para preencher os campos onde não há correspondencia entre índices. Então, se você precisar trabalhar com Dataframes que compartilham linhas mantém `axis=0` se elas compartilham colinas `axis=1`.

Merge( )

Nessa demonstração usaremos dois Dataframes diferentes, mas que compartilham uma única coluna com nome em comum, a coluna key.

```
esquerda = pd.DataFrame({'key': ['K0', 'K1', 'K2', 'K3'],
                         'A': ['A0', 'A1', 'A2', 'A3'],
                         'B': ['B0', 'B1', 'B2', 'B3']})
```

key	A	B
0	K0	A0
1	K1	A1
2	K2	A2
3	K3	A3

```
direita = pd.DataFrame({'key': ['K0', 'K1', 'K2', 'K3'],
                        'C': ['C0', 'C1', 'C2', 'C3'],
                        'D': ['D0', 'D1', 'D2', 'D3']})
```

key	C	D
0	K0	C0
1	K1	C1
2	K2	C2
3	K3	C3

O método `merge` vai nos ajudar a encaixar esses dados e, de alguma forma, utilizar algumas dessas colunas como referência para realizar essa combinação.

```
pd.merge(esquerda, direita, on='key')
```

Precisamos definir um valor para o parâmetro on, que indica onde essa operação deve ser realizada. Nesse caso, estamos dizendo ao Pandas para combinar os DataFrames esquerda e direita usando a coluna “key” como chave de junção.

	key	A	B	C	D
0	K0	A0	B0	C0	D0
1	K1	A1	B1	C1	D1
2	K2	A2	B2	C2	D2
3	K3	A3	B3	C3	D3

O método `merge` verifica os valores na coluna “key” de ambos os DataFrames e junta as linhas onde esses valores forem iguais. Ele combina as linhas e os valores das colunas são sobrepostos, resultando em um novo DataFrame combinado.

```
esquerda = pd.DataFrame({'key1': ['K0', 'K0', 'K1', 'K2'],
                         'key2': ['K0', 'K1', 'K0', 'K1'],
                         'A': ['A0', 'A1', 'A2', 'A3'],
                         'B': ['B0', 'B1', 'B2', 'B3']})
```

	key1	key2	A	B
0	K0	K0	A0	B0
1	K0	K1	A1	B1
2	K1	K0	A2	B2
3	K2	K1	A3	B3

```
direita = pd.DataFrame({'key1': ['K0', 'K1', 'K1', 'K2'],
                         'key2': ['K0', 'K0', 'K0', 'K0'],
                         'C': ['C0', 'C1', 'C2', 'C3'],
                         'D': ['D0', 'D1', 'D2', 'D3']})
```

	key1	key2	A	B
0	K0	K0	C0	D0
1	K1	K0	C1	D1
2	K1	K0	C2	D2
3	K2	K0	C3	D3

Misturamos esses DataFrames passando as colunas ‘key1’ e ‘key2’.

```
pd.merge(esquerda, direita, on=['key1', 'key2'])
```

O método vai usar o DataFrame `esquerda` como referência para mesclar o outro. Ele vai procurar no DataFrame `direita` pares sequenciais das colunas `key1` e `key2` que correspondam à mesma sequência do DataFrame `esquerda`.

	key1	key2	A	B	C	D
0	K0	K0	A0	B0	C0	D0
1	K1	K0	A2	B2	C1	D1
2	K1	K0	A2	B2	C2	D2

Parâmetro how: inner, outer, right e left.

Outro parâmetro que podemos passar é o how. Com ele, vamos dizer ao método como queremos que essa operação seja feita. Por padrão, o valor de how é definido como `inner`, onde é necessária a ocorrência do primeiro DataFrame no segundo para mesclar os valores, mas podemos passar `outer`.

```
pd.merge(esquerda, direita, how='outer', on=['key1', 'key2'])
```

Isso realizará uma junção externa entre os DataFrames esquerda e direita, incluindo todas as linhas de ambos os DataFrames no DataFrame resultante. Quando não há correspondência nas chaves de junção, o Pandas preencherá as células com valores NaN.

	key1	key2	A	B	C	D
0	K0	K0	A0	B0	C0	D0
1	K0	K1	A1	B1	NaN	NaN
2	K1	K0	A2	B2	C1	D1
3	K1	K0	A2	B2	C2	D2
4	K2	K1	A3	B3	NaN	NaN
5	K2	K0	NaN	NaN	C3	D3

```
pd.merge(esquerda, direita, how="right", on=['key1', 'key2'])
```

Com how=`"right"`, o Pandas realizará a junção à direita. Todos os valores do DataFrame da direita, o segundo que passamos, serão mantidos no DataFrame resultante independentemente de haver correspondência nas chaves de junção.

	key1	key2	A	B	C	D
0	K0	K0	A0	B0	C0	D0
1	K1	K0	A2	B2	C1	D1
2	K1	K0	A2	B2	C2	D2
3	K2	K0	NaN	NaN	C3	D3

```
pd.merge(esquerda, direita, how="left", on=['key1', 'key2'])
```

E com how=`"left"`, estamos garantindo que todas as linhas do DataFrame da esquerda sejam incluídas no DataFrame resultante, independentemente de haver correspondência nas chaves de junção. As

linhas correspondentes do DataFrame da direita serão incluídas quando houver correspondência, e os valores NaN serão usados quando não houver correspondência.

	key1	key2	A	B	C	D
0	K0	K0	A0	B0	C0	D0
1	K0	K1	A1	B1	NaN	NaN
2	K1	K0	A2	B2	C1	D1
3	K1	K0	A2	B2	C2	D2
4	K2	K1	A3	B3	NaN	NaN

Join( )

O método join() é uma alternativa conveniente ao merge() quando queremos combinar DataFrames com base nos índices.

```
esquerda = pd.DataFrame({'A': ['A0', 'A1','A2'],
                    'B': ['B0', 'B1', 'B2']},
                    index=['K0', 'K1', 'K2'])
```

	A	B
K0	A0	B0
K1	A1	B1
K2	A2	B2

```
direita = pd.DataFrame({'C': ['C0', 'C2','C3'],
                    'D': ['D0', 'D2', 'D3']},
                    index=['K0', 'K2', 'K3'])
```

	C	D
K0	C0	D0
K2	C2	D2
K3	C3	D3

para acessar o join aplicamos diretamente no DataFrame.

```
esquerda.join(direita)
```

	A	B	C	D
K0	A0	B0	C0	D0
K1	A1	B1	NaN	NaN
K2	A2	B2	C2	D2

O resultado final contém todas as linhas do DataFrame esquerda e as colunas adicionadas do DataFrame direita. Quando não há correspondência para um índice específico, as células correspondentes nas novas colunas são preenchidas com valores NaN.

10. Operações com DataFrames

Agora vamos conhecer algumas operações que o Pandas nos dá para realizar com DataFrames, além das que estudamos até aqui.

```
import pandas as pd

df = pd.DataFrame({'col1': [1,2,3,4], 'col2':[444, 555, 666, 444], 'col3':['abc', 'def',
↪  'ghi', 'xyz']})


df.head()
```

	col1	col2	col3
0	1	444	abc
1	2	555	def
2	3	666	ghi
3	4	444	xyz

info( )

Esse método analisa a estrutura de um DataFrame e os tipos de dados nele.

```
df.info()
```

observe que ele identica o tipo de dados existente coluna a coluna.

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4 entries, 0 to 3
Data columns (total 3 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   col1    4 non-null      int64
 1   col2    4 non-null      int64
 2   col3    4 non-null      object
dtypes: int64(2), object(1)
memory usage: 224.0+ bytes
```

Este é um método importante que auxilia na compreensão da estrutura do DataFrame, na identificação de problemas de tipos de dados e na gestão do uso de memória. Essa prática facilita a otimização e a manipulação eficiente dos dados.

memory_usage( )

Este método fornece o uso de memória do DataFrame em bytes para cada uma de suas colunas. Tal informação pode ser útil para compreender como os dados ocupam espaço na memória e para identificar oportunidades de otimização, especialmente ao lidar com DataFrames muito grandes.

```
df.memory_usage()
```

ele retorna uma serie indicando a memoria ocupada por cada coluna.

```
Index    128
col1      32
col2      32
col3      32
dtype: int64
```

reparem que a coluna index ocupa 128 bytes, a maior memória ocupada, porque o índice é um objeto separado que armazena rótulos de índice para cada linha do DataFrame. Se o índice for do tipo object e contiver strings longas ou únicas, isso também pode aumentar o uso de memória, pois as strings precisam ser armazenadas de forma completa.

Informações sobre valores exclusivos

	col1	col2	col3
0	1	444	abc
1	2	555	def
2	3	666	ghi
3	4	444	xyz

unique( )

Ele nos retorna apenas valores únicos na coluna especificada

```
df["col2"].unique()
```

```
array([444, 555, 666])
```

nunique( )

Retora quantos valores únicos existem

```
df["col2"].nunique()
```

```
3
```

value_counts( )

Esse método faz a contagem da quantidade que um valor aparece na coluna

```
df["col2"].value_counts()
```

```
col2
444    2
555    1
666    1
Name: count, dtype: int64
```

Aplicando funções

Em análise de dados muitas vezes desejamos aplicar funções matemáticas a colunas específicas. Supondo que desejamos aplicar x^2, primeiro criamos a função em python.

```
def comp(x):
    return x** 2 + 3
```

agora passamos a coluna que desejamos e com o método `apply` passamos a função que definimos por parâmetro.

```
df["col1"].apply(comp)
```

ele nos retorna uma serie com os resultados da função naquela coluna.

```
0     4
1     7
2    12
3    19
Name: col1, dtype: int64
```

podemos gravar os resultados em uma coluna do DataFrame, atribuindo o valor por uma string, como já estudamos.

```
df["col1_calc"] = df["col1"].apply(comp)
```

	col1	col2	col3	cal1_calc
0	1	444	abc	4
1	2	555	def	7
2	3	666	ghi	12
3	4	444	xyz	19

lambda

O lambda é uma função anônima que permite criar funções simples de forma rápida e concisa. Ela facilita o uso de funções que precisamos implementar.

```
df["col1_calc"] = df["col1"].apply(lambda x: x** 2 + 3)
```

isso retorna o mesmo DataFrame de antes.

	col1	col2	col3	cal1_calc
0	1	444	abc	4
1	2	555	def	7
2	3	666	ghi	12
3	4	444	xyz	19

Operações Matemáticas Simples

Vamos ver algumas funções numéricas simples que podemos aplicar sobre todo o DataFrame.

soma

```
df["col1"].sum()
```

a soma dos valores na col1

```
10
```

média

calculo da média dos valores de uma coluna

```
df["col1"].mean()
```

```
2.5
```

Produto

O produto dos valores de uma coluna

```
df["col1"].product()
```

```
24
```

Desvio padrão

```
df["col1"].std()
```

```
1.2909944487358056
```

Máximo

Maior valor na coluna

```
df["col1"].max()
```

```
4
```

Mínimo

O menor valor na coluna

```
df["col1"].min()
```

```
1
```

Index com maior valor

Também é possivél encontrar o index que tem o valor máximo

```
df["col1"].idxmax()
```

```
3 #o maior valor está nesse índice
```

Combinando operações com filtros

	col1	col2	col3	cal1_calc
0	1	444	abc	4
1	2	555	def	7
2	3	666	ghi	12
3	4	444	xyz	19

Vamos supor que desejamos fazer o somatorio da 'col1' sempre que o valor da 'col2' for igual a 444, fariamos isso da seguinte forma,

```
df[df["col2"] == 444]
```

fizemos o filtro na 'col2' apenas quando essa tiver valor igual a 444

	col1	col2	col3	col1_calc
0	1	444	abc	4
3	4	444	xyz	19

e aplicamos `sum` na 'col1' em cima desse recorte

```
df[df["col2"] == 444]["col1"].sum()
```

```
5
```

ordenamento

É necessario passar o parametro `by` que indica qual coluna ele deve usar como referencia para realizar o ordenamento.

```
df.sort_values(by="col2")
```

os valores foram ordenados crescentemente seguindo a 'col2' como referencia.

	col1	col2	col3	col1_calc
0	1	444	abc	4

	col1	col2	col3	col1_calc
3	4	444	xyz	19
1	2	555	def	7
2	3	666	ghi	12

Para os últimos métodos que veremos aqui vamos criar um novo DataFrame.

```
data = {'A':['foo', 'foo', 'foo', 'bar', 'bar', 'bar'],
        'B':['one', 'one', 'two', 'two', 'one', 'one'],
        'C': ['x', 'y', 'x', 'y', 'x', 'y'],
        'D':[1,3,2,5,4,1]}

df = pd.DataFrame(data)
```

	A	B	C	D
0	foo	one	x	1
1	foo	one	y	3
2	foo	two	x	2
3	bar	two	y	5
4	bar	one	x	4
5	bar	one	y	1

Mapeando valores

Imagine que desejamos fazer o mapeamento dos valores nesse DataFrame, para isso usaremos o método `map`. Esse método precisa de valores para se guiar, vamos criar um dicionário com o valor que vamos buscar e o valor que deve substitui-lo.

```
dict_map = {"one": "1" , "two": "2"}
```

vamos salvar em uma nova coluna o resultado do mapeamento

```
df["E"] = df["B"].map(dict_map)
```

	A	B	C	D	E
0	foo	one	x	1	1
1	foo	one	y	3	1
2	foo	two	x	2	2
3	bar	two	y	5	2
4	bar	one	x	4	1
5	bar	one	y	1	1

O mapeamento pode conter quantos valores desejar para sua análise.

pivot_table

É uma função em pandas que permite reorganizar e resumir os dados de um DataFrame, criando uma tabela dinâmica (pivot table) semelhante às que são comuns em softwares de planilha, como o Microsoft Excel.

Vamos criar um novo DataFrame onde tenhamos una segregação da coluna 'A' e da coluna 'B', olhando para os valores de 'D'.

```
df.pivot_table(index="A", columns="B", values="D")
```

B	one	two
A		
bar	2.5	5.0
foo	2.0	2.0

As linhas da tabela são determinadas pelos valores únicos da coluna 'A', as colunas são determinadas pelos valores únicos da coluna 'B' e os valores na tabela são calculados a partir dos valores da coluna 'D'.

11. Series Temporais no Pandas

Em Pandas, uma série temporal é uma estrutura de dados unidimensional que contém valores associados a instantes de tempo específicos. Cada valor na série temporal está associado a um índice de tempo, que pode ser uma data, hora, timestamp ou outro tipo de marca temporal. É muito útil quando trabalhamos com dados em produção em tempo real, elas permitem monitorar e entender o comportamento desses dados ao longo do tempo.

Dados Temporais

```
numero_de_dias = 100
```

```
datas = pd.date_range(start='1/1/2021', periods=numero_de_dias)
```

ele preenche com valores iniciando em primeiro de janeiro de 2021 até cem dias depois.

```
DatetimeIndex(['2021-01-01', '2021-01-02', '2021-01-03', '2021-01-04',
               '2021-01-05', '2021-01-06', '2021-01-07', '2021-01-08',
               '2021-01-09', '2021-01-10', '2021-01-11', '2021-01-12',
               '2021-01-13', '2021-01-14', '2021-01-15', '2021-01-16',
               '2021-01-17', '2021-01-18', '2021-01-19', '2021-01-20',
               '2021-01-21', '2021-01-22', '2021-01-23', '2021-01-24',
               '2021-01-25', '2021-01-26', '2021-01-27', '2021-01-28',
               '2021-01-29', '2021-01-30', '2021-01-31', '2021-02-01',
               '2021-02-02', '2021-02-03', '2021-02-04', '2021-02-05',
               '2021-02-06', '2021-02-07', '2021-02-08', '2021-02-09',
               '2021-02-10', '2021-02-11', '2021-02-12', '2021-02-13',
               '2021-02-14', '2021-02-15', '2021-02-16', '2021-02-17',
               '2021-02-18', '2021-02-19', '2021-02-20', '2021-02-21',
               '2021-02-22', '2021-02-23', '2021-02-24', '2021-02-25',
               '2021-02-26', '2021-02-27', '2021-02-28', '2021-03-01',
               '2021-03-02', '2021-03-03', '2021-03-04', '2021-03-05',
               '2021-03-06', '2021-03-07', '2021-03-08', '2021-03-09',
               '2021-03-10', '2021-03-11', '2021-03-12', '2021-03-13',
               '2021-03-14', '2021-03-15', '2021-03-16', '2021-03-17',
               '2021-03-18', '2021-03-19', '2021-03-20', '2021-03-21',
               '2021-03-22', '2021-03-23', '2021-03-24', '2021-03-25',
               '2021-03-26', '2021-03-27', '2021-03-28', '2021-03-29',
               '2021-03-30', '2021-03-31', '2021-04-01', '2021-04-02',
               '2021-04-03', '2021-04-04', '2021-04-05', '2021-04-06',
               '2021-04-07', '2021-04-08', '2021-04-09', '2021-04-10'],
              dtype='datetime64[ns]', freq='D')
```

vamos usar o 'datas' para preencher o indice de um DataFrame arbitrario, para que a gente simule algumas funções.

```
df = pd.DataFrame(range(numero_de_dias), columns=["number"], index= datas)
```

o range recebe 'valores_de_dias' para criar valores de zero até cem e o index será preenchido com datas.

	number
2021-01-01	0
2021-01-02	1
2021-01-03	2
2021-01-04	3
2021-01-05	4
…	…
2021-04-06	95
2021-04-07	96
2021-04-08	97
2021-04-09	98
2021-04-10	99

100 rows × 1 columns

se acessarmos o indice obrevaremos que o tipo dele é `Datetime`. Acessando um valor individualmente obtemos informações sobre a data.

```
df.index[0]
```

```
Timestamp('2021-01-01 00:00:00', freq='D')
```

o ano, mês, dia e a hora. A partir disso podemos usar algumas funções e extrair algumas informações dele.

```
df.index[0].day #pega o dia

#saída
1
```

```
df.index[0].month #pega o mês

#saída
1
```

```
df.index[0].year #pega o ano

#saída
2021
```

```
df.index[0].hour #pega a hora

#saída
0
```

Seleção

Podemos fazer seleções com esses DataFrames.

```
df[df.index.month==1]
```

Com esse comando selecionamos apenas os valores quando o mês for 1, ou seja, apenas as datas de Janeiro.

	number
2021-01-01	0
2021-01-02	1
2021-01-03	2
2021-01-04	3
2021-01-05	4
2021-01-06	5
2021-01-07	6
2021-01-08	7
2021-01-09	8
2021-01-10	9
2021-01-11	10
2021-01-12	11
2021-01-13	12
2021-01-14	13
2021-01-15	14
2021-01-16	15
2021-01-17	16
2021-01-18	17

	number
2021-01-19	18
2021-01-20	19
2021-01-21	20
2021-01-22	21
2021-01-23	22
2021-01-24	23
2021-01-25	24
2021-01-26	25
2021-01-27	26
2021-01-28	27
2021-01-29	28
2021-01-30	29
2021-01-31	30

Puxar todos os valores que sejam dia dez.

```
df[df.index.day==10]
```

	number
2021-01-10	9
2021-02-10	40
2021-03-10	68
2021-04-10	99

podemos criar uma coluna ‘Month’ e salvar nela apenas os valores dos meses.

```
df["Month"] = df.index.month
```

	number	Month
2021-01-01	0	1
2021-01-02	1	1
2021-01-03	2	1
2021-01-04	3	1
2021-01-05	4	1
…	…	…
2021-04-06	95	4
2021-04-07	96	4
2021-04-08	97	4
2021-04-09	98	4
2021-04-10	99	4

100 rows × 2 columns

Portanto, primeiro, importamos essa biblioteca em nosso código com a linha import datetime. Isso nos permite usar todas as funcionalidades dela.

```
import datetime
```

Digamos que queremos selecionar apenas as linhas onde a data seja posterior a uma data específica, por exemplo, depois de 10 de janeiro de 2021.

```
df[df.index > datetime.datetime(2021, 1, 10)]
```

	number	Month
2021-01-11	10	1
2021-01-12	11	1
2021-01-13	12	1
2021-01-14	13	1
2021-01-15	14	1
…	…	…
2021-04-06	95	4

	number	Month
2021-04-07	96	4
2021-04-08	97	4
2021-04-09	98	4
2021-04-10	99	4

90 rows × 2 columns

Então, isso nos dá como resultado um novo DataFrame contendo apenas as linhas onde a data é posterior a 10 de janeiro de 2021. Isso é útil quando queremos trabalhar com dados para um período específico.

12. Entrada e Saída de Dados

No Pandas, temos uma variedade de métodos que nos permitem importar dados de várias fontes e exportar nossos dados para outros locais. Esses métodos são essenciais para manipular dados de forma eficaz em nossos projetos de análise de dados.

read_csv( )

Arquivos CSV (Comma-Separated Values) são um formato comum de armazenamento de dados tabulares. Eles consistem em linhas de texto onde cada linha representa uma linha de dados na tabela e os valores são separados por vírgulas (ou outro delimitador, como ponto e vírgula).

Aqui vamos abrir o arquivo `exemplo.csv` e o primeiro parâmentro a se passar no comando `read_csv` é o caminho do arquivo que desejamos abrir, nesse caso ele está no mesmo diretório do meu código python. Com o parâmetro `sep` indica qual o separador está sendo usado para fragmentar os dados e por ultimo vamos especificar qual o separador decimal, isso informa como os valores decimais estão pontuados, por vírgula ou ponto.

```
df1 = pd.read_csv("exemplo.csv", sep=";", decimal=".")
df1
```

	0,1,2,3
0	-0.3948134131725716,-0.6946298549534513,0.2533…
1	-2.134216836009854,1.1413807607461153,-0.13014…
2	-0.2237544807445792,0.36513315402714586,-0.104…
3	0.10806554946644498,0.3313317379324355,0.05370…
4	0.08528967423746449,0.7408341891386678,-0.5969…

Observe que nesse retorno os dados estão desorganizados e confusos. Isso pode ser um indicador de que o separador desses dados não é um ‘;’ como definido e sim uma vírgula ‘,’.

```
df1 = pd.read_csv("exemplo.csv", sep=",", decimal=".")
df1
```

agora sim, nosso DataFrame faz sentido.

	0	1	2	3
0	-0.394813	-0.694630	0.253393	0.602590
1	-2.134217	1.141381	-0.130142	0.690936
2	-0.223754	0.365133	-0.104233	-0.182788
3	0.108066	0.331332	0.053708	-0.280468
4	0.085290	0.740834	-0.596941	0.243950

é uma boa prática as informações dos dados da tabela criada com o método `.info()`, pra garantir que o tipo ao qual as colunas foram associadas funcionou, algumas vezes o pandas pode interpretar todos os valores como string.

```
df1.info()
```

aqui tudo ocorreu como o esperado, os dados de cada coluna são interpretados como um número `float`.

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 4 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   0       5 non-null      float64
 1   1       5 non-null      float64
 2   2       5 non-null      float64
 3   3       5 non-null      float64
dtypes: float64(4)
memory usage: 288.0 bytes
```

Sempre que você for trabalhar com o método `read_csv()` ler sua documentação pode ser a melhor forma de encontrar funcionalidades e especificações para o que deseja realizar, que possa ajudar a manipular melhor seu conjunto de dados.

to_csv( )

O método `to_csv()` é utilizado para exportar os dados de um DataFrame do Pandas para um arquivo CSV. Com esse método, podemos salvar os dados que manipulamos e analisamos em um formato de arquivo que pode ser facilmente compartilhado, armazenado ou utilizado por outros sistemas.

Vamos passar os dados do DataFrame df1 para um novo arquivo.

```
#mudando o separador decimal para uma vírgula
df1.to_csv("exemplo2.csv", sep=",", decimal=",")
```

agora, quando você atualizar o diretório em que está trabalhando o arquivo `exemplo2.csv` também aparecerá no local.

Excel

O pandas também permite realizar essas funcionalidades com arquivos do excel, entretando, se faz uso de algumas bibliotecas auxiliar.

```
#bibliotecas necessárias para trabalhar com excel
pip install xlrd
pip install openpyxl
```

Entrada via Excel

em aquivos desse formato podemos passar o atributo `sheet_name='Sheet1'` que seria o nome da aba.

```
pd.read_excel('Exemplo_Excel.xlsx', sheet_name='Sheet1')
```

HTML

o HTML pode ser usado como um formato de entrada para carregar dados tabulares de uma página da web em um DataFrame. Isso é possível graças à função do pandas `read_html()`, que extrai as tabelas HTML de uma página da web e retorna uma lista de DataFrames representando essas tabelas encontradas.

Entrada HTML

Vamos usar os dados dessa pagina para uma demonstração simples.

```
df =
↪  pd.read_html('https://www.fdic.gov/resources/resolutions/bank-failures/failed-bank-list/')
```

e na saída temos isso

```
[                         Bank NameBank       CityCity StateSt  CertCert  \
 0                         Citizens Bank       Sac City      IA      8758
 1              Heartland Tri-State Bank        Elkhart      KS     25851
 2                    First Republic Bank  San Francisco      CA     59017
 3                         Signature Bank       New York      NY     57053
 4                    Silicon Valley Bank    Santa Clara      CA     24735
```

```
..                                ...           ...     ...       ...
563                Superior Bank, FSB       Hinsdale      IL     32646
564               Malta National Bank          Malta      OH      6629
565   First Alliance Bank & Trust Co.     Manchester      NH     34264
566  National State Bank of Metropolis     Metropolis      IL      3815
567                  Bank of Honolulu       Honolulu      HI     21029

                Acquiring InstitutionAI Closing DateClosing  FundFund
0             Iowa Trust & Savings Bank    November 3, 2023     10545
1                Dream First Bank, N.A.       July 28, 2023     10544
2             JPMorgan Chase Bank, N.A.         May 1, 2023     10543
3                  Flagstar Bank, N.A.      March 12, 2023     10540
4    First-Citizens Bank & Trust Company      March 10, 2023     10539
..                                   ...                ...       ...
563                Superior Federal, FSB       July 27, 2001      6004
564                    North Valley Bank         May 3, 2001      4648
565  Southern New Hampshire Bank & Trust    February 2, 2001      4647
566              Banterra Bank of Marion   December 14, 2000      4646
567                   Bank of the Orient    October 13, 2000      4645

[568 rows x 7 columns]]
```

Os dados tabulares foram extraídos diretamente da página daquele site. Esta operação é suportada apenas por alguns sites e requer um comando bastante específico. Os exemplos foram executados no Google Colab, onde não foi necessário instalar outras bibliotecas.

Entenda mais

Para aproveitar ao máximo este material, é fundamental ler a documentação pandas io, onde você encontrará informações mais específicas e detalhadas, além de uma variedade de formatos de arquivos que podem ser lidos com o Pandas.