

Pandas

Conceitos e Elementos Básicos

Laboratório de Ciência de Dados- Estatística - Albert E. F.
Muritiba



Introdução

História e Contexto do Pandas

Pandas, abreviação de "Python Data Analysis Library," foi criado por Wes McKinney em 2008. Foi desenvolvido para fornecer ferramentas de análise e manipulação de dados semelhantes às encontradas em R e Excel, mas construídas sobre a linguagem de programação Python.



Pandas

Python for Data Analysis

Importância do Pandas na Ciência de Dados

- **Estruturas de Dados Flexíveis:** Essas estruturas permitem a manipulação eficiente e intuitiva de dados rotulados, facilitando operações complexas de dados.
- **Integração com Outras Bibliotecas:** Pandas é frequentemente usada em conjunto com outras bibliotecas de análise de dados, como NumPy, scipy, Matplotlib, entre outras.
- **Suporte a Diversos Formatos de Dados:** Pandas suporta a leitura e escrita de dados em diversos formatos, como CSV, Excel, SQL, JSON, HTML, entre outros.

- **Desempenho e Eficiência:** Se bem utilizada, Pandas pode ser uma ferramenta poderosa para manipulação de dados, haja vista suas operações internas otimizadas e vetorizadas.
- **Comunidade Ativa e Suporte:** Pandas é uma das bibliotecas mais populares para manipulação de dados em Python, com uma comunidade ativa e fóruns de suporte.

Pandas vs SQL

- Pandas é uma biblioteca de manipulação de dados em memória, enquanto SQL é uma linguagem de consulta a bancos de dados.
- Pandas é mais adequado para operações de **análise exploratória de dados e manipulação de dados em memória**, enquanto SQL é mais adequado para **consultas complexas em bancos de dados**.
- Pandas é mais **flexível e intuitivo** para operações de análise de dados, enquanto SQL é mais **eficiente** para consultas em bancos de dados.

Estruturas de Dados Fundamentais

Series

- Objeto unidimensional semelhante a um *array*
- Rótulos de índice para cada elemento
- dados homogêneos (mesmo tipo)

DataFrames

- Objeto bidimensional semelhante a uma tabela ou planilha
- Rótulos de índice e colunas para cada elemento
- coleção de Series

Nota: Há outras estruturas de dados no Pandas, mas Series e DataFrames são as mais comuns e amplamente utilizadas.

Importando a Biblioteca Pandas

```
import pandas as pd
```

`pd` é um apelido (*alias*) para 'pandas' que é comumente utilizado na comunidade Python.



Criando Series

Usando Listas

```
>>> s = pd.Series([1, 3, 5, np.nan, 6, 8])
>>> s
0    1.0
1    3.0
2    5.0
3    NaN
4    6.0
5    8.0
dtype: float64
```

Note que o Pandas automaticamente atribui um índice para cada elemento da *Series* (0-5). E o tipo de dado é inferido automaticamente (`float64`).

`dtype` é um atributo que retorna o tipo de dado da *Series*.

Designando um índice personalizado, nome e tipo de dado:

```
s = pd.Series([25, 30, 35, 40, 45],  
...              index=['Edu', 'Ana', 'Bob', 'Jon', 'Lia'],  
...              name='Idade', dtype='int64')  
>>> s  
Edu    25  
Ana    30  
Bob    35  
Jon    40  
Lia    45  
Name: Idade, dtype: int64
```

[documentação](#)

Usando um *array* NumPy

```
>>> data = np.array([25, 30, 35, 40, 45])
>>> s = pd.Series(data, copy=True)
```

`copy` é um argumento que indica ao Pandas se deve fazer uma cópia dos dados.
`default` é `copy=False`.

- Quando usamos um *array* NumPy para criar uma *Series*, por padrão, o Pandas não faz uma cópia dos dados, mas sim **referencia** o *array* original.
- Isso pode ser útil para **economizar memória** em grandes conjuntos de dados.
- Mas é importante ter **cuidado ao modificar o *array* original**, pois isso pode afetar a *Series*.

Usando um Dicionário

```
>>> data = {'Edu': 25, 'Ana': 30, 'Bob': 35, 'Jon': 40, 'Lia': 45}  
>>> s = pd.Series(data)  
>>> s  
Edu    25  
Ana    30  
Bob    35  
Jon    40  
Lia    45  
dtype: int64
```

Quando usamos um dicionário para criar uma *Series*, as **chaves** do dicionário são automaticamente atribuídas como **rótulos de índice** da *Series*.

Dica de performance

Se seu algoritmo precisa 'montar' uma *Series*, é mais eficiente criar um dicionário ou lista e depois transformá-lo em *Series*.

Execute o teste abaixo no *Colab*:

```
%%timeit
s = pd.Series()
for i in range(10000):
    s.loc[i] = i
```

```
%%timeit
d = []
for i in range(10000):
    d.append(i)
s = pd.Series(d)
```

Iterando sobre uma Series

```
>>> s = pd.Series([25, 30, 35, 40, 45], index=['Edu', 'Ana', 'Bob', 'Jon', 'Lia'])
>>> for key, value in s.items():
...     print(f'{key}: {value}')
Edu: 25
Ana: 30
Bob: 35
Jon: 40
Lia: 45
```

O método `items()` retorna um iterador sobre os rótulos de índice e os valores da Series.

Acessando Elementos de uma Series

Por Índice

```
>>> s = pd.Series([25, 30, 35, 40, 45], index=['Edu', 'Ana', 'Bob', 'Jon', 'Lia'])  
>>> s[0]  
25
```

Por Rótulo de Índice

```
>>> s['Edu']  
25
```

Atenção: As duas formas apresentadas acima são ambíguas e podem causar confusão. É recomendado usar `.iloc` para indexação por posição e `.loc` para indexação por rótulo.

Exemplo ambíguo

```
>>> s = pd.Series([25, 30, 35, 40], index=[1, 2, 3, 4])
>>> s[3]
35
>>> s = pd.Series([25, 30, 35, 40])
>>> s[3]
40
```

`s[3]` é o elemento com rótulo `3` ou o elemento na posição `3`? Depende do contexto. Por isso, é recomendado usar `.iloc` e `.loc` para evitar ambiguidades.

Por Posição

```
>>> s = pd.Series([25, 30, 35, 40, 45], index=['Edu', 'Ana', 'Bob', 'Jon', 'Lia'])
>>> s.iloc[0]
25
```

Por Rótulo de Índice

```
>>> s.loc['Edu']
25
```

Dica de performance

- Tudo bem usar `.loc` e `.iloc` para acessar um elemento específico para depuração ou inspeção durante o desenvolvimento.
- O mesmo para `for k, v in s.items():`.
- Mas suas empregos **denunciam** o acesso elemento por elemento, o que é **ineficiente**.
- Procure sempre usar **operações vetorizadas** para acessar elementos de uma Series ou DataFrame.
- Exemplo:

```
>>> a = pd.Series([1, 2, 3, 4, 5])
>>> x = all(a % 2 == 0) # Verifica se todos os elementos são pares
>>> x
False
```

Modificando Elementos de uma Series

```
>>> s = pd.Series([25, 30, 35, 40, 45], index=['Edu', 'Ana', 'Bob', 'Jon', 'Lia'])
>>> s.loc['Edu'] = 55
>>> s.iloc[1] = 77
>>> s
Edu    55
Ana    77
Bob    35
Jon    40
Lia    45
dtype: int64
```

- Quando atribuímos a uma **posição** (`.iloc`) que não existe, o Pandas retorna um `IndexError`.

```
>>> s.iloc[5] = 100  
IndexError: iloc cannot enlarge its target object
```

- Quando atribuímos a um **rótulo** (`.loc`) que não existe, o Pandas **adiciona** um novo elemento à Serie.

```
>>> s.loc['Rau'] = 100
```

- Modificando mais de um elemento de uma vez:

```
>>> s = pd.Series([25, 30, 35, 40, 45], index=['Edu', 'Ana', 'Bob', 'Jon', 'Lia'])
>>> s.loc[['Edu', 'Ana', 'Bob']] = 100
>>> s
Edu    100
Ana    100
Bob    100
Jon     40
Lia     45
dtype: int64
```

- Modificando elementos com base em uma condição:

```
>>> s = pd.Series([25, 30, 35, 40, 45], index=['Edu', 'Ana', 'Bob', 'Jon', 'Lia'])
>>> s[s > 30] = 100
>>> s
Edu    25
Ana    30
Bob    100
Jon    100
Lia    100
dtype: int64
```

Removendo Elementos de uma Series

Por Rótulo de Índice

```
>>> s = pd.Series([25, 30, 35, 40, 45], index=['Edu', 'Ana', 'Bob', 'Jon', 'Lia'])
>>> s.drop('Edu', inplace=True)
>>> s
Ana    30
Bob    35
Jon    40
Lia    45
dtype: int64
```

O método `drop()` remove um elemento da Series com base no rótulo de índice.

O argumento `inplace=True` modifica a Series original.

Se `inplace=False` (padrão), o método retorna uma nova Series sem o elemento removido.

Por Posição

```
>>> s = pd.Series([25, 30, 35, 40, 45], index=['Edu', 'Ana', 'Bob', 'Jon', 'Lia'])
>>> s.drop(s.index[0], inplace=True) # Remove o primeiro elemento
>>> s
Ana    30
Bob    35
Jon    40
Lia    45
dtype: int64
```

`.index` retorna os rótulos de índice da Series. Assim, `s.index[0]` retorna o rótulo do primeiro elemento da Series.

Removendo Elementos por Condição

```
>>> s = pd.Series([25, 30, 35, 40, 45], index=['Edu', 'Ana', 'Bob', 'Jon', 'Lia'])
>>> s = s[s > 30] # Remove elementos menores ou iguais a 30
>>> s
Bob    35
Jon    40
Lia    45
dtype: int64
```

Slicing em uma Series

Por Posição

```
>>> s = pd.Series([25, 30, 35, 40, 45], index=['Edu', 'Ana', 'Bob', 'Jon', 'Lia'])
>>> s.iloc[1:3]
Ana    30
Bob    35
dtype: int64
```

Por Rótulo de Índice

```
>>> s.loc['Ana':'Jon']
Ana    30
Bob    35
Jon    40
dtype: int64
```

Atenção: O *slicing* por rótulo de índice é inclusivo.

Concatenando Series

```
>>> s1 = pd.Series([25, 30, 35], index=['Edu', 'Ana', 'Bob'])
>>> s2 = pd.Series([40, 45], index=['Jon', 'Lia'])
>>> s = pd.concat([s1, s2])
>>> s
Edu    25
Ana    30
Bob    35
Jon    40
Lia    45
dtype: int64
```

Em caso de duplicidade de rótulos, o Pandas mantém os rótulos duplicados.

Tratando rótulos duplicados

- Forçar a verificação de duplicidade de rótulos:

```
>>> s1 = pd.Series([25, 30, 35], index=['Edu', 'Ana', 'Bob'])
>>> s2 = pd.Series([40, 45], index=['Bob', 'Ana'])
>>> s = pd.concat([s1, s2], verify_integrity=True)
ValueError: Index has duplicates
```

- Ignorar os rótulos dos elementos concatenados, novos rótulos numéricos são criados:

```
>>> s = pd.concat([s1, s2], ignore_index=True)
```

- Concatenar com a diferença:

```
>>> s2_minus_s1 = set(s2.index) - set(s1.index)
>>> s = pd.concat([s1, s2.loc[s2_minus_s1]])
```

Ordenando uma Series

Por Índice

```
>>> s = pd.Series([25, 30, 35, 40, 45], index=['Edu', 'Ana', 'Bob', 'Jon', 'Lia'])  
>>> s.sort_index(ascending=False, inplace=True)
```

O método `sort_index()` ordena a Series com base nos rótulos de índice. O argumento `ascending=False` ordena em ordem decrescente.

O argumento `inplace=True` modifica a Series original.

Se `inplace=False` (padrão), o método retorna uma nova Series ordenada.

Por Valor

```
>>> s = pd.Series([25, 30, 35, 40, 45], index=['Edu', 'Ana', 'Bob', 'Jon', 'Lia'])  
>>> s.sort_values(ascending=False, inplace=True)
```

DataFrames



Criando DataFrames

Usando Listas

```
>>> nomes = ['Edu', 'Ana', 'Bob', 'Jon', 'Lia']
>>> idades = [25, 30, 35, 40, 45]
>>> pesos = [70, 65, 80, 75, 85]
>>> cols = ['Nome', 'Idade', 'Peso']
>>> data = [nomes, idades, pesos]
>>> df = pd.DataFrame(data, index=cols).T
>>> df
   Nome  Idade  Peso
0   Edu     25    70
1   Ana     30    65
2   Bob     35    80
3   Jon     40    75
4   Lia     45    85
```

O método `.T` transpõe o DataFrame, trocando linhas por colunas.

Sem transpor o DataFrame:

```
>>> rows = []
>>> rows.append(['Edu', 25, 70])
>>> rows.append(['Ana', 30, 65])
>>> rows.append(['Bob', 35, 80])
>>> rows.append(['Jon', 40, 75])
>>> rows.append(['Lia', 45, 85])
>>> cols = ['Nome', 'Idade', 'Peso']
>>> df = pd.DataFrame(rows, columns=cols)
>>> df
```

	Nome	Idade	Peso
0	Edu	25	70
1	Ana	30	65
2	Bob	35	80
3	Jon	40	75
4	Lia	45	85

Usando um Dicionário

```
>>> data = {'Nome': ['Edu', 'Ana', 'Bob', 'Jon', 'Lia'],
...           'Idade': [25, 30, 35, 40, 45],
...           'Peso': [70, 65, 80, 75, 85]}
>>> df = pd.DataFrame(data)
>>> df
   Nome  Idade  Peso
0   Edu     25    70
1   Ana     30    65
2   Bob     35    80
3   Jon     40    75
4   Lia     45    85
```

Juntando Series

```
>>> nomes = pd.Series(['Edu', 'Ana', 'Bob', 'Jon', 'Lia'], name='Nome')
>>> idades = pd.Series([25, 30, 35, 40, 45], name='Idade')
>>> pesos = pd.Series([70, 65, 80, 75, 85], name='Peso')
>>> df = pd.concat([nomes, idades, pesos], axis=1)
>>> df
```

	Nome	Idade	Peso
0	Edu	25	70
1	Ana	30	65
2	Bob	35	80
3	Jon	40	75
4	Lia	45	85

O argumento `axis=1` indica que a concatenação deve ser feita ao longo das colunas.

Carregando um DataFrame de um arquivo CSV

```
>>> df = pd.read_csv('data.csv')
```

O método `read_csv()` carrega um arquivo CSV em um DataFrame. O arquivo deve estar no mesmo diretório do script Python ou o caminho completo deve ser fornecido.

[documentação](#)

Carregando um DataFrame de um arquivo Excel

```
>>> df = pd.read_excel('data.xlsx')
```

Pode ser necessário instalar a biblioteca `openpyxl` para ler arquivos Excel. O arquivo deve estar no mesmo diretório do script Python ou o caminho completo deve ser fornecido. No colab, execute `!pip install openpyxl`.

[documentação](#)

Carregando de um arquivo de campos de largura fixa

```
>>> df = pd.read_fwf('data.txt', widths=[10, 10, 10])
```

[documentação](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_fwf.html)

Carregando de um arquivo JSON

```
>>> df = pd.read_json('data.json')
```

[documentação](#)

Também é possível carregar arquivos SQL, HTML, entre outros formatos além de se conectar a bancos de dados SQL. Mas não abordaremos esses métodos neste curso.

Atributos principais de um DataFrame

- `shape` : Retorna uma tupla com o número de linhas e colunas do DataFrame.
- `columns` : Retorna uma lista com os rótulos das colunas do DataFrame.
- `index` : Retorna uma lista com os rótulos das linhas do DataFrame.
- `dtypes` : Retorna uma Series com os tipos de dados das colunas do DataFrame.
- `values` : Retorna um *array* NumPy com os valores do DataFrame.

```
>>> df = pd.DataFrame({'Nome': ['Edu', 'Ana', 'Bob', 'Jon', 'Lia'],
...                      'Idade': [25, 30, 35, 40, 45],
...                      'Peso': [70, 65, 80, 75, 85]})

>>> df.shape
(5, 3)

>>> df.columns
Index(['Nome', 'Idade', 'Peso'], dtype='object')

>>> df.index
RangeIndex(start=0, stop=5, step=1)

>>> df.dtypes
Nome      object
Idade     int64
Peso      int64
dtype: object

>>> df.values
array([['Edu', 25, 70],
       ['Ana', 30, 65],
       ['Bob', 35, 80],
       ['Jon', 40, 75],
       ['Lia', 45, 85]], dtype=object)
```

Indexação e Seleção de Dados



Indexação Básica

Vamos usar o seguinte DataFrame como exemplo:

```
>>> df = pd.DataFrame({'Nome': ['Edu', 'Ana', 'Bob', 'Jon', 'Lia'],
...                     'Idade': [25, 30, 35, 40, 45],
...                     'Peso': [70, 65, 80, 75, 85]})
```

	Nome	Idade	Peso
0	Edu	25	70
1	Ana	30	65
2	Bob	35	80
3	Jon	40	75
4	Lia	45	85

Acessando por linha

Para acessar uma linha específica, usamos o método

`.loc[]` com o rótulo da linha ou `.iloc[]` com a posição da linha.

```
>>> df.loc[0]
Nome      Edu
Idade     25
Peso      70
Name: 0, dtype: object
```

	Nome	Idade	Peso
0	Edu	25	70
1	Ana	30	65
2	Bob	35	80
3	Jon	40	75
4	Lia	45	85

Naturalmente, *slice* é possível. Exemplo:

```
df.loc[1:3].
```

Também é possível acessar várias linhas com uma lista de rótulos ou posições. Exemplo: `df.loc[[0, 2, 4]]`.

Acessando por coluna

Para acessar uma coluna específica, usamos o nome da coluna entre colchetes.

```
>>> df[ 'Nome' ]  
0    Edu  
1    Ana  
2    Bob  
3    Jon  
4    Lia  
Name: Nome, dtype: object
```

	Nome	Idade	Peso
0	Edu	25	70
1	Ana	30	65
2	Bob	35	80
3	Jon	40	75
4	Lia	45	85

Também é possível acessar várias colunas com uma lista de nomes. Exemplo: `df[['Nome', 'Peso']]`.

Acessando por coluna (alternativa)

Também podemos acessar uma coluna específica usando a **notação de ponto**.

```
>>> df.Idade  
0    25  
1    30  
2    35  
3    40  
4    45  
Name: Idade, dtype: int64
```

	Nome	Idade	Peso
0	Edu	25	70
1	Ana	30	65
2	Bob	35	80
3	Jon	40	75
4	Lia	45	85

Acessando por linha e coluna

Para acessar um elemento específico, usamos `.loc[]` com o rótulo da linha e o nome da coluna.

```
>>> df.loc[0, 'Nome']
'Edu'
>>> df.loc[1:2, ['Nome', 'Peso']]
   Nome  Peso
1  Ana    65
2  Bob    80
```

	Nome	Idade	Peso
0	Edu	25	70
1	Ana	30	65
2	Bob	35	80
3	Jon	40	75
4	Lia	45	85

Seleção Condicional

Assim como em NumPy, podemos usar **operações de comparação** para selecionar elementos em um DataFrame.

```
>>> df[df['Idade'] > 30]
   Nome  Idade  Peso
2  Bob    35    80
3  Jon    40    75
4  Lia    45    85
```

	Nome	Idade	Peso
0	Edu	25	70
1	Ana	30	65
2	Bob	35	80
3	Jon	40	75
4	Lia	45	85

Quando queremos filtrar um DataFrame com base em duas ou mais condições, usamos operadores de conjunto:

- & *interseção* para E
- | *união* para OU
- ~ *complemento* para NÃO
- ^ *diferença simétrica* para OU EXCLUSIVO

```
>>> df[(df['Idade'] > 30) & (df['Peso'] < 80)]  
    Nome  Idade  Peso  
2  Bob     35     80  
3  Jon     40     75
```

	Nome	Idade	Peso
0	Edu	25	70
1	Ana	30	65
2	Bob	35	80
3	Jon	40	75
4	Lia	45	85

Modificando DataFrames

Alterando/Adicionando Colunas

Quando atribuímos uma lista de valores ou um valor escalar a uma nova coluna, o Pandas adiciona a coluna ao DataFrame se ela não existir.

```
>>> df['Altura'] = [1.70, 1.65, 1.80, 1.75, 1.85]
>>> df
   Nome  Idade  Peso  Altura
0  Edu     25    70    1.70
1  Ana     30    65    1.65
2  Bob     35    80    1.80
3  Jon     40    75    1.75
4  Lia     45    85    1.85
```

Caso a coluna exista, o Pandas substitui os valores existentes.

Removendo Colunas

Para remover uma coluna, usamos o método `.drop()` com o nome da coluna e o argumento `axis=1`.

```
>>> df.drop('Altura', axis=1, inplace=True)
>>> df
   Nome  Idade  Peso
0  Edu     25    70
1  Ana     30    65
2  Bob     35    80
3  Jon     40    75
4  Lia     45    85
```

O argumento `axis=1` indica que a remoção deve ser feita ao longo das colunas.

O argumento `inplace=True` modifica o DataFrame original.

Se `inplace=False` (padrão), o método retorna um novo DataFrame sem a coluna removida.

Alterando/Adicionando Linhas

Para adicionar uma nova linha, usamos o método `.loc[]` com o rótulo da nova linha.

```
>>> df.loc[5] = ['Rau', 50, 90]
>>> df
   Nome  Idade  Peso
0  Edu    25    70
1  Ana    30    65
2  Bob    35    80
3  Jon    40    75
4  Lia    45    85
5  Rau    50    90
```

Também é possível adicionar uma nova linha com um dicionário de valores, usando o nome das colunas como chaves. Exemplo: `df.loc[5] = {'Nome': 'Rau', 'Idade': 50, 'Peso': 90}`.

Removendo Linhas

Para remover uma linha, usamos o método `.drop()` com o rótulo da linha e o argumento `axis=0` (*default*).

```
>>> df.drop(5, inplace=True)
>>> df
   Nome  Idade  Peso
0  Edu    25    70
1  Ana    30    65
2  Bob    35    80
3  Jon    40    75
4  Lia    45    85
```

Adicionando linhas (Alternativas)

Usando `append()`

```
>>> df = df.append({'Nome': 'Rau', 'Idade': 50, 'Peso': 90}, ignore_index=True)
```

Usando `concat()`

```
>>> df = pd.concat([df, pd.DataFrame([[ 'Rau', 50, 90]], columns=df.columns)], ignore_index=True)
```

`ignore_index=True` é necessário para que o Pandas gere novos rótulos de índice para a nova linha.

Iterando sobre um DataFrame

```
>>> for index, row in df.iterrows():
...     print(f'{index}: {row["Nome"]}, {row["Idade"]}, {row["Peso"]}')
0: Edu, 25, 70
1: Ana, 30, 65
2: Bob, 35, 80
3: Jon, 40, 75
4: Lia, 45, 85
```

	Nome	Idade	Peso
0	Edu	25	70
1	Ana	30	65
2	Bob	35	80
3	Jon	40	75
4	Lia	45	85

O método `iterrows()` retorna um iterador sobre os rótulos de índice e as linhas do DataFrame.

Lembrando...

A manipulação de DataFrames é **vetorizada** e otimizada para **desempenho**. Portanto,
evite usar laços para modificar ou acessar elementos de um DataFrame um por um.

Ordenando um DataFrame

Por Índice

```
>>> df.sort_index(ascending=False, inplace=True)
```

Por Coluna

```
>>> df.sort_values('Idade', ascending=False, inplace=True)
```

Mais de uma coluna pode ser usada para ordenação. Exemplo:

```
df.sort_values(['Idade', 'Peso'], ascending=[False, True], inplace=True) .
```

Manipulação de Dados



Manipulação de Índices

- Índices são **importantes** para **identificar** e **acessar** linhas em um DataFrame.
- Podem ser simples (inteiros, strings) ou **hierárquicos** (MultiIndex).
- Há casos onde o índice padrão (0, 1, 2, ...) não é adequado.
- Em certos contextos, é útil **redefinir** ou **configurar** os índices de um DataFrame.

Definindo um Índice

Para definir uma coluna como índice, usamos o método

`.set_index()` com o nome da coluna.

```
>>> df.set_index('Nome', inplace=True)
>>> df
   Idade  Peso
Nome
Edu      25    70
Ana      30    65
Bob      35    80
Jon      40    75
Lia      45    85
```

	Nome	Idade	Peso
0	Edu	25	70
1	Ana	30	65
2	Bob	35	80
3	Jon	40	75
4	Lia	45	85

O argumento `inplace=True` modifica o DataFrame original.

Redefinindo um Índice

Para redefinir o índice para a sequência padrão (0, 1, 2, ...), usamos o método `.reset_index()`.

```
>>> df.reset_index(inplace=True)  
>>> df  
   Nome  Idade  Peso  
0  Edu    25    70  
1  Ana    30    65  
2  Bob    35    80  
3  Jon    40    75  
4  Lia    45    85
```

	Nome	Idade	Peso
0	Edu	25	70
1	Ana	30	65
2	Bob	35	80
3	Jon	40	75
4	Lia	45	85

O índice anterior é movido para uma coluna e um novo índice é gerado.

Se quisermos **descartar** o índice anterior, usamos o argumento `drop=True`.

Hierarquia de Índices (MultiIndex)

- Índices hierárquicos são úteis para representar dados multidimensionais.
- Podemos criar um MultiIndex passando uma lista de rótulos de índice para o método `.set_index()`.

```
>>> df = pd.DataFrame({'estado': ['SP', 'SP', 'RJ', 'RJ', 'MG'],
...                     'cidade': ['Cascavel', 'Bonito', 'Bom Jesus', 'Planalto', 'Bonito'],
...                     'população': [10000, 20000, 15000, 25000, 30000]})  
>>> df.set_index(['estado', 'cidade'], inplace=True)  
>>> df
```

		população
estado	cidade	
SP	Cascavel	10000
	Bonito	20000
RJ	Bom Jesus	15000
	Planalto	25000
MG	Bonito	30000

Acessando Dados com MultiIndex

Para acessar dados em um DataFrame com MultiIndex, usamos o método `.loc[]` com uma tupla de rótulos de índice.

```
>>> df.loc[('SP', 'Cascavel')]  
população    10000  
Name: (SP, Cascavel), dtype: int64
```

Para acessar todos os dados de um nível do índice, na ordem hierárquica, basta usar o rótulo do nível.

```
>>> df.loc['SP']  
população  
cidade  
Cascavel    10000  
Bonito      20000
```

Para acessar todos os dados de um nível aleatório do índice, usamos o método `.xs()`.

```
>>> df.xs('Bonito', level='cidade')
      população
estado
SP          20000
MG          30000
```

[documentação](#)

Operações de Agregação e Agrupamento

- Métodos de agregação (`sum` , `mean` , `count` , etc.)
- Agrupamento de dados (`groupby`)

Operações de Mesclagem e Junção

- Concatenação de DataFrames (`concat`)
- Mesclagem de DataFrames (`merge` , `join`)

Módulo 5: Limpeza e Preparação de Dados

Tratamento de Dados Faltantes

- Identificação de dados faltantes (`isna` , `notna`)
- Métodos de preenchimento e remoção (`fillna` , `dropna`)

Transformação de Dados

- Aplicação de funções em dados (`apply` , `map` , `applymap`)
- Manipulação de tipos de dados (`astype`)

Manipulação de Texto

- Métodos de string (`str`)
- Operações básicas de texto (`replace` , `contains` , `split`)

Módulo 6: Análise Exploratória de Dados (EDA)

Visualização de Dados

- Introdução ao Matplotlib e Seaborn
- Criação de gráficos básicos com Pandas (`plot`)

Estatísticas Descritivas

- Métodos de estatísticas descritivas (`mean` , `median` , `mode` , etc.)
- Resumo estatístico com `describe`

Módulo 7: Desempenho e Otimização

Operações Vetorizadas

- Vantagens das operações vetorizadas
- Aplicação prática em grandes conjuntos de dados

Uso de DataFrames de Grandes Dimensões

- Leitura e escrita de grandes arquivos (chunking)
- Otimização de desempenho (`dtype` , `memory_usage`)

Módulo 8: Projetos Práticos e Aplicações

Projeto Integrador

- Aplicação de todos os conceitos aprendidos em um projeto prático
- Análise completa de um conjunto de dados

Casos de Uso do Mundo Real

- Exemplos de aplicação do Pandas em diferentes indústrias

Recursos Adicionais

- Documentação oficial do Pandas
- Comunidade e fóruns de suporte
- Bibliografia recomendada