

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.

Removendo Colunas (Alternativa)

Quando número de colunas a ser removida for muito maior do que as que serão mantidas, podemos simplesmente **selecionar** as colunas que queremos manter.

```
>>> df = df[['Nome', 'Idade', 'Peso']]  
>>> df  
   Nome  Idade  Peso  
0  Edu      25    70  
1  Ana      30    65
```

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) .
```

Amostragem básica e inspeção

Quando trabalhamos com grandes conjuntos de dados, é útil **visualizar** apenas uma amostra dos dados para **inspeção e depuração**.

Amostras

```
>>> df.head(3) # Primeiras linhas  
>>> df.tail(3) # Últimas linhas  
>>> df.sample(3) # Amostra aleatória
```

Informações gerais

```
>>> df.info()
```

Descrição estatística

```
>>> df.describe()
```

Verificando valores nulos

```
>>> df.isnull().sum() # Contagem de valores nulos por coluna
```

Valores únicos

```
>>> df['Cidade'].unique() # Array com valores únicos
```

Contagem de valores

```
>>> df['Cidade'].value_counts() # Contagem de valores únicos
```

Manipulação de Dados



Transformação de Dados

Renomeando Colunas

```
>>> df.rename(columns={'Nome': 'name', 'Idade': 'age', 'Peso': 'weight'}, inplace=True)
```

Renomeando Índices

```
>>> df.rename(index={0: 'zero', 1: 'one'}, inplace=True)
```

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

Os argumentos `columns` e `index` são dicionários onde as **chaves** são os rótulos atuais e os **valores** são os novos rótulos.

Aplicando Funções

- Python oferece várias funções **vetorizadas** que podem ser aplicadas ou recebem como argumento uma Series ou DataFrame. Exemplo:

```
>>> df['imc'] = df['weight'] / (df['height'] ** 2)
>>> max_por_linha = df.max(axis=1)
>>> media_por_coluna = df.mean()
```

- Porém, se precisarmos de uma função mais complexa, podemos usar os métodos como: `.apply()` , `.applymap()` , `.transform()` , `.map()` ou `.replace()` .
- Eles permitem aplicar uma dada função a um DataFrame ou Series, retornando um novo DataFrame ou Series com os resultados, de forma **mais eficiente** que um laço `for` .

- **.apply()** : Aplica uma função a cada coluna ou linha de um DataFrame.[doc](#)
 - Ao longo das colunas:

```
# máximo de cada coluna
>>> df.apply(lambda x: x.max())
```
 - Ao longo das linhas:

```
# soma de cada linha
>>> df.apply(lambda x: x.sum(), axis=1)
```
- **.applymap()** : Aplica uma função a cada elemento de um DataFrame.[doc](#)

```
# duplica cada elemento
>>> df.applymap(lambda x: x * 2)
```

Observe que o DataFrame original não é modificado.

- `.transform()` : Semelhante ao `.apply()`, mas com mais opções para a função. [doc](#)

- Função ou String

```
# normaliza cada coluna
>>> df.transform(lambda x: (x - x.mean()) / x.std())
>>> df.transform('mean')
```

- Lista de funções ou Strings

```
# média, desvio padrão e amplitude de cada coluna
>>> df.transform(['mean', 'std', lambda x: x.max() - x.min()])
```

- Dicionário

```
# média de 'age' e desvio padrão de 'weight'
>>> df.transform({'age': 'mean', 'weight': 'std'})
```

Obs.: Quando aplicado sobre um agrupamento, `.transform()` retorna um objeto com o mesmo índice do DataFrame original. *Veremos mais adiante.*

- `.replace()` : Substitui valores em um DataFrame. [doc](#)

- Substituição de valores

```
>>> df.replace('Edu', 'Eduardo')
```

- Substituição de valores em uma coluna

```
>>> df['Nome'].replace('Edu', 'Eduardo')
```

- Substituição de valores em uma lista

```
>>> df.replace(['Edu', 'Ana'], ['Eduardo', 'Analu'])
```

- Substituição de valores em um dicionário

```
>>> df.replace({'Edu': 'Eduardo', 'Ana': 'Analu'})
```

Obs.: Use `inplace=True` se deseja modificar o DataFrame original.

- `.map()` : Substitui valores em uma **Series**. [doc](#)

- Substituição de valores

```
>>> df['Nome'].map({'Edu': 'Eduardo', 'Ana': 'Analu'})
```

- Substituição de valores com uma função

```
>>> df['Nome'].map(lambda x: x.upper(), na_action='ignore')
```

`na_action='ignore'` propaga valores **nulos** sem substituição.

Execícios de Fixação

Usando o banco de dados disponível em [Spotify Tracks DB](#) faça o seguinte:

1. Carregue o arquivo `SpotifyFeatures.csv` em um DataFrame.
2. Obtenha as informações gerais do DataFrame (número de linhas, colunas, tipos de dados).
3. Mantenha apenas as colunas `genre`, `artist_name`, `track_name`, `popularity`, `duration_ms`, `key`, `mode`, `tempo` e `track_id`.
4. Renomeie as colunas para `genero`, `artista`, `musica`, `popularidade`, `duracao`, `tom`, `modo`, `tempo`.
5. O tom está no padrão americano (C, D, E, F, G, A, B). Converta para o padrão brasileiro (Dó, Ré, Mi, Fá, Sol, Lá, Si). Faça o mesmo para o modo (Major, Minor) mudando para (Maior, Menor).

7. Crie uma nova coluna `duracao_min` com a duração da música em minutos.
8. Crie uma nova coluna `popularidade_cat` com a popularidade da música categorizada em baixa (0-33), média (34-66) e alta (67-100).
9. Reordene o DataFrame de acordo com gênero e popularidade (decrescente).
10. Salve o DataFrame em um arquivo CSV chamado `spotify.csv`.

Obs.: Procure usar as funções **vetoriais** do Pandas sempre que possível, evitando laços `for`.

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` indica que a modificação deve ser feita no DataFrame original.

Redefinindo um Índice

Para redefinir o índice para a sequência numérica padrão, 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`.

Definindo Índice no Carregamento de Arquivos

Se os dados a serem carregados possuem uma coluna, ou mais, que devem ser usadas como índice, podemos definir o índice diretamente no carregamento do arquivo.

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

Hierarquia de Índices (MultiIndex)

- Índices hierárquicos são úteis para representar dados multidimensionais.
- Também são comuns em resultados de agregações e operações de agrupamento.
- 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 indicar o rótulo até o nível desejado.

```
>>> 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 Concatenação e Mesclagem

- **Concatenação:** Combina DataFrames ao longo de um eixo (linhas ou colunas). Alinhamento é feito com base em **rótulos de índice**.
- **Mesclagem:** Combina DataFrames com base em uma ou mais **chaves**. Alinhamento é feito com base nos valores das chaves.

Concatenação de DataFrames (Linhas)

```
>>> df1 = pd.DataFrame({'Nome': ['Edu', 'Ana'],
...                      'Idade': [25, 30]})
>>> df2 = pd.DataFrame({'Nome': ['Bob', 'Jon'],
...                      'Idade': [35, 40]})
>>> df = pd.concat([df1, df2], ignore_index=True)
>>> df
   Nome  Idade
0   Edu     25
1   Ana     30
2   Bob     35
3   Jon     40
```

Concatenação de DataFrames (Colunas)

```
>>> df1 = pd.DataFrame({'Nome': ['Edu', 'Ana'],
...                      'Idade': [25, 30]})
>>> df2 = pd.DataFrame({'Peso': [70, 65],
...                      'Altura': [1.70, 1.65]})
>>> df = pd.concat([df1, df2], axis=1)
>>> df
   Nome  Idade  Peso  Altura
0  Edu      25    70    1.70
1  Ana      30    65    1.65
```

Obs.: A concatenação no eixo das colunas (`axis = 1`) é feita com base nos **rótulos de índice**. Se os rótulos não forem iguais, o Pandas preenche com valores nulos.

```
>>> df1 = pd.DataFrame({'Nome': ['Edu', 'Ana'],
...                      'Idade': [25, 30]})
>>> df1.set_index('Nome', inplace=True)
>>> df2 = pd.DataFrame({'Nome': ['Bob', 'Ana'],
...                      'Idade': [35, 40]})
>>> df2.set_index('Nome', inplace=True)
>>> df = pd.concat([df1, df2], axis=1)
>>> df
      Idade    Idade
Ana    30.0     40
Bob      NaN     35
Edu    25.0     NaN
```

Para ignorar os rótulos de índice, usamos o argumento `ignore_index=True`.

Mesclagem de DataFrames

Mesclagem (*join*) é uma operação que combina colunas de dois DataFrames com base em uma ou mais **chaves**. Uma das operações mais comuns em **bancos de dados relacionais**.

```
>>> df1 = pd.DataFrame({'Nome': ['Edu', 'Ana', 'Bob'],
...                      'Idade': [25, 30, 35]})
>>> df2 = pd.DataFrame({'Nome': ['Ana', 'Bob', 'Jon'],
...                      'Peso': [65, 80, 75]})
>>> df = pd.merge(df1, df2, on='Nome', how='inner')
>>> df
   Nome  Idade  Peso
0  Ana     30    65
1  Bob     35    80
```

O argumento `on='Nome'` indica a coluna-chave para a junção.

O argumento `how='inner'` indica o tipo de junção. *default* é `inner`.

Tipos de Junção

- **Inner Join:** Retorna apenas as linhas que têm chaves em ambos os DataFrames.
`how='inner'`.
- **Left Join:** Retorna todas as linhas do DataFrame da esquerda e as linhas correspondentes do DataFrame da direita.
`how='left'`.
- **Right Join:** Retorna todas as linhas do DataFrame da direita e as linhas correspondentes do DataFrame da esquerda.
`how='right'`.
- **Outer Join:** Retorna todas as linhas quando há uma correspondência em um dos DataFrames.
`how='outer'`.

Operações de Agregação e Agrupamento

Operações de Agregação

- Agregação é o processo de **combinação de múltiplos valores em um único valor**.
- Principais métodos de agregação:
 - `sum()` , `prod()` : Soma e produto dos valores.
 - `mean()` , `median()` : Média e mediana dos valores.
 - `count()` : Contagem dos valores.
 - `min()` , `max()` : Mínimo e máximo dos valores.
 - `idxmin()` , `idxmax()` : Índice do valor mínimo e máximo.
 - `std()` , `var()` : Desvio padrão e variância dos valores.
 - `quantile()` : Quantil dos valores.

Exemplo de Agregação

```
>>> df = pd.DataFrame({'nome': ['Edu', 'Ana', 'Bob', 'Jon', 'Lia'],
...                      'idade': [25, 30, 35, 40, 45],
...                      'peso': [70, 65, 80, 75, 85]})  
>>> df.set_index('nome', inplace=True)  
>>> df.mean() # Média dos valores de cada coluna  
idade    35.0  
peso    75.0  
dtype: float64  
>>> df['idade'].sum() # Soma dos valores da coluna 'idade'  
175
```

Observe que, por padrão, as agregações são feitas na vertical (colunas). Para agregações horizontais (linhas), usamos o argumento `axis=1`.

Adicionando Linhas de Agregações para Impressão

Atenção: Este é um exemplo didático, não é uma prática comum. Os valores de agregação **não devem** ser colocados no DataFrame original.

```
>>> media = df.mean()
>>> soma = df.sum()
>>> df.loc['Média'] = media
>>> df.loc['Total'] = soma
>>> df
```

	idade	peso
nome		
Edu	25.0	70.0
Ana	30.0	65.0
Bob	35.0	80.0
Jon	40.0	75.0
Lia	45.0	85.0
Média	35.0	75.0
Total	175.0	375.0

Max e Min argumento

Quando queremos saber o índice do valor máximo ou mínimo, usamos os métodos `idxmax()` e `idxmin()`.

```
>>> df['idade'].idxmax()
'Lia'
>>> df['peso'].idxmin()
'Ana'
```

Em caso de **empate**, o método retorna o primeiro índice encontrado.

Várias Agregações

```
>>> df.agg(['mean', 'sum', 'count'])
      idade    peso
mean    35.0   75.0
sum    175.0  375.0
count    5.0    5.0
```

O método `agg()` permite aplicar várias funções de agregação de uma vez.

Agregando por uma função personalizada

```
>>> df.agg(lambda x: sum(x**2)) # Soma dos quadrados dos valores de cada coluna  
          idade     peso  
0      6250.0  22750.0
```

Podemos definir uma função personalizada e passá-la como argumento para o método `agg()`.

A função recebe uma `Series` com os valores de cada coluna.

Operações de Agrupamento

- **Agrupamento** é o processo de **divisão** dos dados em **grupos** com base em **critérios** específicos.
- Principais métodos de agrupamento:
 - `groupby()` : Agrupa os dados com base em uma ou mais colunas.
 - `agg()` : Aplica uma ou mais funções de agregação aos grupos.
 - `transform()` : Aplica uma função de transformação aos grupos.
 - `filter()` : Filtra os grupos com base em um critério.
 - `apply()` : Aplica uma função arbitrária aos grupos.
- **Agrupamento** é uma operação **importante** em **análise de dados** e **preparação de dados**.

Para os exemplos a seguir, vamos usar o arquivo *Spotify Tracks DB* disponível no [Kaggle](#)

```
>>> import pandas as pd  
>>> df = pd.read_csv('SpotifyFeatures.csv')  
>>> df.dtypes  
genre                  object  
artist_name            object  
track_name              object  
track_id                object  
popularity             int64  
acousticness           float64  
danceability           float64  
duration_ms             int64  
energy                 float64  
instrumentalness       float64  
key                    object  
liveness               float64  
loudness               float64  
mode                   object  
speechiness             float64  
tempo                  float64  
time_signature          object  
valence                float64
```

Qual é a média de popularidade da amostra?

Para isso usamos uma operação de agregação bem simples.

```
>>> df['popularity'].mean()  
41.12750297467383
```

Qual é a média de popularidade por gênero?

Aqui usamos a operação de agrupamento `groupby()` seguida da operação de agregação `mean()`.

```
>>> df.groupby('genre')['popularity'].mean() # Média de popularidade por gênero  
genre  
A Capella          9.302521  
Alternative        50.213430  
Anime               24.258729  
...  
...
```

Vamos compreender melhor o que aconteceu.

- `df.groupby('genre')` cria um objeto `DataFrameGroupBy` que contém os grupos de dados. Seria como se os dados fossem separados em varias tabelas, uma para cada gênero.
- `['popularity']` seleciona a coluna de popularidade dos grupos, ignorando as outras colunas.
- `mean()` calcula a média de popularidade para cada grupo.
- O resultado é uma `Series` com índice igual aos rótulos de gênero e valores iguais às médias de popularidade.

Qual é a média, o mínimo e o máximo de popularidade por gênero?

```
>>> df.groupby('genre')['popularity'].agg(['mean', 'min', 'max'])  
              mean   min   max  
genre  
A Capella      9.302521    0    44  
Alternative    50.213430    0    83  
Anime          24.258729    0    65  
Blues          34.742879    0    80  
Children's Music  4.252637    0    51  
...
```

O método `agg()` permite aplicar várias funções de agregação de uma vez.

Qual é a música mais popular de cada gênero?

```
>>> df.loc[df.groupby('genre')['popularity'].idxmax()][['genre', 'track_name']]  
      genre          track_name  
552     A Capella    For the Longest Time  
671     Alternative   Sanctuary  
27732    Anime        Unravel  
45632    Blues        Sweet Home Alabama  
...  
...
```

Qual o tom mais comum das músicas de cada gênero?

```
>>> df.groupby('genre')['key'].agg(lambda x: x.value_counts().idxmax())
genre
A Capella          G
Alternative        C
Anime              C
...
...
```

Aqui, usamos uma função lambda para contar os valores únicos de cada tom e retornar o tom mais comum.

O método `value_counts()` retorna a contagem de valores únicos em uma Series.

O método `idxmax()` retorna o índice do valor máximo, equivalente ao valor mais comum ou **moda**.

Aggregação e Agrupamento com Múltiplas Colunas

Qual é a média de popularidade por gênero e modo?

```
>>> df.groupby(['genre', 'mode'])['popularity'].mean()  
genre          mode  
A Capella    Major      9.873563  
                  Minor     7.750000  
Alternative   Major     49.993874  
                  Minor    50.594507  
Anime         Major     24.465058  
                  Minor    23.920213  
...
```

Observe que passamos uma lista de colunas para o método `groupby()`.
E que o resultado é uma `Series` com um `MultilIndex`.

Qual é a média, o máximo e o mínimo de popularidade e duração por gênero?

```
>>> df.groupby('genre').agg({'popularity': ['mean', 'min', 'max'],
...                           'duration_ms': ['mean', 'min', 'max']})
      popularity           duration_ms
      mean   min    max       mean     min    max
genre
A Capella        9.302521     0    44  204467.697479  89947  303720
Alternative      50.213430     0    83  233241.364245 24000 1355938
Anime            24.258729     0    65  229937.067927 30027 1295600
...
```

Aqui, passamos um dicionário de colunas e funções de agregação para o método `agg()`.

O resultado é um DataFrame com um MultiIndex nas colunas.

Iterando sobre Grupos

```
>>> for genre, group in df.groupby('genre'):
...     print(genre)
...     print(group.head())
```

O método `groupby()` retorna um iterador sobre os grupos, onde cada grupo é um DataFrame.

É sempre bom evitar o uso de laços para manipular DataFrames, pois isso pode ser menos eficiente do que **operações vetorizadas**.

Exercícios de Fixação

1. Carregue o arquivo `zonas.csv` em um DataFrame.
2. Imprima as informações gerais do DataFrame.
3. Quantos locais ativos e inativos existem?
4. Quantos eleitores aptos a votar há no total?
5. Quantos locais de votação existem? **Observe que `cod_local` não é um identificador único.**
6. Qual local possui o maior número de eleitores aptos a votar?
7. Quantas seções eleitorais existem por município?
8. Algum local de votação possui seções vazias? Se sim, quais?

9. Para cada município, a quantidade de eleitores aptos, a quantidade de seções eleitorais, a média de eleitores por seção e o nome do local com o maior número de eleitores.
10. Qual é a razão entre o número de eleitores aptos a votar e o número de seções eleitorais por município? Dê a resposta em ordem decrescente.

Textos e Datas

Manipulação de Textos

- As *strings* são reconhecidas como **objetos** em Pandas.
- Portanto, não podemos usar diretamente os métodos de *strings* do Python.
- Para isso, usamos os métodos da série `.str`.

Exemplos:

```
>>> df['cidade'].str.upper() # Converte para maiúsculas  
>>> df['cidade'].str.len() # Tamanho das strings  
>>> df['cidade'].str.contains('São') # Verifica se contém 'São'
```

O método `.str` permite acessar os métodos de *strings* do Python.

Trabalhando com Datas

- Inicialmente, precisamos **converter** as colunas de datas para o tipo **datetime**, usando o método `pd.to_datetime()` . [doc](#)

Exemplos:

```
>>> df['data'] = pd.to_datetime(df['data']) # Converte para datetime
>>> df['data'].dt.year # Ano
>>> df['data'].dt.month # Mês
>>> df['data'].dt.day # Dia
>>> df['data'].dt.day_name() # Dia da semana
```

Exemplo de Conversão de Datas

- Formato padrão: YYYY-MM-DD HH:MM:SS .

```
>>> df = pd.DataFrame({'data': ['2021-01-01 08:00:00', '2021-02-01 09:00:00', '2021-03-01 10:00:00']})  
>>> df['data'] = pd.to_datetime(df['data'])
```

- Quando a data está em outro formato, usamos o argumento format .

```
>>> df = pd.DataFrame({'data': ['01/01/2021 08:00:00', '01/02/2021 09:00:00', '01/03/2021 10:00:00']})  
>>> df['data'] = pd.to_datetime(df['data'], format='%d/%m/%Y %H:%M:%S')
```

- O argumento format deve seguir a formatação strftime.
- [strftime](#)

Exemplo Mais Complexo

```
>>> df = pd.DataFrame({'data': ['27 de Janeiro de 2021',  
                                '28 de Fevereiro de 2021',  
                                '31 de Março de 2021']})  
>>> df['data'] = pd.data.replace('de ', '')\n    .replace('Janeiro', 'January')\n    .replace('Fevereiro', 'February')\n    .replace('Março', 'March')  
>>> df['data'] = pd.to_datetime(df['data'], format='%d %B %Y')
```

Aqui, removemos a palavra 'de' e traduzimos os nomes dos meses para o inglês, utilizando o método `.replace()`.

Conversão de Datas no Carregamento de Arquivos

```
>>> df = pd.read_csv('exemplo.csv', parse_dates=['data'])
```

Se for necessário usar um formato diferente, usamos o argumento `date_parser`.

```
>>> df = pd.read_csv('exemplo.csv',
    parse_dates=['data'],
    date_parser=lambda x: pd.to_datetime(x, format='%d/%m/%Y %H:%M:%S'))
```

Criando Intervalos de Datas

- Podemos criar intervalos de datas com o método `pd.date_range()` . [doc](#)

```
>>> pd.date_range('2021-01-01', '2021-12-31', freq='M') # Mensal  
>>> pd.date_range('2021-01-01', periods=12, freq='M') # Mensal  
>>> pd.date_range('2021-01-01', '2021-12-31', freq='W') # Semanal  
>>> pd.date_range('2021-01-01', '2021-12-31', freq='D') # Diário
```

O argumento `freq` define a **frequência** do intervalo.

O argumento `periods` define o **número de períodos**.

Operações com Datas

- Diferença de Datas: Subtração de datas resulta em um objeto **timedelta**. [doc](#)

```
>>> x = df['data2'] - df['data1']
>>> x.dt.days # Diferença em dias
>>> x.dt.seconds # Diferença em segundos
```

- Adição de Datas: Adição de um **timedelta** a uma série de datas.

```
>>> df['data'] + pd.Timedelta(days=25) # Adiciona 25 dias
```

- Filtragem por Datas: Seleção de linhas com base em datas.

```
>>> df[df['data'] > '2021-01-01'] # Linhas após 01/01/2021
>>> df[(df['data'].dt.year == 2021) & (df['data'].dt.month == 1)] # Janeiro de 2021
>>> df[df['data'].dt.day_name() == 'Monday'] # Segundas-feiras
```

Agrupamento por Datas

- **Por componente:** Usamos os métodos `.dt` para acessar os componentes de uma data. `.dt.year` , `.dt.month` , `.dt.day` , `.dt.day_of_week` , `.dt.hour` , `.dt.minute` , `.dt.second` .

```
>>> df.groupby(df['data'].dt.month)['valor'].sum() # Soma de cada mês
```

Aqui **não** teremos a soma **mês-a-mês**, mas a soma de todos os valores de cada mês.

- **Por período:** Agrupamento de dados por intervalos de tempo.
 - `.dt.to_period` : Converte uma série temporal em um período de tempo. [doc](#)
 - `resample` : Agrupamento de dados por períodos de tempo com **reamostragem**. [doc](#)

Agrupamento por Período Usando `to_period`

```
>>> df.groupby(df['data'].dt.to_period('M'))['valor'].sum() # Soma mês a mês
```

Agrupamento por Período Usando `resample`

```
>>> df.set_index('data', inplace=True)
>>> df.resample('M')['valor'].sum() # Soma mês a mês
```

A reamostragem exige que o índice seja do tipo `datetime`.

Legendas de Frequência Comuns [doc](#)

- s, min, h: Segundo, minuto, hora.
- D: Diário
- B: Dias úteis (segunda a sexta)
- W: Semanal
- M ou ME: Mensal, final do mês; MS: Mensal, início do mês.
- Q: Trimestral, final do trimestre; QS: Trimestral, início do trimestre.
- Y ou YE: Anual, final do ano; YS: Anual, início do ano.

Para obter um múltiplo de uma unidade, adicione um número antes da unidade.

Exemplo: 2W , 3M , 4Q , 5Y .

Resample

- Permite **reamostragem** de séries temporais para **diferentes frequências**, incluindo **downsampling e upsampling** [doc](#)

```
>>> df.set_index('data', inplace=True)
>>> df.resample('M').sum() # Soma mensal
>>> df.resample('2W').mean() # Média a cada 2 semanas
```

- **Downsampling:** Redução da frequência dos dados. Equivalente a uma **agregação**, mas nos **índices de tempo**.
- **Upsampling:** Aumento da frequência dos dados. Necessário preencher os novos valores.

Exemplo de *upsampling*

- **Upsampling:** Preenchimento de valores para frequências mais altas.
- O resultado possui **mais linhas** que o DataFrame original.

```
>>> df = pd.DataFrame({'data': pd.date_range('2021-01-01', periods=3, freq='M'),  
...                      'valor': [100, 200, 300]})  
>>> df.set_index('data').resample('D').ffill()
```

- Para completar os novos valores criados use:
 - `.ffill()` para preencher com o último valor conhecido.
 - `.bfill()` para preencher com o próximo valor conhecido.
 - `.interpolate()` para preencher com valores interpolados.
 - `.asfreq()` para manter os valores `NaN` ou preencher com um valor específico. [doc](#)

- Uma situação comum é termos uma série temporal com **datas faltantes**.
- Por exemplo, se temos amostras diárias para um dado sensor, mas alguns dias não estão presentes.
- Fazemos a *reamostragem* com a **mesma frequência do índice original** e preenchemos os valores faltantes.

```
# Preenche com valores interpolados (intermediários)
>>> df.set_index('data').resample('D').interpolate()
# Preenche com o último valor conhecido
>>> df.set_index('data').resample('D').ffill()
# Preenche com o próximo valor conhecido
>>> df.set_index('data').resample('D').bfill()
# Mantém os valores NaN
>>> df.set_index('data').resample('D').asfreq()
# Preenche com um valor específico
>>> df.set_index('data').resample('D').asfreq(fill_value=0)
```

Deslocamento

De modo geral, podemos deslocar os valores de uma `DataFrame` ou `Series` para frente ou para trás, usando o método `shift()`. [doc](#)

```
>>> df['valor'].shift(1) # Desloca os valores para frente  
>>> df['valor'].shift(-1) # Desloca os valores para trás
```

Em uma série temporal, o deslocamento pode ser útil para calcular **diferenças** ou **taxas de variação**.

```
>>> df['variação'] = df['valor'] - df['valor'].shift(1) # Variação diária
```

Se o índice for uma série temporal, podemos deslocar os valores de acordo com uma frequência.

```
>>> df.set_index('data', inplace=True)
>>> df.shift(1, freq='D') # Desloca os valores para frente em um dia
>>> df.shift(-1, freq='D') # Desloca os valores para trás em um dia
```

Exemplo variação semanal:

```
>>> df['variação'] = df.shift(7, freq='D')['valor'] - df['valor'] # Variação semanal
```

Janelas Temporais

- Permitem calcular **estatísticas móveis** em séries temporais.
- O método `rolling()` cria um objeto `Rolling` que permite calcular estatísticas móveis. [doc](#)

```
>>> df['valor'].rolling(window=3).mean() # Média móvel de 3 entradas  
>>> df['valor'].rolling(window=7).sum() # Soma acumulada de 7 entradas  
>>> df['valor'].rolling(window=5).std() # Desvio padrão móvel de 5 entradas
```

O argumento `window` define o tamanho da janela (**número de linhas**). Para que este valor corresponda a um intervalo de tempo, não deve haver datas faltantes, ou ...

Quando o índice é uma série temporal, podemos calcular estatísticas móveis com base em **intervalos de tempo**.

```
>>> df.set_index('data', inplace=True)
>>> df['valor'].rolling(window='7D').mean() # Média móvel de 7 dias
>>> df['valor'].rolling(window='1W').sum() # Soma acumulada de 1 semana
>>> df['valor'].rolling(window='1M').std() # Desvio padrão móvel de 1 mês
```

Aqui, a janela é definida em intervalos de tempo, como **dias, semanas e meses**, não em número de linhas.

Variação Percentual

- A variação percentual é uma medida comum em séries temporais.
- Podemos calcular a variação percentual com o método `pct_change()`. [doc](#)

```
>>> df['variação'] = df['valor'].pct_change() # Variação entre linhas
```

Se o índice for uma série temporal, podemos calcular a variação percentual com base em **intervalos de tempo**.

```
>>> df.set_index('data', inplace=True)
>>> df['variação'] = df['valor'].pct_change(freq='7D') # Variação percentual semanal
```

Horizontalização de Dados

- Quando adicionamos colunas com **dados atrasados ou estatísticas móveis**, estamos **horizontalizando** os dados.
- Isso pode ser útil para alguns algoritmos de ***machine learning***.

```
>>> df['valor_1'] = df['valor'].shift(1, freq='D') # Valor de ontem  
>>> df['valor_2'] = df['valor'].shift(2, freq='D') # Valor de anteontem  
>>> df['valor_7'] = df['valor'].rolling(window='7D').mean() # Média móvel de 7 dias
```

Agora, cada linha do DataFrame contém **dados históricos** que podem ser usados como **variáveis preditoras**.

Será que o 'valor' de hoje está relacionado com o de ontem, anteontem ou com a média dos últimos 7 dias?

Exercícios de Fixação

Usando o banco [Boi Gordo](#)

1. Carregue o arquivo `.csv` em um DataFrame, imprima as informações gerais do DataFrame.
2. Converta a coluna `Data` para o tipo `datetime`, ordene o DataFrame por data (crescente) e informe o horizonte temporal dos dados (primeira e última data) e a quantidade de dias entre elas.
3. Verifique se há datas duplicadas no DataFrame, se sim, remova-as.
4. Os campos "Último","Abertura","Máxima","Mínima","Vol." e "Var%" estão com a formatação incorreta. Corrija-os e converta para o tipo numérico.

6. Qual é a média de abertura, máxima, mínima e fechamento por ano?
7. Algum dia da semana é mais propício para a alta do preço do boi gordo?
8. Para cada mês, destaque aqueles dias em que o fechamento foi maior que $0.7 \times IQR$ d'aquele mês.
9. Calcule a média móvel de 30 dias para o preço de fechamento.
10. Quais foram os dias com: maior variação percentual absoluta, maior volume e maior variação absoluta vezes volume?

Limpeza e Preparação de Dados

Tratamento de Dados Faltantes

- Identificação de dados faltantes (`isna` , `notna`) [doc](#)

```
>>> df.isna().sum() # Contagem de valores faltantes por coluna
```

- Remoção de dados faltantes (`dropna`) [doc](#)

```
>>> df.dropna(inplace=True,  
subset=['coluna', 'outra_coluna'],  
how='any') # Remove linhas com valores faltantes
```

O argumento `subset` define as colunas que serão consideradas. default é todas as colunas.

O argumento `how` define se a linha é removida se **qualquer** valor for faltante (`any`) ou se **todos** os valores forem faltantes (`all`). default é `any`.

- Preenchimento de dados faltantes (`fillna`) [doc](#)

```
>>> df.fillna(0, inplace=True) # Preenche valores faltantes com zero
```

- Preenchimento de dados faltantes com a moda para colunas categóricas

```
>>> moda = df['coluna'].mode()[0]
>>> df['coluna'].fillna(moda, inplace=True)
```

- Preenchimento de dados faltantes séries temporais

- `method='ffill'` preenche com o valor anterior
- `method='bfill'` preenche com o valor seguinte.

```
>>> df['coluna'].fillna(method='ffill', inplace=True)
```

- Preenchimento de dados faltantes com valor agregado

```
>>> media = df['coluna'].mean()  
>>> df['coluna'].fillna(media, inplace=True)
```

- Preenchimento de dados faltantes com valor agregado **por grupo**

```
>>> df['coluna'] = df.groupby('genero')['coluna'].\  
transform(lambda x: x.fillna(x.mean()))
```

`transform()` aplica a função a cada grupo de dados, retornando uma Series com os valores preenchidos com a média do grupo com os mesmos índices do DataFrame original. Ou seja, os valores são repetidos para cada linha.

Remoção de Dados Duplicados

- Identificação de dados duplicados (`duplicated`) [doc](#)

```
>>> df.duplicated(subset=['coluna1', 'coluna2'], keep='first')
```

- Remoção de dados duplicados (`drop_duplicates`) [doc](#)

```
>>> df.drop_duplicates(subset=['coluna1', 'coluna2'], keep='first', inplace=True)
```

O argumento `subset` define as colunas que serão consideradas. default é todas as colunas.

O argumento `keep` define qual das linhas duplicadas será mantida. (`'first'` , `'last'` , `False`). default é `first` .

Normalização e Padronização

- **Normalização:** Escala os valores de uma variável para um intervalo específico.
- **Padronização:** Transforma os valores de uma variável para ter média zero e desvio padrão um.

Problemática

Imagine que temos um DataFrame com os resultados experimentais da aplicação de tipos diferentes de fertilizantes em plantas frutíferas.

Fertilizante	Fruta	Peso (g)	Maturação (dias)
A	Maçã	150	10
A	Maçã	200	12
...
B	Maçã	180	11
...
A	Melancia	1600	40

Vamos escolher o melhor fertilizante?

Basta calcular a média do peso e do tempo de maturação para cada fertilizante e escolher aquele que gera maior peso ou menor tempo médio, certo?

O que está errado nesse raciocínio?

É possível que um fertilizante que é ótimo para maçãs, laranjas, acerolas, etc., perca a comparação para um fertilizante que é apenas bom para melancias.

- Por quê?
 - Um ótimo fertilizante para maçãs pode aumentar o peso em 50g, enquanto um bom fertilizante para melancias pode aumentar o peso em 1000g.
 - Além disso, um bom fertilizante para melancias pode gerar melancias com o dobro do peso, enquanto um ótimo fertilizante para maçãs pode gerar maçãs com 10% a mais de peso.

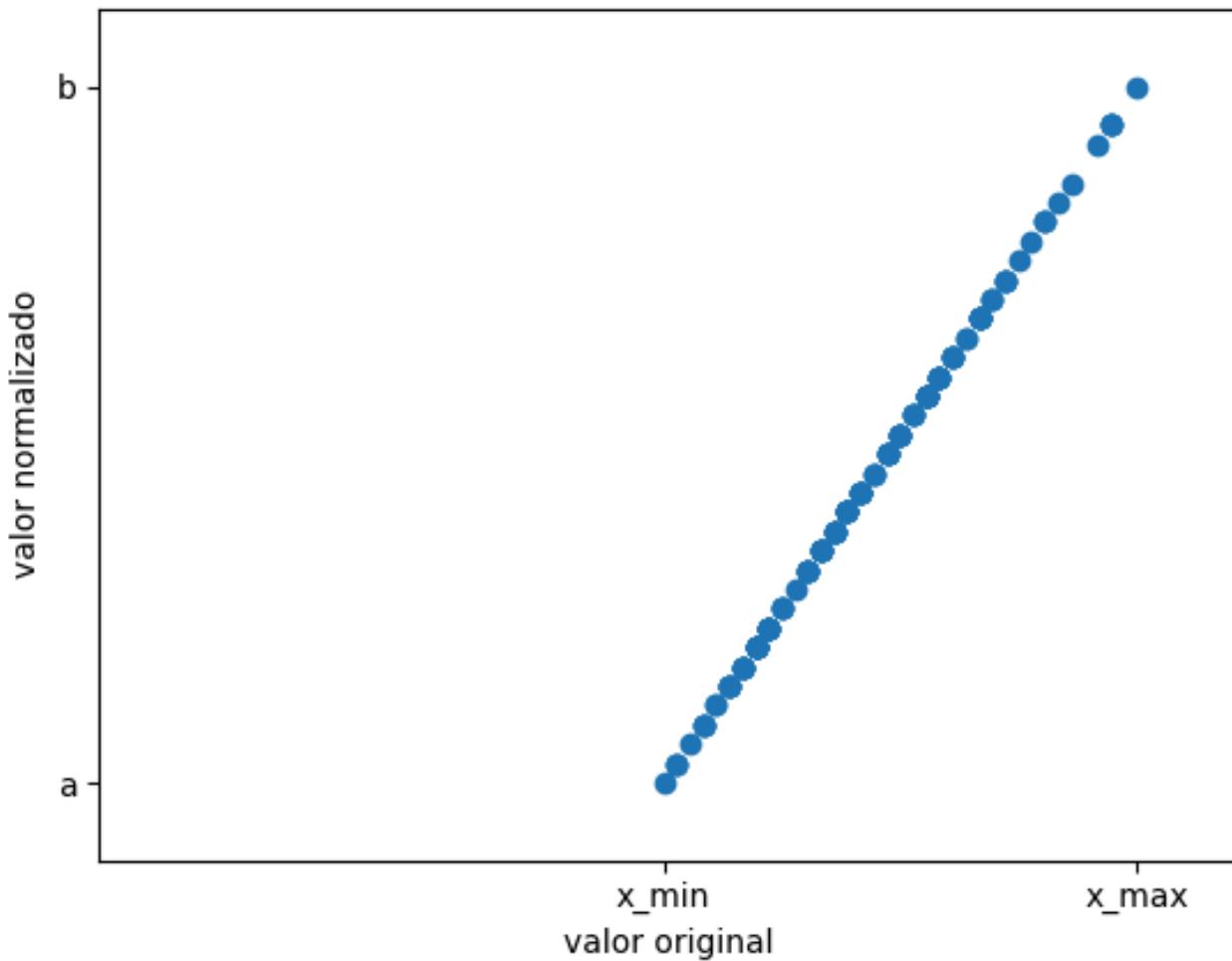
Não só a **ordem de grandeza** dos valores é diferente para cada espécie de fruta, mas também a **variabilidade** dos valores é diferente.

Se vamos **escolher apenas um fertilizante**, precisamos **comparar os resultados de cada fruta em uma mesma escala**.

Min-Max Scaling

$$x_{\text{norm}} = \frac{x - x_{\min}}{x_{\max} - x_{\min}} \times (b - a) + a$$

- Onde a e b são os limites do intervalo de normalização.
- Os valores são deslocados e escalados para o intervalo $[a, b]$.



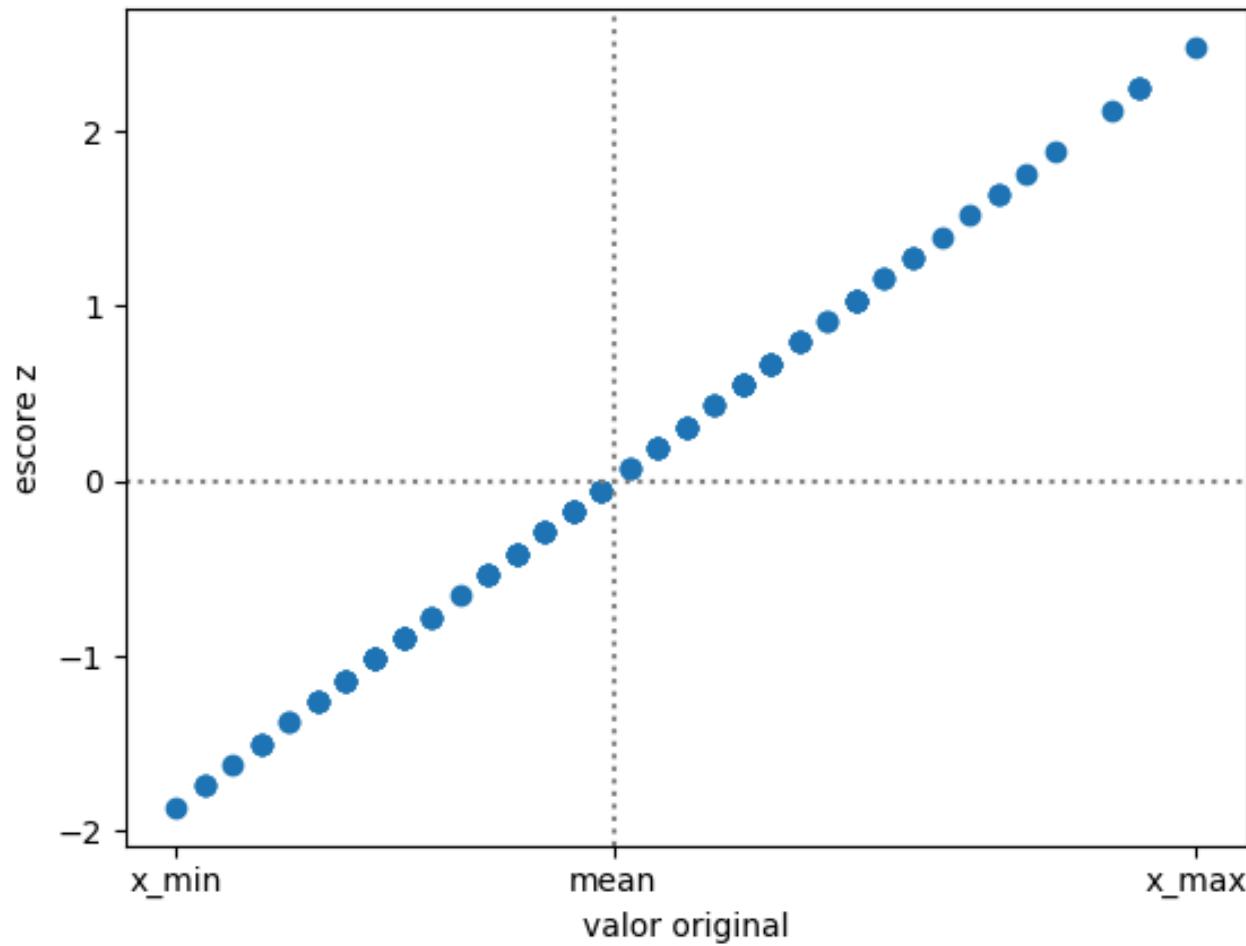
Aplicando Fórmula

```
# Normalização do peso para o intervalo [0, 100]
>>> df['peso_norm'] = (df.peso - df.peso.min()) / (df.peso.max() - df.peso.min()) * (100 - 0) + 0
```

Padronização com Standard Scaling (z-score)

$$z = \frac{x - \mu}{\sigma}$$

- A média é deslocada para zero
- A variância é escalada para um.



Aplicando Fórmula

```
>>> df['peso_norm'] = (df.peso - df.peso.mean()) / df.peso.std()
```

Usando scipy.stats.zscore

```
>>> from scipy.stats import zscore  
>>> df['peso_norm'] = zscore(df['peso'])
```

Outros Métodos de Normalização

Não abordaremos aqui, mas existem outros métodos de normalização, como:

- Robust Scaling
- Quantile Transformation
- etc.

Aplicando Padronização e Normalização em Agrupamentos

```
# z-score por grupo
>>> df['duracao_zscore'] = df.groupby('genre')['duration_ms'].transform(lambda x: zscore(x))
# Min-Max por grupo
>>> def minmax(x, a = 1, b = 0):
...     return (x - x.min()) / (x.max() - x.min()) * (b - a) + a
>>> df['pop_norm'] = df.groupby('genre')['popularity'].transform(lambda x: minmax(x))
```

Observamos que a função `transform()` é aplicada a cada grupo de dados, retornando uma Series com os valores normalizados com os mesmos índices do DataFrame original. Ou seja, os valores transformados são distribuídos para cada índice do DataFrame.

Tratamento de Outliers

- Outliers são valores que diferem significativamente do restante dos dados.
- Identificação de Outliers
 - Z-Score: Medida estatística da diferença entre um valor e a média em termos de desvio padrão.
 - IQR Score: Medida estatística da diferença entre o primeiro e terceiro quartil.

Z-Score

- O Z-Score é uma medida estatística da diferença entre um valor e a média em termos de desvio padrão.
- Valores com Z-Score maior que 3 ou menor que -3 costumam ser considerados **outliers**.

```
>>> from scipy.stats import zscore  
>>> zs = zscore(df['coluna']) # Calcula o Z-Score  
>>> outliers = df.loc[(zs > 3) | (zs < -3)].index # Índices dos outliers
```

Para remover os outliers, basta usar o método `drop()`.

```
>>> df.drop(outliers, inplace=True)
```

Z-Score em Agrupamentos

```
>>> zs = df.groupby('genre')['duration_ms'].transform(zscore)
>>> outliers = df.loc[(zs > 3) | (zs < -3)].index
>>> df.drop(outliers, inplace=True)
```

IQR Score

- O IQR Score é uma medida estatística da diferença entre o primeiro e terceiro quartil.
- Valores fora do intervalo $(Q1 - 1.5 \times IQR, Q3 + 1.5 \times IQR)$ costumam ser considerados **outliers**.

```
>>> Q1, Q3 = df['coluna'].quantile([0.25, 0.75])
>>> IQR = Q3 - Q1
>>> outliers = df.loc[(df.coluna < Q1 - 1.5 * IQR) | (df.coluna > Q3 + 1.5 * IQR)].index
>>> df.drop(outliers, inplace=True)
```

IQR Score em Agrupamentos

```
>>> Q1 = df.groupby('genre')['duration_ms'].transform(lambda x: x.quantile(0.25))
>>> Q3 = df.groupby('genre')['duration_ms'].transform(lambda x: x.quantile(0.75))
>>> IQR = Q3 - Q1
>>> outliers = df.loc[
...  (df['duration_ms'] < Q1 - 1.5 * IQR) | (df['duration_ms'] > Q3 + 1.5 * IQR)].index
>>> df.drop(outliers, inplace=True)
```

Alternativa: Criar uma função para retornar 'True' para outliers.

```
>>> def is_outlier(x, f=1.5):
...     Q1, Q3 = x.quantile([0.25, 0.75])
...     IQR = Q3 - Q1
...     return (x < Q1 - f * IQR) | (x > Q3 + f * IQR)
>>> outliers = df.groupby('genre')['duration_ms'].transform(is_outlier)
>>> df.drop(outliers, inplace=True)
```

df.drop remove as linhas com índices True .

Exercícios de Fixação

1. Carregue o arquivo `benchmark.csv` em um DataFrame, imprima as informações gerais do DataFrame.
2. Para cada instância, imprima o valor **máximo, médio e mínimo** para o valor **objetivo e tempo** de execução.
3. Faça o mesmo para cada método.
4. Cada instância pode apresentar escala de valores diferente para tempo e objetivo. Para evitar viés nas comparações seguintes, normalize estes dois campos **por instância**
5. Para cada instância, indique o método que apresenta menor **objetivo médio**.
6. Quantas vezes cada método foi o que apresentou o **menor objetivo médio** para as instâncias?

7. Para cada método, qual conjunto de parâmetros apresentou o melhor **objetivo médio** para as diferentes sementes aleatórias?
8. Levando em consideração apenas a melhor configuração de parâmetros de cada método, refaça as questões 2, 3, 5 e 6. Houve alteração nas respostas?
9. Faça um gráfico de dispersão onde cada ponto corresponde a um método, o eixo X corresponde ao objetivo normalizado médio, Y corresponde ao tempo normalizado médio, a cor corresponde ao desvio padrão do objetivo normalizado. Coloque o nome dos métodos como rótulos ao lado de cada ponto.
10. Faça um gráfico *boxplot* para os tempos de execução normalizados, dividindo os dados em dois, de acordo com o valor do campo `first_imp`.

Gráficos Rápidos com Pandas

- Pandas possui métodos para **visualização de dados**.
- Servem para visualização rápida e exploração de dados.
- Gráficos mais complexos podem ser feitos com **Matplotlib**.

Gráficos Básicos

- Gráfico de Linhas: `plot()` [doc](#)

```
>>> df['coluna'].plot()
```

- Gráfico de Barras: `plot.bar()` [doc](#)

```
>>> df['coluna'].value_counts().plot.bar()
```

- Gráfico de Pizza: `plot.pie()` [doc](#)

```
>>> df['coluna'].value_counts().plot.pie()
```

- Gráfico de Dispersão: `plot.scatter()` [doc](#)

```
>>> df.plot.scatter(x='coluna1', y='coluna2')
```

- Gráfico de Histograma: `plot.hist()` [doc](#)

```
>>> df['coluna'].plot.hist()
```

- Gráfico de Boxplot: `plot.box()` [doc](#)

```
>>> df['coluna'].plot.box()
```

- Gráfico de Área: `plot.area()` [doc](#)

```
>>> df.plot.area()
```

- Gráfico de Boxplot por Grupo: `boxplot()` [doc](#)

```
>>> df.boxplot(column='valor', by='grupo')
```

- Gráfico de Dispersão com Cores por Grupo e Tamanho Proporcional

```
>>> df.plot.scatter(x='coluna1', y='coluna2', c='grupo', s='coluna3', colormap='viridis')
```

