**Conceitos em Pandas**

Arquivo contendo as anotações sobre pandas, seus conceitos básicos e outras coisas.

Anotações baseadas em pesquisas, retiradas também de trechos das aulas do curso da Stack Academy – Data Science, e também com ajuda do ChatGPT

**Sumário**

[1. O que é o Pandas? 2](#_Toc209201107)

[2. Tipos de arquivos suportados pelo pandas 2](#_Toc209201108)

[3. Exemplo básico de um DataFrame: 4](#_Toc209201109)

[4. Métodos comuns em Pandas 4](#_Toc209201110)

[5. Alterando o dataset 6](#_Toc209201111)

# O que é o Pandas?

Pandas é uma **biblioteca do Python** feita para **análise e manipulação de dados**.  
Pensa nele como um “Excel turbinado dentro do Python”: com ele você consegue **ler, transformar, analisar e exportar dados** de maneira muito mais flexível e automatizada do que em planilhas manuais.

**Para que serve na prática?**  
No dia a dia, usamos Pandas para:

* **Ler dados** de diferentes fontes: CSV, Excel, SQL, JSON, entre outros.
* **Explorar dados**: ver as primeiras linhas, descrever estatísticas, entender o formato.
* **Transformar dados**: limpar colunas, tratar valores nulos, converter tipos.
* **Analisar dados**: agrupar, filtrar, ordenar, calcular médias/somas, gerar tabelas resumidas.
* **Exportar dados**: salvar de volta em CSV, Excel ou até mandar para um banco.

**Principais estruturas do Pandas**

* **Series**: uma coluna de dados (como uma coluna do Excel).
* **DataFrame**: uma tabela com linhas e colunas (como uma planilha inteira) que suporta diferentes tipos de dados (inteiro, float, string, etc)

# Tipos de arquivos suportados pelo pandas

O **Pandas** consegue ler e gravar dados em vários formatos. Vou te listar os **principais tipos de arquivos suportados** que realmente usamos no mercado:

**📄 Arquivos de texto/planilhas**

* **CSV** → pd.read\_csv() e df.to\_csv()
* **TSV** (tab separado) → também com read\_csv(sep="\t")
* **TXT** → idem, se for estruturado com separadores
* **Excel (.xls, .xlsx, .xlsm, .xlsb, .odf, .ods, .odt)**
  + pd.read\_excel() e df.to\_excel()
  + Suporta múltiplas abas de uma planilha.

**🗄️ Bancos de dados**

* **SQL (MySQL, PostgreSQL, SQLite, etc.)**
  + pd.read\_sql() e df.to\_sql() (precisa de conexão via SQLAlchemy ou driver).

**🌐 Estruturados em texto**

* **JSON** → pd.read\_json() e df.to\_json()
* **HTML (tabelas)** → pd.read\_html()
* **XML** → pd.read\_xml()

**📦 Outros formatos comuns em ciência de dados**

* **Parquet** → pd.read\_parquet() e df.to\_parquet()
* **Feather** → pd.read\_feather() e df.to\_feather()
* **ORC** → pd.read\_orc() e df.to\_orc()
* **Stata (.dta)** → pd.read\_stata() e df.to\_stata()
* **SPSS (.sav)** → pd.read\_spss()
* **HDF5 (.h5)** → pd.read\_hdf() e df.to\_hdf()
* **Pickle (.pkl)** → pd.read\_pickle() e df.to\_pickle()

👉 Ou seja: **se é um formato tabular ou semi-estruturado, o Pandas provavelmente consegue lidar**.

💡 **No dia a dia**:

* CSV é o mais comum (troca de dados entre sistemas).
* Excel é onipresente em empresas (relatórios, planilhas financeiras, etc).
* SQL é essencial (quase sempre os dados “moram” em um banco).
* Parquet está se tornando padrão em ambientes de Big Data (porque é compacto e rápido).

# Exemplo básico de um DataFrame:



# Métodos comuns em Pandas

O Pandas tem centenas de métodos, mas no **dia a dia de trabalho** usamos sempre um conjunto pequeno que resolve 80% dos casos. Abaixo vou listar alguns exemplos que são mais comumente usadas no dia a dia e que é sempre bom ter ciência de suas utilizações.

Vale também observar que, **df**, que se encontra no inicio do comando será o nome do dataframe que você estiver utilizando.

**Exploração de dados**

* **df.head(n)** → mostra as primeiras n linhas (por padrão o valor é 5).
* **df.tail(n)** → mostra as últimas n linhas (por padrão o valor é 5).
* **df.info()** → mostra tipos de dados, colunas e memória.
* **df.describe()** → estatísticas descritivas (média, min, max, std).
* **df.shape** → retorna (linhas, colunas).
* **df**.**columns** → lista os nomes das colunas.
* **df.dtypes** → mostra os tipos de cada coluna.

**Seleção e filtragem**

* **df["coluna"]** → seleciona uma coluna.
* **df[["col1","col2"]]** → seleciona várias colunas.
* **df.loc[linha, coluna]** → seleção por rótulo.
* **df.iloc[linha, coluna]** → seleção por posição.
* **df[df["coluna"] > 10]** → filtragem com condição.
* **df.query("coluna > 10 and outra\_coluna == 'X'")** → filtragem estilo SQL.

**Transformação**

* **df.sort\_values(by="coluna")** → ordena linhas.
* **df.rename(columns={"antigo": "novo"})** → renomeia colunas.
* **df.drop(columns=["coluna"])** → remove coluna(s).
* **df.dropna()** → remove valores nulos.
* **df.fillna(valor)** → preenche nulos.
* **df.astype(tipo)** → converte tipo de coluna.
* **df.apply(func)** → aplica função a cada linha ou coluna.
* **df["coluna"].map(func)** → aplica função em uma série.
* **df.assign(nova=df["coluna"] \* 2)** → cria colunas derivadas.

**Agregação e estatísticas**

* **df["coluna"].mean()** → média.
* **df["coluna"].sum()** → soma.
* **df["coluna"].value\_counts()** → contagem de valores únicos.
* **df.groupby("coluna").mean()** → média por grupo.
* **df.groupby(["col1","col2"]).agg({"col3": "sum"})** → agregações mais complexas.
* **df.pivot\_table(values="col", index="col1", columns="col2", aggfunc="mean")** → tabela dinâmica.

**Junções e concatenações**

* **pd.concat([df1, df2])** → concatena DataFrames.
* **df.merge(df2, on="coluna")** → faz join (inner join padrão).
* **df.merge(df2, how="left", on="coluna")** → left join.

**Entrada e saída**

* pd.**read\_csv**("arquivo.csv") / df.**to\_csv**("arquivo.csv")
* pd.**read\_excel**("arquivo.xlsx") / df.**to\_excel**("arquivo.xlsx")
* pd.**read\_sql**(query, con) / df.**to\_sql**("tabela", con)

# Alterando o dataset

Existem vários jeitos, mas dá pra organizar em **6 grupos principais**:

**1. Alterar colunas**

* **Renomear**: df.rename(columns={"idade": "Idade"}, inplace=True)
* **Criar coluna nova**: df["idade\_mais10"] = df["idade"] + 10
* **Alterar valores com função**: df["nome"] = df["nome"].str.upper()

**2. Alterar linhas**

* **Remover linhas pelo índice**: df.drop(index=[0, 1], inplace=True)
* **Filtrar e manter só o que quero**: df = df[df["idade"] > 18]

**3. Alterar valores específicos**

* **Com .loc**: df.loc[0, "cidade"] = "São Paulo" # altera célula
* **Com condição**: df.loc[df["cidade"] == "SP", "cidade"] = "São Paulo"

**4. Tratar valores nulos**

* **Remover nulos**: df.dropna(inplace=True)
* **Preencher nulos**: df.fillna("Desconhecido", inplace=True)

**5. Alterar tipos de dados**

* **Converter tipo de coluna**: df["idade"] = df["idade"].astype(int)
* **Datas**: df["data\_nascimento"] = pd.to\_datetime(df["data\_nascimento"])

**6. Reestruturar o dataset**

* **Ordenar**: df.sort\_values(by="idade", ascending=False, inplace=True)
* **Resetar índices**: df.reset\_index(drop=True, inplace=True)
* **Adicionar linhas (append/concat)**:   
  novo = pd.DataFrame({"nome": ["João"], "idade": [25], "cidade": ["BH"]})  
  df = pd.concat([df, novo], ignore\_index=True)

**Resumo**:

* Alterar dataset = mexer em **colunas, linhas, valores, nulos, tipos ou estrutura**.
* A cada modificação, você pode usar inplace=True (altera direto) ou reatribuir (df = ...).

# Percorrendo linhas de um DataFrame

Percorrer as linhas de um DataFrame é algo que às vezes precisamos, mas é importante já adiantar: **no Pandas, a ideia não é percorrer linha por linha como em um for tradicional** — porque isso costuma ser **lento** em grandes datasets.

Ainda assim, existem várias formas de fazer, dependendo da situação:

**1. Usando .iterrows() (mais comum)**

Percorre linha por linha, devolvendo o índice e a linha como Series.

for idx, row in df.iterrows():

print(idx, row["nome"], row["idade"])

É simples, mas não é muito rápido (porque converte cada linha para Series).

**2. Usando .itertuples() (mais eficiente)**

Retorna cada linha como uma *namedtuple* (mais leve que Series).

for row in df.itertuples():

print(row.Index, row.nome, row.idade)

Esse é o jeito mais usado em situações reais quando precisa iterar.

**3. Usando .apply() (jeito “pandas”)**

Aplica uma função em cada linha ou coluna.

df.apply(lambda row: print(row["nome"], row["idade"]), axis=1)

* axis=1 → percorre linhas.
* axis=0 → percorre colunas.

**4. Usando .iloc (manual, estilo lista)**

Se você realmente quiser “for clássico”:

for i in range(len(df)):

print(df.iloc[i, 0], df.iloc[i, 1]) # acessa por posição

**Melhor prática**

Na maioria dos casos, **não percorremos linhas**.  
Em vez disso, usamos operações **vetorizadas** (que aplicam direto em todas as linhas sem for).

* 1. Exemplo ruim (iterando):

for idx, row in df.iterrows():

df.loc[idx, "idade\_mais10"] = row["idade"] + 10

* 1. Exemplo bom (vetorizado):

df["idade\_mais10"] = df["idade"] + 10

O segundo é **muito mais rápido** e o jeito recomendado de trabalhar com Pandas.

# Visualização de Dados

## Missing Values

* **O que são *missing values*?**

“Missing values” = **valores ausentes** ou **dados faltantes** em um dataset.  
Ou seja, quando uma célula/coluna que deveria ter um valor está **vazia** ou marcada com algo que indica ausência.

No **Pandas**, valores ausentes geralmente aparecem como:

* **NaN** (*Not a Number*) → valor nulo em dados numéricos.
* **None** → valor nulo em objetos (strings, datas, etc).

Exemplo:

|  |
| --- |
| import pandas as pd  import numpy as np  df = pd.DataFrame({  "nome": ["Ana", "Bruno", "Carla"],  "idade": [23, np.nan, 29],  "cidade": ["SP", None, "BH"]  })  print(df) |

Saída:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Nome | Idade | Cidade |
| 0 | Ana | 23.0 | SP |
| 1 | Bruno | NaN | None |
| 2 | Carla | 29.0 | BH |

Aqui, a idade do Bruno e a cidade dele estão **faltando**.

* **Por que os dados podem estar faltando?**
  + Erro humano (alguém esqueceu de preencher).
  + Problemas no sistema que gerou o dado.
  + Dado não se aplica (exemplo: "número de filhos" para quem não é casado).
  + Integração de várias fontes de dados.
* **Como lidar com missing values no Pandas**

1. **Detectar**

df.isna().sum() # mostra quantos valores faltam por coluna

df.isnull() # retorna DataFrame de True/False

1. **Remover**

df.dropna() # remove linhas com missing

df.dropna(axis=1) # remove colunas com missing

1. **Preencher (imputar)**

df.fillna(0) # substitui por 0

df["cidade"].fillna("Desconhecido", inplace=True)

df["idade"].fillna(df["idade"].mean(), inplace=True) # média

## Visualização básica de dados

* **O que é visualização de dados?**

É a etapa em que você pega um dataset e cria **gráficos simples** para entender padrões, distribuições e relações entre variáveis.

Ela serve para:

* Explorar os dados (*EDA – Exploratory Data Analysis*).
* Comunicar achados para outras pessoas.
* Ajudar na tomada de decisão.
* **Ferramentas no Python**

1. **Pandas (built-in)** → já tem métodos rápidos.
2. **Matplotlib** → biblioteca padrão de gráficos.
3. **Seaborn** → mais bonita e simplifica estatísticas.

* **Exemplos de visualização básica**

Suponha esse dataset:

|  |
| --- |
| import pandas as pd  import matplotlib.pyplot as plt  df = pd.DataFrame({  "nome": ["Ana", "Bruno", "Carla", "Daniel", "Eva"],  "idade": [23, 35, 29, 40, 30],  "cidade": ["SP", "RJ", "SP", "BH", "RJ"]  }) |

**1. Histograma (distribuição de idades)**

df["idade"].plot(kind="hist", bins=10, title="Distribuição de Idades")

plt.show()

**2. Gráfico de barras (quantidade por cidade)**

df["cidade"].value\_counts().plot(kind="bar", title="Quantidade por Cidade")

plt.show()

**3. Dispersão (idade vs índice)**

df.plot(kind="scatter", x="nome", y="idade", title="Idade por Pessoa")

plt.show()

**4. Linha (evolução ao longo do índice/tempo)**

df["idade"].plot(kind="line", title="Idade ao longo da Tabela")

plt.show()

* **Resumindo**
  + **Visualização básica = gráficos simples** para conhecer os dados.
  + Usamos plot() do Pandas, matplotlib e seaborn.
  + Tipos comuns: histograma, barras, dispersão, linha.

## Conceitos sobre os gráficos

É muito importante conhecer **os tipos de gráficos** e **quando usar cada um**. É essencial para não só explorar os dados, mas também **contar a história certa**.

* **1. Gráfico de barras**
  + **O que mostra:** comparação entre categorias.
  + **Exemplo:** número de alunos por turma.
  + **Uso:** quando você tem categorias nominais (cidade, produto, grupo).

df["cidade"].value\_counts().plot(kind="bar", title="Alunos por Cidade")

* **2. Gráfico de colunas**
  + Igual ao de barras, mas na vertical.
  + Muito usado em relatórios de negócios.
* **3. Histograma**
  + **O que mostra:** a distribuição de uma variável numérica.
  + **Exemplo:** distribuição das idades dos alunos.
  + **Uso:** entender se os dados estão concentrados, espalhados, simétricos.

df["idade"].plot(kind="hist", bins=5, title="Distribuição de Idades")

* **4. Gráfico de linha**
  + **O que mostra:** evolução de uma variável ao longo do tempo (ou de uma sequência).
  + **Exemplo:** vendas por mês.
  + **Uso:** detectar tendências, crescimento, sazonalidade.

df["vendas"].plot(kind="line", title="Vendas ao longo do tempo")

* **5. Gráfico de dispersão (scatter plot)**
  + **O que mostra:** relação entre duas variáveis numéricas.
  + **Exemplo:** idade vs nota dos alunos.
  + **Uso:** ver correlação, agrupamentos, outliers.

df.plot(kind="scatter", x="idade", y="nota", title="Idade x Nota")

* **6. Boxplot (ou diagrama de caixa)**
  + **O que mostra:** distribuição, mediana, quartis e outliers.
  + **Exemplo:** distribuição de salários em uma empresa.
  + **Uso:** ótimo para identificar valores extremos (outliers).

df["salario"].plot(kind="box", title="Distribuição de Salários")

* **7. Gráfico de pizza (pie chart)**
  + **O que mostra:** proporções de um todo.
  + **Exemplo:** porcentagem de alunos por cidade.
  + **Uso:** bom para mostrar “fatias” de participação, mas não ideal se tiver muitas categorias.

df["cidade"].value\_counts().plot(kind="pie", autopct="%1.1f%%")

* **8. Heatmap (mapa de calor)**
  + **O que mostra:** intensidade de valores numa matriz.
  + **Exemplo:** correlação entre variáveis.
  + **Uso:** ótimo para análises rápidas de dependência entre variáveis.  
    *(Usa mais o Seaborn do que o Pandas puro)*

import seaborn as sns

sns.heatmap(df.corr(), annot=True, cmap="coolwarm")

* **Dica prática**
  + **Barras/colunas** → comparar categorias.
  + **Histograma/boxplot** → analisar distribuições.
  + **Linha** → evolução ao longo do tempo.
  + **Dispersão** → relação entre variáveis.
  + **Pizza** → proporções (só se poucas categorias).
  + **Heatmap** → dependência/correlação entre variáveis.

## Estatística Descritiva