**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? 3](#_Toc209789915)

[2. Tipos de arquivos suportados pelo pandas 3](#_Toc209789916)

[3. Exemplo básico de um DataFrame: 5](#_Toc209789917)

[4. Métodos comuns em Pandas 5](#_Toc209789918)

[5. Alterando o dataset 7](#_Toc209789919)

[6. Percorrendo linhas de um DataFrame 9](#_Toc209789920)

[7. Visualização de Dados 10](#_Toc209789921)

[7.1. Missing Values 10](#_Toc209789922)

[7.2. Visualização básica de dados 12](#_Toc209789923)

[7.3. Conceitos sobre os gráficos 14](#_Toc209789924)

[7.4. Entendendo um pouco mais sobre gráficos de bloxplot 16](#_Toc209789925)

[7.4.1. Correlação 19](#_Toc209789926)

[7.4.2. Outliers 20](#_Toc209789927)

[7.5. Estatística Descritiva 21](#_Toc209789928)

[8. Trabalhando com Tabelas e Planilhas 23](#_Toc209789929)

[8.1. Tabelas Pivot 23](#_Toc209789930)

[8.2. Crosstab 24](#_Toc209789931)

[8.3. Planilhas Excel 24](#_Toc209789932)

[8.3.1. Como ler planilhas Excel 24](#_Toc209789933)

[8.3.2. Transformar DataFrame para xlsx 24](#_Toc209789934)

# 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.

## Entendendo um pouco mais sobre gráficos de bloxplot

O **boxplot** (ou gráfico de caixa) é um dos gráficos estatísticos mais importantes porque resume, de forma **visual e compacta**, a **distribuição de um conjunto de dados numéricos**. Ele é excelente para identificar **tendências centrais, dispersão e outliers (valores atípicos)**.

* **Estrutura do boxplot**

O gráfico é formado por uma "caixa" e alguns "bigodes" (linhas). Vamos entender cada parte:

1. **Mediana (linha dentro da caixa)**
   * Representa o valor central dos dados (50%).
   * Divide o conjunto em duas metades iguais.
2. **Quartis**
   * **Q1 (1º quartil):** 25% dos dados estão abaixo dele.
   * **Q3 (3º quartil):** 75% dos dados estão abaixo dele.
   * A caixa vai de **Q1 até Q3**, mostrando onde está a maior parte dos dados.
3. **Amplitude interquartil (IQR – *Interquartile Range*)**
   * Calculada como: IQR=Q3−Q1I QR = Q3 - Q1
   * Mede a **dispersão dos dados no meio**.
4. **Bigodes (whiskers)**
   * Saem das extremidades da caixa.
   * Normalmente vão até:

Limite inferior=Q1−1.5×IQR\text{Limite inferior} = Q1 - 1.5 \times IQR Limite superior=Q3+1.5×IQR\text{Limite superior} = Q3 + 1.5 \times IQR

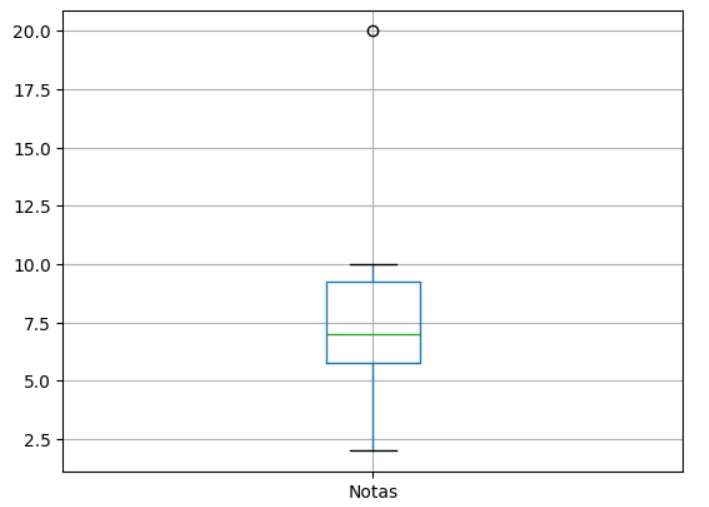
* + Ou seja, mostram os valores que ainda são considerados “normais” no conjunto.

1. **Outliers**
   * Valores que ficam **fora dos limites dos bigodes**.
   * São geralmente plotados como **pontos isolados**.

* **O que podemos identificar em um boxplot?**
  + **Distribuição dos dados:** Se os dados estão mais concentrados em uma região.
  + **Assimetria:**
    - Se a mediana está mais próxima de Q1 → cauda direita mais longa.
    - Se está mais próxima de Q3 → cauda esquerda mais longa.
  + **Dispersão:** Quanto maior a caixa, maior a variação dos dados.
* **Outliers:** Ficam visíveis de forma clara.

**🔧 Exemplo prático em Pandas**

|  |
| --- |
| import pandas as pd  import matplotlib.pyplot as plt  # Exemplo de dataset  dados = pd.DataFrame({  "Notas": [4, 5, 6, 6, 7, 7, 8, 9, 10, 10, 2, 20]  })  # Criando boxplot  dados.boxplot(column="Notas")  plt.show() |



Nesse exemplo:

* A **caixa** mostra onde estão 50% das notas.
* A **linha no meio da caixa** é a mediana.
* O número **20** provavelmente vai aparecer como um **outlier**.

Resumindo:  
O **boxplot** é um gráfico que permite, de forma rápida, ver **mediana, dispersão, assimetria e outliers** em seus dados. É uma ferramenta essencial de **análise exploratória de dados (EDA)**.

### Correlação

Correlação é um conceito **estatístico** que mede a **força e a direção da relação entre duas variáveis**.

**Em outras palavras:**

* Ela responde: **“quando uma variável muda, a outra também muda?”**
* Se sim, muda **no mesmo sentido** ou **em sentido oposto**?
* **Tipos de correlação**

1. **Correlação positiva**
   * Quando uma variável aumenta, a outra também tende a aumentar.
   * Ex.: Horas de estudo sobe → Nota na prova sobe.
2. **Correlação negativa**
   * Quando uma variável aumenta, a outra tende a diminuir.
   * Ex.: Preço de um produto subiu → Quantidade vendida diminuiu.
3. **Correlação nula (ou fraca)**
   * Não existe relação clara entre as variáveis.
   * Ex.: Número de sapatos que você tem **X** Nota na prova.

* **Como medir?**

O valor da correlação normalmente vai de **-1 a 1**:

* **+1** → Correlação perfeita positiva (relação direta).
* **-1** → Correlação perfeita negativa (relação inversa).
* **0** → Nenhuma correlação.

Exemplo de interpretação:

* **0.8** → forte correlação positiva.
* **-0.6** → moderada correlação negativa.
* **0.1** → correlação fraca.
* **Resumindo**
  + **Correlação** mostra **se e como** duas variáveis se relacionam.
  + Serve muito em **análise exploratória** para entender quais variáveis podem influenciar outras.
  + Mas **atenção**:  
    **Correlação ≠ Causalidade**  
    → Duas coisas estarem correlacionadas não significa que uma causa a outra.

### Outliers

Os *outliers* são dados que se diferenciam drasticamente de todos os outros. Em outras palavras, um outlier é um valor que foge da normalidade e que pode (e provavelmente irá) causar anomalias nos resultados obtidos por meio de algoritmos e sistemas de análise.

Entender os outliers é fundamental em uma análise de dados por pelo menos dois aspectos:

1. os outliers podem viesar negativamente todo o resultado de uma análise;
2. o comportamento dos *outliers* pode ser justamente o que está sendo procurado.

Os outliers presentes em datasets possuem diversos outros nomes, como:

* dados discrepantes;
* pontos fora da curva;
* observações fora do comum;
* anomalias;
* valores atípicos;
* entre outros.

## Estatística Descritiva

A **estatística descritiva** é o ramo da estatística que tem como objetivo **resumir, organizar e interpretar um conjunto de dados**, sem tirar conclusões além do que os dados mostram.

Pensa assim: você tem uma planilha gigante com milhares de registros. A estatística descritiva ajuda a transformar esse “mundaréu” de números em **informações compreensíveis**.

* **Principais elementos da estatística descritiva**

**1. Medidas de tendência central**

Mostram **onde os dados tendem a se concentrar**.

* **Média** → soma de todos os valores dividido pela quantidade.
* **Mediana** → valor central (quando os dados estão ordenados).
* **Moda** → valor que mais se repete.

Exemplo: notas de alunos = [5, 6, 7, 8, 9]

* Média = 7
* Mediana = 7
* Moda = não tem (todos aparecem só uma vez).

**2. Medidas de dispersão**

Mostram **o quão espalhados os dados estão**.

* **Amplitude** → diferença entre o maior e o menor valor.
* **Variância** → média dos quadrados dos desvios em relação à média.
* **Desvio padrão** → raiz quadrada da variância (quanto maior, mais dispersos os dados).

Exemplo: duas turmas têm média = 7

* Turma A: notas = [6, 7, 8] → desvio padrão baixo.
* Turma B: notas = [2, 7, 12] → desvio padrão alto.

**3. Medidas de posição**

Mostram onde certos valores estão em relação ao conjunto.

* **Quartis** → dividem os dados em 4 partes.
* **Percentis** → dividem em 100 partes.
* Muito usados em análise de desempenho (ex.: estar no “percentil 90” significa estar melhor que 90% do grupo).

**4. Distribuição**

* Mostra **como os dados estão distribuídos**.
* Pode ser simétrica (ex.: distribuição normal, em forma de sino) ou assimétrica (com caudas mais longas para um lado).

**Exemplo prático em Pandas**

|  |  |
| --- | --- |
| import pandas as pd  dados = pd.DataFrame({  "notas": [5, 6, 7, 8, 9, 10, 10]  })  # Estatística descritiva geral  print(dados.describe()) | **Saída (resumida):**  count 7.000000 # quantidade de registros  mean 7.857143 # média  std 1.574696 # desvio padrão  min 5.000000 # valor mínimo  25% 6.500000 # 1º quartil  50% 8.000000 # mediana  75% 9.500000 # 3º quartil  max 10.000000 # valor máximo |

# Trabalhando com Tabelas e Planilhas

## Tabelas Pivot

As **tabelas pivot** (ou **tabelas dinâmicas**) são uma forma de **reestruturar e resumir dados** em um DataFrame, permitindo análises rápidas e agregações personalizadas.

* **O que é uma tabela pivot?**

Pensa assim: você tem um **dataset com linhas e colunas**, e quer responder perguntas como:

* Qual foi a média de vendas por mês e por produto?
* Quantos alunos faltaram em cada turma?
* Qual o faturamento por cidade?

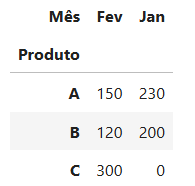
A **tabela pivot** pega esses dados brutos e reorganiza, de modo que:

* **index** → define as linhas.
* **columns** → define as colunas.
* **values** → define o que será calculado.
* **aggfunc** → define a função de agregação (soma, média, contagem, etc.).
* **Sintaxe no Pandas**

|  |
| --- |
| df.pivot\_table(  values='coluna\_valor',  index='coluna\_linha',  columns='coluna\_coluna',  aggfunc='mean'  ) |

* **values** → o que você quer calcular (ex.: "vendas").
* **index** → como você quer organizar as linhas (ex.: "produto").
* **columns** → como você quer organizar as colunas (ex.: "mês").
* **aggfunc** → função usada (ex.: sum, mean, count).
* **Exemplo prático**

|  |
| --- |
| import pandas as pd  # Criando dataset de exemplo  dados = pd.DataFrame({  "Produto": ["A", "A", "B", "B", "A", "C"],  "Mês": ["Jan", "Fev", "Jan", "Fev", "Jan", "Fev"],  "Vendas": [100, 150, 200, 120, 130, 300]  })  # Criando tabela pivot  tabela = dados.pivot\_table(  values="Vendas",  index="Produto",  columns="Mês",  aggfunc="sum",  fill\_value=0  )  print(tabela) |

****

Saída:

* **Vantagens**
  + Permite **resumir dados complexos** em poucas linhas/colunas.
  + Fácil de visualizar padrões, comparações e tendências.
  + Muito útil em **análise exploratória** e **relatórios**.

## Crosstab

O **crosstab** em Pandas é uma função que cria uma **tabela de contingência** — ou seja, uma tabela que mostra a **frequência de ocorrência** entre duas (ou mais) variáveis categóricas.

É muito usado quando você quer ver **como duas variáveis se relacionam em termos de contagem**.

* **Sintaxe básica**

pd.crosstab(index, columns)

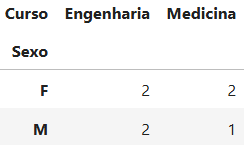
* **index** → o que vai nas linhas.
* **columns** → o que vai nas colunas.

Também é possível adicionar:

* values + aggfunc → para aplicar agregações.
* margins=True → adiciona totais gerais.
* **Exemplo simples**

|  |
| --- |
|  |

* **Saída:**



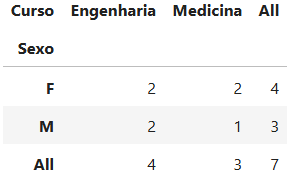
Interpretação:

* Mulheres: 2 em Engenharia, 2 em Medicina.
* Homens: 2 em Engenharia, 1 em Medicina.
* **Com totais (margins=True)**

tabela = pd.crosstab(dados["Sexo"], dados["Curso"], margins=True)

print(tabela)

* **Saída:**



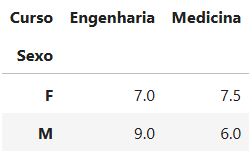
A última linha e última coluna mostram os **totais**.

* **Usando valores com agregação**

Digamos que além de contar pessoas, eu queira **somar notas**:

|  |
| --- |
|  |

* **Saída:**



Agora vemos a **média das notas** por sexo e curso.

* **Diferença principal**
  + **pivot\_table** → mais flexível, usado para **resumir valores numéricos** (somas, médias, etc.).
  + **crosstab** → mais direto, usado para **frequências e tabelas de contingência** (contagem de combinações entre categorias).

## Planilhas Excel

### Como ler planilhas Excel

### Transformar DataFrame para xlsx

# Mesclagem de Dados

No **Pandas**, mesclar dados significa **combinar DataFrames diferentes** em um só, seja para:

* juntar informações complementares,
* organizar dados que estão espalhados em várias tabelas,
* ou consolidar relatórios.

**Tipos principais de mesclagem no Pandas**

* **1. Concatenação (pd.concat)**

Usada quando você quer **empilhar tabelas**:

* **empilhar por linha** (adicionar mais registros);
* **empilhar por coluna** (adicionar mais atributos).

Exemplo:

|  |
| --- |
|  |

Saída:



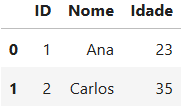
* **2. Merge (pd.merge)**

Parecido com o **JOIN do SQL**.  
Você combina DataFrames **com base em uma chave (coluna em comum)**.

Exemplo:

|  |
| --- |
| df1 = pd.DataFrame({"ID": [1, 2, 3], "Nome": ["Ana", "Carlos", "João"]})  df2 = pd.DataFrame({"ID": [1, 2, 4], "Idade": [23, 35, 40]})  df\_merge = pd.merge(df1, df2, on="ID", how="inner")  print(df\_merge) |

Saída:



Aqui só aparecem os IDs que estão nos **dois DataFrames**.

**Tipos de merge (how)**:

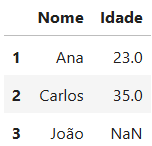
* **inner** → pega apenas o que existe nos dois DataFrames.
* **left** → mantém todos do da esquerda, e completa com dados do da direita (ou NaN se não tiver).
* **right** → o contrário do left.
* **outer** → mantém todos de ambos, preenchendo onde não existir.
* **3. Join (df.join)**

É mais simples, usado quando os índices já são as chaves.

Exemplo:

|  |
| --- |
| df1 = pd.DataFrame({"Nome": ["Ana", "Carlos", "João"]}, index=[1, 2, 3])  df2 = pd.DataFrame({"Idade": [23, 35, 40]}, index=[1, 2, 4])  df\_join = df1.join(df2, how="left")  print(df\_join) |

Saída:



* **Resumindo**
  + **concat** → junta DataFrames em sequência (por linha ou coluna).
  + **merge** → combina DataFrames usando uma chave (como no SQL JOIN).
  + **join** → atalho do merge, mas usando índices como chave.
* Uma boa prática é sempre pensar:
  + Se estou apenas **empilhando dados iguais** → concat.
  + Se estou **relacionando tabelas com chaves** → merge.
  + Se os **índices são minha chave** → join.

# SQL no Pandas

Quando a gente fala em **trabalhar SQL no Pandas**, na prática estamos falando de duas coisas principais:

* **1. Conectar o Pandas a um banco SQL**

O Pandas permite **ler dados diretamente de um banco** e até **escrever DataFrames para tabelas SQL**.  
Isso é feito com:

* **pd.read\_sql()** → lê dados de uma tabela ou de uma query.
* **df.to\_sql()** → escreve um DataFrame em uma tabela do banco.

Exemplo:

|  |
| --- |
| import pandas as pd  import sqlite3  # Criando conexão com SQLite  con = sqlite3.connect("escola.db")  # Lendo direto de uma query SQL  df = pd.read\_sql("SELECT \* FROM alunos", con)  print(df.head()) |

Aqui você já consegue usar SQL para buscar dados e depois manipular no Pandas.

* **2. Usar SQL “dentro do Pandas”**

Às vezes você quer consultar seus DataFrames usando **sintaxe SQL** em vez de métodos Pandas.  
Para isso, existem bibliotecas que ajudam, como:

* [pandasql](https://github.com/yhat/pandasql)
* [duckdb](https://duckdb.org/) (bem mais rápido e recomendado hoje)

Exemplo com **pandasql**:

|  |
| --- |
| import pandas as pd  import pandasql as ps  df = pd.DataFrame({  "id": [1, 2, 3, 4],  "nome": ["Ana", "Carlos", "João", "Maria"],  "idade": [23, 35, 19, 40]  })  # Query SQL no DataFrame  query = "SELECT nome, idade FROM df WHERE idade > 30"  resultado = ps.sqldf(query, locals())  print(resultado) |

Saída:

nome idade

0 Carlos 35

1 Maria 40

Exemplo com **duckdb**:

|  |
| --- |
| import duckdb  resultado = duckdb.query("SELECT nome, AVG(idade) as media\_idade FROM df GROUP BY nome").to\_df()  print(resultado) |

* **3. Diferença entre Pandas e SQL**
  + **SQL** → mais usado para **consultar, filtrar e agrupar** dados dentro de bancos relacionais.
  + **Pandas** → mais usado para **análise exploratória, limpeza e manipulação** de dados já carregados em memória.
* Na prática:
  + Muitas vezes você usa **SQL para extrair** e depois **Pandas para transformar/analisar**.  
    Isso é tão comum que tem até nome no mundo dos dados: **ETL (Extract, Transform, Load)**.