



**CICLO FORMATIVO**

# Python



## ARQUIVOS, PACOTES E MÓDULO

### Introdução ao manuseio de arquivos em Python

#### O que são arquivos em Python?

Em Python, um arquivo armazena dados de uma maneira permanente. Na linguagem, existem dois tipos principais:

- ➔ **Arquivos de texto**, comprehensíveis a humanos;
- ➔ **Arquivos binários**, com dados como imagens e vídeos, executáveis ou não.

#### Abrir, fechar, ler e editar arquivos

Através de *scripts* de código em Python, podemos manusear nossos arquivos, abri-los, fechá-los, editá-los e executá-los.

No exemplo abaixo, vemos uma pasta contendo um arquivo de extensão **.py (manipulando\_arquivos.py)**. No arquivo em Python, criaremos nosso primeiro trecho de código. Nele, a variável **arquivo** recebe o método **open()**, com dois parâmetros:

- ➔ **myFavoriteSound.txt**, o arquivo que desejamos abrir;
- ➔ **'modo'**, o modo de abertura do arquivo.

Tipos de modos incluem:

'**r**', o modo de leitura padrão;

'**w**', modo de escrita que cria um novo arquivo ou sobrescreve um já existente;

'**a**', modo de escrita que adiciona conteúdo ao final de um arquivo já existente;

'**r+**', modo de leitura e escrita.

É importante fechar o arquivo após o término das operações instanciando o método **close()** para encerrar sua execução:

```
● ● ●
1 # Exemplo de escrita em arquivo
2 arquivo = open('myFavoriteSound.txt', 'w')
3 arquivo.write('Bigger - Beyoncé!\n')
4 arquivo.write('If you feel insignificant, you better think again\n')
5 arquivo.write('Better wake up because you are part of something way bigger\n')
6 arquivo.close()
7
8 # Exemplo de leitura de arquivo
9 arquivo = open('myFavoriteSound.txt', 'r')
10 conteudo = arquivo.read()
11 print(conteudo)
12 arquivo.close()
```

Para executar abra o seu terminal da pasta do arquivo, e execute o seguinte comando abaixo:

```
● ● ●
1 python3 manipulando_arquivos.py
```

O resultado será a criação de um novo arquivo **myFavoriteSound.txt** dentro da pasta, com música e leitura do texto dentro do arquivo no terminal:

```
myFavoriteSound.txt ×  
myFavoriteSound.txt  
1 Bigger – Beyoncé!  
2 If you feel insignificant, you better think again  
3 Better wake up because you are part of something way bigger
```

Arquivo myFavoriteSound.txt

```
amandaprisciladasilvamandy@MacBook-Air-de-Amanda:~/Desktop$ python3 manipulando_arquivos.py  
Bigger – Beyoncé!  
If you feel insignificant, you better think again  
Better wake up because you are part of something way bigger
```

Texto do arquivo em modo leitura no terminal

## Na prática

Agora que entendemos como abrir, ler e editar arquivos em Python, um exercício prático:

- ➔ Crie um arquivo chamado **dados.txt** com seu nome e idade escritos;
- ➔ Abra o arquivo, leia seu conteúdo e exiba-o na tela.

## Explorando pacotes e módulos em Python

### O que são?

Alguns pacotes em Python são coleções, com diversos módulos relacionados entre si. Cada módulo possui um arquivo com definições e instruções em Python; cada arquivo, o nome do módulo e a extensão '**.py**'. O arquivo **calculos.py** abaixo realiza estas quatro operações:

```
● ● ●  
1 def soma(a, b):  
2     return a + b  
3  
4 def subtracao(a, b):  
5     return a - b  
6  
7 def multiplicacao(a, b):  
8     return a * b  
9  
10 def divisao(a, b):  
11     return a / b
```

### Importando pacotes e módulos

Podemos usar os módulos de Python dentro uns dos outros importando o módulo inteiro ou apenas funções específicas. Abaixo, exemplo importando um módulo inteiro.

```
● ● ●  
1 import calculos  
2  
3 print(calculos.soma(5, 3)) # Saída: 8  
4 print(calculos.divisao(10, 2)) # Saída: 5.0
```

Importando funções específicas de um módulo:

```
● ● ●  
1 from calculos import soma, multiplicacao  
2  
3 print(soma(5, 3)) # Saída: 8  
4 print(multiplicacao(5, 3)) # Saída: 15
```

## Criando e utilizando seus próprios módulos

É possível criar módulos próprios em Python. Basta criar um arquivo **.py** com as definições e instruções que quiser utilizar em outros arquivos. No exemplo a seguir, criamos um módulo **operacoes.py**, com algumas funções matemáticas:

```
● ● ●  
1 def quadrado(x):  
2     return x ** 2  
3  
4 def cubo(x):  
5     return x ** 3
```

Em seguida, podemos importar e utilizar esse módulo em outro arquivo Python:

```
● ● ●  
1 from operacoes import quadrado, cubo  
2  
3 print(quadrado(5)) # Saída: 25  
4 print(cubo(5)) # Saída: 125
```

### Na prática

Crie um módulo Python **conversoes.py** com duas funções:

➔ Uma função **celsius\_para\_fahrenheit**, que recebe uma temperatura em graus Celsius como argumento e retorna a temperatura equivalente em Fahrenheit;

➔ Outra função, **quilometros\_para\_milhas**, recebe uma distância em quilômetros como argumento e retorna a distância equivalente em milhas.

Em seguida, importe o módulo **conversoes.py** e teste as funções criadas.

## BANCO DE DADOS

### Introdução ao conceito de banco de dados

#### O que é

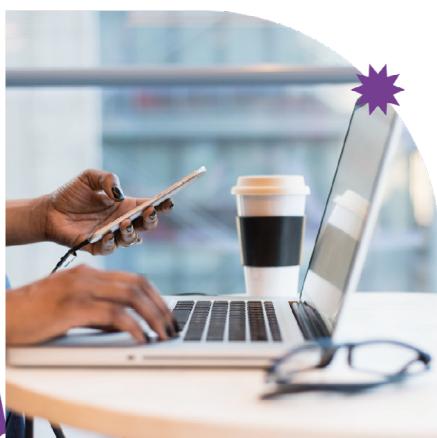
Um banco de dados é um sistema que organiza, mantém e recupera informações. Ele é composto por uma ou mais tabelas contendo campos de dados; cada tabela é composta por linhas e colunas, que podem estar relacionadas ou não, como em uma planilha.

#### Tipos de bancos de dados

Existem dois tipos de bancos de dados:

➔ **Bancos de dados relacionais** organizam dados em tabelas relacionadas entre si (exemplos incluem SQLite, MySQL, PostgreSQL, Oracle e SQL Server, dentre outros);

➔ **Bancos de dados não relacionais (NoSQL)** não seguem o modelo de tabelas usado em bancos de dados relacionais (exemplos incluem MongoDB, Cassandra e Redis, dentre outros).



## Introdução ao SQLite

SQLite é um sistema de gerenciamento de banco de dados relacional que não requer servidor separado, o que o torna simples de configurar e ideal para pequenos projetos.

## Conectando a um banco de dados SQLite

Você pode usar a extensão SQLite do VSCode para criar e interagir com um banco de dados SQLite diretamente no VSCode.

Abra o VSCode e crie um novo arquivo com a extensão .sqlite, por exemplo, **media\_escolar.db**. No arquivo, você define a estrutura do banco de dados e insere dados, conforme exemplo abaixo:



```
CREATE TABLE alunos (
    id INTEGER PRIMARY KEY,
    nome TEXT,
    idade INTEGER
);
INSERT INTO alunos (nome, idade) VALUES ('Ana', 25);
INSERT INTO alunos (nome, idade) VALUES ('Bruno', 28);
INSERT INTO alunos (nome, idade) VALUES ('Carla', 30);
```

Criamos a tabela **alunos** com três colunas:

- ➔ **id**, do tipo INTEGER, é a chave primária da tabela. Isso significa que cada linha terá um valor único para **id**;
- ➔ **nome**, do tipo TEXT, armazenará os nomes dos alunos;
- ➔ **idade**, do tipo INTEGER, armazenará as idades dos alunos.

Depois, inserimos dados na **tabela alunos** em três linhas:

- ➔ Uma linha com o nome “**Ana**”, idade **25**;
- ➔ Uma linha com o nome “**Bruno**”, idade **28**;
- ➔ Uma linha com o nome “**Carla**”, idade **30**.

Os comandos em SQL criaram uma tabela de alunos inserindo alguns dados como exemplo. O arquivo **teste.db** do banco de dados SQLite contém essa tabela e dados inseridos.

## Executando consultas SQL

Com a extensão SQLite instalada, você poderá executar consultas SQL diretamente no VSCode:

- ➔ Abra o arquivo **media\_escolar.db**;
- ➔ Clique com o botão direito em qualquer lugar do arquivo e selecione “**Run SQLite Query**”;

Você pode executar consultas SQL diretamente no editor, conforme exemplos abaixo.

```
● ● ●
1 -- Selecionando todos os registros da tabela alunos
2 SELECT * FROM alunos;
3
4 -- Atualizar a idade de um aluno
5 UPDATE alunos SET idade = 31 WHERE nome = 'Carla';
6
7 -- Excluir um aluno da tabela
8 DELETE FROM alunos WHERE nome = 'Bruno';
```

## Atualizando e excluindo dados

A figura abaixo exemplifica como atualizar ou excluir dados em uma tabela:

```
● ● ●
1 -- Atualizando a idade de um aluno
2 UPDATE alunos SET idade = 31 WHERE nome = 'Carla';
3
4 -- Excluindo um aluno da tabela
5 DELETE FROM alunos WHERE nome = 'Bruno';
```

No exemplo acima, atualizamos a idade da aluna **Carla** para 31 anos e excluímos o aluno **Bruno** da tabela. Também podemos executar consultas e acompanhar resultados selecionando todos os registros da tabela novamente:

```
● ● ●
1 -- Selecionar todos os registros da tabela alunos
2 SELECT * FROM alunos;
```

## APROFUNDANDO EM BANCO DE DADOS RELACIONAL SQL

### Conceitos avançados de banco de dados relacional

#### Modelagem de dados

Este processo cria um modelo de dados para determinado sistema com informações vindas da abstração de entidades relevantes e suas relações:

- ⇒ **Entidade** representa um objeto do mundo real, como uma pessoa, um lugar, um produto, etc.;
- ⇒ **Atributo** é a característica que descreve uma entidade, como nome, idade, preço, etc.;
- ⇒ **Relacionamento** é o tipo de associação entre entidades.

#### Chaves primárias e estrangeiras

Chaves primárias e estrangeiras são conceitos fundamentais em bancos de dados relacionais. Um relacionamento entre tabelas, por exemplo, é estabelecido usando **chaves primárias e estrangeiras**:

- ⇒ **Chave Primária** representa um atributo, ou conjunto de atributos, que identifica individualmente cada linha em uma tabela para garantir a integridade dos dados. É usada como referência em relacionamentos com outras tabelas;
- ⇒ **Chave Estrangeira** representa um atributo de uma tabela, ou um conjunto de atributos, relacionado à chave primária de outra. Ela estabelece um relacionamento entre as tabelas garantindo a integridade referencial e a construção de consultas complexas.

## Relacionamentos entre tabelas

Suponha que tenhamos duas tabelas, **alunos** e **cursos**, e queiramos registrar os cursos nos quais cada aluno está matriculado. Podemos fazê-lo usando uma chave estrangeira na tabela **alunos** que se refira à chave primária da tabela **cursos**.

```
CREATE TABLE alunos (
    id INTEGER PRIMARY KEY,
    nome TEXT,
    curso_id INTEGER,
    FOREIGN KEY (curso_id) REFERENCES cursos(id)
);
CREATE TABLE cursos (
    id INTEGER PRIMARY KEY,
    nome TEXT
);
```

No exemplo acima, criamos a tabela **alunos**, na qual:

- ➔ **id INTEGER PRIMARY KEY** define um campo chamado **id** na tabela **alunos** como chave primária (**PRIMARY KEY**). Cada registro na tabela **alunos** terá um valor único para o campo **id**;
- ➔ **nome TEXT** define um campo **nome** na tabela **alunos**, como um texto (**TEXT**). Este campo armazenará o nome do aluno;
- ➔ **curso\_id INTEGER** define um campo **curso\_id** na tabela **alunos**, como um número inteiro (**INTEGER**). Este campo armazena o identificador do curso ao qual o aluno estiver associado.

Na tabela **cursos**:

- ➔ **id INTEGER PRIMARY KEY** define um campo **id** na tabela **cursos**, como uma chave primária (**PRIMARY KEY**). Cada registro na tabela **cursos** terá um valor único para o campo **id**;
- ➔ **nome TEXT** define um campo **nome** na tabela **cursos**, como um texto (**TEXT**). O campo armazenará o nome do curso.

Em resumo, as declarações **CREATE TABLE** foram usadas para criar duas tabelas em um banco de dados SQLite: uma armazenou informações sobre os alunos (**alunos**); outra, informações sobre os cursos (**cursos**). O campo **curso\_id**, na tabela **alunos**, é uma chave estrangeira, que estabelece uma relação entre duas tabelas permitindo que cada aluno seja associado a um curso específico.

## Consultas avançadas em SQL

Agora é hora de explorar consultas mais avançadas para extrair informações específicas de um banco de dados. Aprenderemos como combinar várias cláusulas SQL para consultas mais complexas e poderosas, que nos permitam obter *insights* detalhados a partir dos nossos dados.

### Cláusulas SQL (SELECT, WHERE, JOIN, GROUP BY, HAVING, ORDER BY)

Cláusulas em SQL são elementos fundamentais que recuperam e manipulam os dados em um banco de dados relacional. Cada cláusula desempenha um papel específico na construção de consultas SQL - que, por sua vez, atendem a demandas de análise e extração de dados.

Neste capítulo, vamos explorar as cláusulas **SELECT**, **WHERE**, **JOIN**, **GROUP BY**, **HAVING**, e **ORDER BY**, aprender como utilizá-las individualmente e em conjunto, para realizar consultas complexas que nos permitam obter informações precisas e relevantes a partir de um banco de dados:

➔ **SELECT** é usada para selecionar dados em uma ou mais tabelas;

```
● ● ●
1 -- Selecionar todos os alunos
2 SELECT * FROM alunos;
3
4 -- Selecionar apenas os nomes dos alunos
5 SELECT nome FROM alunos;
6
7 -- Selecionar nome e idade dos alunos
8 SELECT nome, idade FROM alunos;
```

➔ **WHERE** é usada para filtrar registros;

```
● ● ●  
1 -- Selecionar alunos com idade maior que 25 anos  
2 SELECT * FROM alunos WHERE idade > 25;  
3  
4 -- Selecionar alunos do curso de Matemática  
5 SELECT * FROM alunos WHERE curso_id = 1;
```

➔ **JOIN** é usada para combinar registros de duas ou mais tabelas;

```
● ● ●  
1 -- Selecionar nome do aluno e nome do curso que ele está cursando  
2 SELECT alunos.nome, cursos.nome  
3 FROM alunos  
4 JOIN cursos ON alunos.curso_id = cursos.id;
```

➔ **GROUP BY** é usada para agrupar registros com base em um critério específico;

```
● ● ●  
1 -- Contar o número de alunos em cada curso  
2 SELECT curso_id, COUNT(*) as num_alunos  
3 FROM alunos  
4 GROUP BY curso_id;  
5
```

➔ **HAVING** é usada para filtrar registros agrupados;

```
● ● ●  
1 -- Selecionar cursos com mais de 3 alunos  
2 SELECT curso_id, COUNT(*) as num_alunos  
3 FROM alunos  
4 GROUP BY curso_id  
5 HAVING COUNT(*) > 3;
```

➔ **ORDER BY** é usada para classificar resultados.

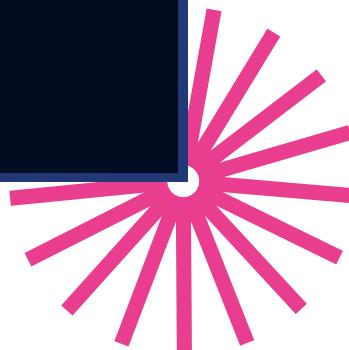
```
● ● ●  
1 -- Selecionar alunos ordenados por idade de forma decrescente  
2 SELECT * FROM alunos ORDER BY idade DESC;  
3  
4 -- Selecionar cursos ordenados por nome de forma crescente  
5 SELECT * FROM cursos ORDER BY nome;
```

## Funções agregadas

As funções agregadas realizam cálculos em um conjunto de valores e retornam um único valor:

- ➔ **COUNT** retorna o número de linhas;
- ➔ **SUM** retorna a soma dos valores;
- ➔ **AVG** retorna a média dos valores;
- ➔ **MIN** retorna o valor mínimo;
- ➔ **MAX** retorna o valor máximo.

```
● ● ●  
1 -- Contar o número de alunos com idade maior que 25 anos  
2 SELECT COUNT(*) AS num_alunos_maiores_de_idade FROM alunos WHERE idade > 25;  
3  
4 -- Calcular a soma das idades de todos os alunos  
5 SELECT SUM(idade) AS soma_idades FROM alunos;  
6  
7 -- Calcular a média das idades de todos os alunos  
8 SELECT AVG(idade) AS media_idades FROM alunos;  
9  
10 -- Encontrar a idade mínima entre todos os alunos  
11 SELECT MIN(idade) AS idade_minima FROM alunos;  
12  
13 -- Encontrar a idade máxima entre todos os alunos  
14 SELECT MAX(idade) AS idade_maxima FROM alunos;
```



## Subconsultas

Uma subconsulta é uma consulta dentro de outra. Abaixo, exemplo de seleção de dados de múltiplas tabelas usando **JOIN**:

```
● ● ●  
1 SELECT alunos.nome, cursos.nome  
2 FROM alunos  
3 JOIN cursos ON alunos.curso_id = cursos.id;
```

Agrupamos e contamos o número de alunos em cada curso:

```
● ● ●  
1 SELECT cursos.nome, COUNT(alunos.id) AS total_alunos  
2 FROM cursos  
3 JOIN alunos ON cursos.id = alunos.curso_id  
4 GROUP BY cursos.nome;
```

Selecionamos os cursos com mais de cinco alunos matriculados:

```
● ● ●  
1 SELECT cursos.nome, COUNT(alunos.id) AS total_alunos  
2 FROM cursos  
3 JOIN alunos ON cursos.id = alunos.curso_id  
4 GROUP BY cursos.nome  
5 HAVING COUNT(alunos.id) > 5;
```

Selecionamos o curso com mais alunos matriculados:

```
● ● ●  
1 SELECT cursos.nome, COUNT(alunos.id) AS total_alunos  
2 FROM cursos  
3 JOIN alunos ON cursos.id = alunos.curso_id  
4 GROUP BY cursos.nome  
5 ORDER BY total_alunos DESC  
6 LIMIT 1;
```

## TRATAMENTO DE DADOS UTILIZANDO PANDAS E NUMPY

### O que é o Pandas?

Pandas é uma biblioteca de código aberto com estruturas de dados de alto desempenho e ferramentas de análise de dados para a linguagem de programação Python. Ela fornece estruturas de dados flexíveis como o DataFrame, semelhantes a tabelas de banco de dados, que permitem manipular dados de forma rápida e fácil. É usada principalmente em análise, limpeza, preparação e visualização de dados.

### Estruturas de dados: Séries e DataFrame

Duas das estruturas de dados fundamentais do Pandas são Series e DataFrames:

➔ **Séries** é uma estrutura de dados unidimensional que armazena dados de qualquer tipo (números inteiros, de ponto flutuante, strings, etc.). Semelhante a uma matriz unidimensional ou coluna em uma planilha. Series possuem índice, que permite acessar e manipular os dados de forma eficiente. Abaixo, exemplo de criação de Series em Pandas;

```
● ● ●
1 # Criando uma série a partir de uma lista
2 idades = pd.Series([25, 30, 35, 40, 45])
3 print(idades)
4
5 # Criando uma série com um índice personalizado
6 notas = pd.Series([8, 7, 9, 6, 8], index=['Ana', 'Bruno', 'Carla', 'Daniel', 'Eva'])
7 print(notas)
8
9 # Criando uma série a partir de um dicionário
10 dic_notas = {'Ana': 8, 'Bruno': 7, 'Carla': 9, 'Daniel': 6, 'Eva': 8}
11 notas = pd.Series(dic_notas)
12 print(notas)
```

➔ **DataFrame** é uma estrutura de dados bidimensional que armazena dados de diferentes tipos (números inteiros, números de ponto flutuante, *strings*, etc.) em linhas e colunas, como uma planilha ou tabela de banco de dados. DataFrames também possuem índice para acessar e manipular os dados de forma eficiente. Abaixo, exemplo de DataFrames em Pandas.

```
● ● ●
1 # Criando um DataFrame a partir de um dicionário de listas
2 data = {'Nome': ['Ana', 'Bruno', 'Carla', 'Daniel', 'Eva'],
3          'Idade': [25, 30, 35, 40, 45],
4          'Nota': [8, 7, 9, 6, 8]}
5 df = pd.DataFrame(data)
6 print(df)
7
8 # Criando um DataFrame a partir de uma lista de dicionários
9 data = [{('Nome': 'Ana', 'Idade': 25, 'Nota': 8),
10          ('Nome': 'Bruno', 'Idade': 30, 'Nota': 7),
11          ('Nome': 'Carla', 'Idade': 35, 'Nota': 9),
12          ('Nome': 'Daniel', 'Idade': 40, 'Nota': 6),
13          ('Nome': 'Eva', 'Idade': 45, 'Nota': 8}]
14 df = pd.DataFrame(data)
15 print(df)
```

Operações com Series e DataFrames em Pandas:

```
● ● ●
1 # Acessando elementos de uma série
2 print(notas['Ana'])
3
4 # Acessando elementos de um DataFrame
5 print(df['Nome'])
6 print(df[['Nome', 'Nota']])
7
8 # Filtrando linhas com base em uma condição
9 print(df[df['Idade'] > 30])
10
11 # Adicionando uma nova coluna
12 df['Sexo'] = ['F', 'M', 'F', 'M', 'F']
13 print(df)
14
15 # Calculando estatísticas básicas
16 print(notas.mean()) # Média das notas
17 print(df['Idade'].max()) # Idade máxima no DataFrame
```

## Leitura e escrita de dados

Pandas oferece várias funções para ler e escrever dados de/para diferentes formatos de arquivo, como CSV, Excel, SQL, JSON, e outros, o que lhe permite importar dados de diferentes fontes para análises ou exportar resultados de análises para compartilhar com outras pessoas. Abaixo, exemplo de leitura de dados:

```
● ● ●
1 import pandas as pd
2
3 # Ler dados de um arquivo CSV
4 dados_csv = pd.read_csv('dados.csv')
5
6 # Ler dados de um arquivo Excel
7 dados_excel = pd.read_excel('dados.xlsx')
8
9 # Ler dados de uma tabela SQL
10 import sqlite3
11 conexao = sqlite3.connect('banco.db')
12 dados_sql = pd.read_sql_query('SELECT * FROM tabela', conexao)
13
14 # Ler dados de um arquivo JSON
15 dados_json = pd.read_json('dados.json')
```

Exemplo de escrita de dados:

```
● ● ●
1 # Escrever dados em um arquivo CSV
2 dados_csv.to_csv('dados_novos.csv', index=False)
3
4 # Escrever dados em um arquivo Excel
5 dados_excel.to_excel('dados_novos.xlsx', index=False)
6
7 # Escrever dados em uma tabela SQL
8 dados_sql.to_sql('tabela_nova', conexao, if_exists='replace', index=False)
9
10 # Escrever dados em um arquivo JSON
11 dados_json.to_json('dados_novos.json', orient='records')
12
```

## Indexação e seleção de dados

Uma das características mais poderosas da biblioteca Pandas é sua capacidade de indexação e seleção de dados, que permite acessar, filtrar e manipular dados de forma eficiente.

Existem várias formas de indexar e selecionar dados em um DataFrame do Pandas: indexação por rótulos de linhas e colunas, indexação posicional, e seleção condicional. Abaixo, indexação por rótulos e posicional:

```
● ● ●
1 import pandas as pd
2
3 # Criando um DataFrame de exemplo
4 data = {'A': [1, 2, 3, 4, 5],
5          'B': [10, 20, 30, 40, 50]}
6 df = pd.DataFrame(data, index=['a', 'b', 'c', 'd', 'e'])
7
8 # Acesso a uma coluna por rótulo
9 print(df['A'])
10
11 # Acesso a uma coluna por posição
12 print(df.iloc[:, 0])
13
14 # Acesso a uma linha por rótulo
15 print(df.loc['b'])
16
17 # Acesso a uma linha por posição
18 print(df.iloc[1])
19
```

Exemplo de seleção condicional:

```
● ● ●
1 # Seleção de linhas com base em uma condição
2 print(df[df['A'] > 2])
3
4 # Seleção de células com base em uma condição
5 print(df.loc[df['A'] > 2, 'B'])
6
```

Exemplo de indexação e atribuição de dados:

```
● ● ●  
1 # Atribuição de valor a uma célula específica  
2 df.at['c', 'B'] = 35  
3 print(df)  
4  
5 # Atribuição de valor a uma coluna inteira  
6 df['C'] = [100, 200, 300, 400, 500]  
7 print(df)  
8
```

## Operações básicas de manipulação de dados

Pandas oferece uma ampla variedade de operações básicas para manipular e transformar dados em um DataFrame, como adicionar ou remover colunas, filtrar dados, ordenar dados e muito mais. Exemplo de adição e remoção de colunas:

```
● ● ●  
1 import pandas as pd  
2  
3 # Criando um DataFrame de exemplo  
4 data = {'A': [1, 2, 3, 4, 5],  
5          'B': [10, 20, 30, 40, 50]}  
6 df = pd.DataFrame(data)  
7  
8 # Adicionando uma nova coluna  
9 df['C'] = [100, 200, 300, 400, 500]  
10 print(df)  
11  
12 # Removendo uma coluna  
13 df.drop('C', axis=1, inplace=True)  
14 print(df)  
15
```

Exemplo de filtragem de dados:

```
● ● ●  
1 # Filtragem de linhas com base em uma condição  
2 print(df[df['A'] > 2])  
3  
4 # Filtragem de células com base em uma condição  
5 print(df.loc[df['A'] > 2, 'B'])  
6
```

Exemplo de ordenação de dados:

```
● ● ●  
1 # Ordenando o DataFrame por uma coluna  
2 print(df.sort_values(by='A'))  
3  
4 # Ordenando o DataFrame por múltiplas colunas  
5 print(df.sort_values(by=['A', 'B']))  
6
```

Exemplo de renomeação de colunas:

```
● ● ●  
1 # Renomeando uma coluna  
2 df.rename(columns={'A': 'Coluna_A'}, inplace=True)  
3 print(df)  
4
```

Exemplo de “resetamento” de índice:

```
● ● ●  
1 # Resetando o índice  
2 df.reset_index(drop=True, inplace=True)  
3 print(df)  
4
```



## O que é o Numpy?

NumPy é uma biblioteca fundamental para computação numérica em Python. Ela fornece um objeto de matriz multidimensional de alto desempenho e ferramentas para trabalhar com essas matrizes. É eficiente para operações numéricas, como operações matriciais e cálculos de álgebra linear. É usado principalmente para manipulação de *arrays* numéricos e cálculos matemáticos.

### Criação e manipulação em operações básicas de arrays Numpy

Os *arrays* NumPy são estruturas de dados fundamentais para computação numérica em Python, por oferecerem uma maneira eficiente de armazenar e manipular dados numéricos em várias dimensões.

Nesta seção, vamos explorar a criação de *arrays* NumPy, realizar operações básicas, aprender como selecionar e manipular elementos em um *array*. Exemplo de criação de *arrays* NumPy:



```
1 import numpy as np
2
3 # Criando um array a partir de uma lista
4 arr1 = np.array([1, 2, 3, 4, 5])
5 print(arr1)
6
7 # Criando um array de zeros
8 arr_zeros = np.zeros(5)
9 print(arr_zeros)
10
11 # Criando um array de uns
12 arr_ones = np.ones(5)
13 print(arr_ones)
14
15 # Criando um array de valores sequenciais
16 arr_seq = np.arange(1, 10, 2)
17 print(arr_seq)
18
19 # Criando um array de números aleatórios
20 arr_rand = np.random.rand(5)
21 print(arr_rand)
22
```

## Operações básicas com *arrays*:

```
● ● ●  
1 # Operações matemáticas com arrays  
2 arr2 = np.array([6, 7, 8, 9, 10])  
3 print(arr1 + arr2)  
4 print(arr1 * arr2)  
5 print(arr1.dot(arr2)) # Produto escalar  
6  
7 # Redimensionando um array  
8 arr3 = np.arange(1, 10)  
9 arr3 = arr3.reshape(3, 3)  
10 print(arr3)  
11  
12 # Transpondo um array  
13 print(arr3.T)
```

## Indexação e seleção de elementos em *arrays*:

```
● ● ●  
1 # Acessando elementos de um array  
2 print(arr1[0])  
3  
4 # Acessando fatias de um array  
5 print(arr1[1:4])  
6  
7 # Alterando elementos de um array  
8 arr1[0] = 10  
9 print(arr1)
```



## Explorando funcionalidades avançadas em Pandas e Numpy

Pandas e NumPy oferecem uma variedade de funcionalidades avançadas para análise de dados, computação numérica e manipulação de *arrays*.

Nesta seção, vamos explorar algumas dessas funcionalidades avançadas.

### Trabalhando com dados ausentes em Pandas

Pandas oferece métodos eficientes para lidar com dados ausentes:

- ➔ **NaN (Not a Number):**
- ➔ **isnull()** retorna um DataFrame de valores *booleanos* indicando células com valores nulos;
- ➔ **dropna()** permite remover linhas ou colunas contendo valores nulos;
- ➔ **fillna()** permite preencher valores nulos com um valor específico, como média dos dados (exemplo abaixo).

```
● ● ●
1 import pandas as pd
2 import numpy as np
3
4 # Criando um DataFrame com dados ausentes
5 data = {'A': [1, np.nan, 3, np.nan, 5],
6          'B': [np.nan, 2, 3, np.nan, 5],
7          'C': [1, 2, np.nan, np.nan, 5]}
8 df = pd.DataFrame(data)
9 print(df)
10
11 # Verificando valores nulos
12 print(df.isnull())
13
14 # Removendo linhas com valores nulos
15 df.dropna(inplace=True)
16 print(df)
17
18 # Preenchendo valores nulos com a média
19 df.fillna(df.mean(), inplace=True)
20 print(df)
```

## Aplicando funções a dados em Pandas

Pandas também permite aplicar funções a todos os elementos de um DataFrame de maneira eficiente:

- ➔ **apply()** aplica uma função a cada elemento do DataFrame (exemplo abaixo).

```
● ● ●  
1 # Aplicando uma função a todos os elementos de um DataFrame  
2 df = pd.DataFrame(np.random.randn(5, 3), columns=['A', 'B', 'C'])  
3 print(df)  
4  
5 # Aplicando a função exp() a todos os elementos  
6 print(df.apply(np.exp))
```

## Operações de agrupamento em Pandas

Pandas permite ainda agrupar dados com base em uma ou mais colunas e realizar operações em grupos:

- ➔ **groupby()** agrupa dados com base em uma ou mais chaves;
- ➔ operações como **mean()**, **sum()**, **count()** podem ser aplicadas a grupos de dados (exemplo abaixo).

```
● ● ●  
1 # Criando um DataFrame de exemplo  
2 data = {'Key': ['A', 'B', 'A', 'B', 'A'],  
3          'Value': [1, 2, 3, 4, 5]}  
4 df = pd.DataFrame(data)  
5  
6 # Calculando a média dos valores agrupados por chave  
7 print(df.groupby('Key').mean())
```

## Fusão e concatenação de DataFrames em Pandas

Pandas oferece ainda métodos para combinar DataFrames por meio de concatenação e mesclagem:

- ⇒ **concat()** concatena DataFrames ao longo de um eixo;
- ⇒ **merge()** mescla DataFrames com base em uma ou mais colunas:

```
● ● ●  
1 # Concatenando DataFrames verticalmente  
2 df1 = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]})  
3 df2 = pd.DataFrame({'A': [7, 8, 9], 'B': [10, 11, 12]})  
4 result = pd.concat([df1, df2])  
5 print(result)  
6  
7 # Mesclando DataFrames  
8 df1 = pd.DataFrame({'key': ['A', 'B', 'C'], 'value': [1, 2, 3]})  
9 df2 = pd.DataFrame({'key': ['A', 'B', 'D'], 'value': [4, 5, 6]})  
10 result = pd.merge(df1, df2, on='key', how='outer')  
11 print(result)
```

## Funções universais Numpy (ufuncs)

As funções universais NumPy **ufuncs** operam em *arrays* NumPy de maneira rápida e eficiente. Exemplos incluem funções matemáticas como **sqrt()**, **exp()**, **sin()**, **cos()**, etc. (abaixo).

```
● ● ●  
1 # Operações vetorizadas com NumPy  
2 arr = np.array([1, 2, 3, 4, 5])  
3 print(np.sqrt(arr))  
4 print(np.exp(arr))
```



## Operações de álgebra linear em Numpy

O NumPy oferece uma ampla variedade de funções para operações de álgebra linear:

- ➔ **dot()** representa multiplicação de matrizes;
- ➔ **linalg.svd()** representa decomposição de valores singulares (SVD) (abaixo).

```
1 # Multiplicação de matrizes
2 A = np.array([[1, 2], [3, 4]])
3 B = np.array([[5, 6], [7, 8]])
4 print(np.dot(A, B))
5
6 # Decomposição de valores singulares (SVD)
7 arr = np.random.randn(3, 3)
8 U, S, V = np.linalg.svd(arr)
9 print(U)
10 print(S)
11 print(V)
```

## Geração de números aleatórios em Numpy

O NumPy também oferece funções para gerar números aleatórios de várias distribuições:

- ➔ **rand()** gera números aleatórios com distribuição uniforme entre 0 e 1;
- ➔ **randn()** gera números aleatórios com distribuição normal;
- ➔ **randint()** gera números inteiros aleatórios em um intervalo especificado (abaixo).

```
1 # Geração de números aleatórios
2 print(np.random.rand(5)) # Números aleatórios entre 0 e 1
3 print(np.random.randn(5)) # Números aleatórios com distribuição normal
4 print(np.random.randint(1, 10, 5)) # Números aleatórios inteiros entre 1 e 10
```

## ESTATÍSTICA COM PHYTON: PROBABILIDADE, AMOSTRAGEM E TESTES DE HIPÓTESES

### Conceitos básicos de estatística em Python

#### Distribuições de probabilidade

A Estatística é fundamental para a análise de dados, pois permite extrair *insights* significativos e tomar decisões informadas. Com o Python, podemos aplicar uma ampla variedade de técnicas estatísticas para analisar e interpretar dados.

#### Probabilidade

A probabilidade é a medida da chance de um evento ocorrer. Em Python, podemos usar a biblioteca Numpy para gerar números aleatórios e calcular probabilidades, conforme exemplo abaixo.

```
● ● ●
1 import numpy as np
2
3 # Lançamento de uma moeda
4 # 0 representa cara, 1 representa coroa
5 lancamentos = np.random.randint(0, 2, size=100)
6 prob_cara = np.mean(lancamentos == 0)
7 prob_coroa = np.mean(lancamentos == 1)
8
9 print(f'Probabilidade de sair cara: {prob_cara}')
10 print(f'Probabilidade de sair coroa: {prob_coroa}')
```

## Amostragem

Amostragem envolve a seleção de uma parte representativa de uma população maior para análise. Em Python, podemos usar a biblioteca Numpy para criar amostras aleatórias de dados, conforme abaixo.

```
● ● ●  
1 # Criando uma amostra aleatória de uma população  
2 populacao = np.random.randint(0, 100, size=1000)  
3 amostra = np.random.choice(populacao, size=100, replace=False)  
4  
5 print('Média da população:', np.mean(populacao))  
6 print('Média da amostra:', np.mean(amostra))
```

## Testes de hipóteses

Testes de hipóteses são procedimentos estatísticos que permitem tomar decisões sobre as propriedades de uma população com base em uma amostra de dados. Em Python, a biblioteca SciPy fornece uma variedade de testes estatísticos para realizar testes de hipóteses, conforme abaixo.

```
● ● ●  
1 from scipy import stats  
2  
3 # Teste de hipótese para média  
4 # Vamos testar se a média da nossa amostra é significativamente diferente de 50  
5 amostra = np.random.normal(loc=55, scale=10, size=100)  
6 t_stat, p_valor = stats.ttest_1samp(amostra, 50)  
7  
8 print('Estatística t:', t_stat)  
9 print('Valor p:', p_valor)  
10 if p_valor < 0.05:  
11     print('Rejeitamos a hipótese nula - A média da amostra é significativamente diferente de 50')  
12 else:  
13     print('Não há evidências suficientes para rejeitar a hipótese nula')
```





## REFERÊNCIAS

### Python

[Documentação oficial do Python](#)

### Livros

“Python para Análise de Dados,” por Wes McKinney

“Python Fluente,” por Luciano Ramalho

“Automate the Boring Stuff with Python,” por Al Sweigart

### Manipulação de Arquivos em Python

[Documentação oficial do Python: Manipulação de Arquivos](#)

[Tutorial W3Schools: Manipulação de Arquivos em Python](#)

### Banco de Dados Relacional (SQLite) em Python:

[Documentação oficial do SQLite3 em Python](#)

### Consultas Avançadas em SQL

Tutorial W3Schools: SQL Tutorial

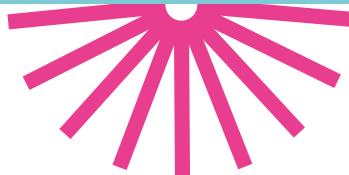
W3Schools SQL Tutorial

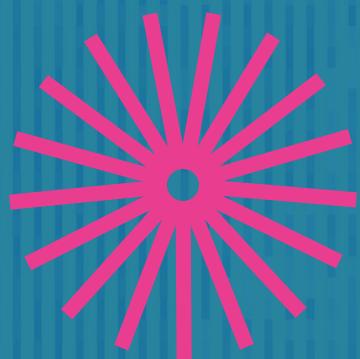
Tutorial Mode Analytics: SQL Tutorial

SQL Tutorial

### Modelagem de Dados

[Tutorial W3Schools: Modelo de Dados](#)





# CICLO FORMATIVO

---

# Python

