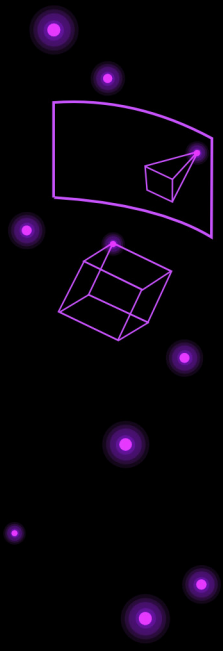
AKCIT
CENTRO DE COMPETÊNCIA EMBRAPII
EM TECNOLOGIAS IMERSIVAS

Python para Processamento de Dados

Autoria:
Otávio Calaça Xavier



Organizadores:
Renata Dutra Braga
Taciana Novo Kudo
Deborah Silva Alves Fernandes
Cristiane Bastos Rocha Ferreira
Arlindo Rodrigues Galvão Filho

Cegraf UFG

Python para Processamento de Dados

Autoria:
Otávio Calaça Xavier

Organizadores:
Renata Dutra Braga
Taciana Novo Kudo
Deborah Silva Alves Fernandes
Cristiane Bastos Rocha Ferreira
Arlindo Rodrigues Galvão Filho

Cegraf UFG

2024

Revisão Técnica
Cristiane Bastos Rocha Ferreira
Deborah Silva Alves Fernandes

Revisão Editorial
Ana Laura de Sene Amâncio Zara Brisolla

Capa
Iuri Vaz Miranda

Editoração Eletrônica
Luma Wanderley de Oliveira
Layane Grazielle Souza Dias

https://doi.org/10.5216/XAV.pyt.ebook.978-85-495-1014-3/2024

Dados Internacionais de Catalogação na Publicação (CIP)
(Câmara Brasileira do Livro, SP, Brasil)

Xavier, Otávio Calaça
Python para processamento de dados [livro eletrônico] / Otávio Calaça Xavier ; organização Renata Dutra Braga ... [et al.]. -- Goiânia, GO : Cegraf UFG, 2024.
PDF

Bibliografia.
ISBN 978-85-495-1014-3

1. Ciência da Computação 2. Interface de programas aplicativos (Software) 3. Processamento de dados 4. Python (Linguagem de programação para computadores) I. Braga, Renata Dutra. II. Título.

24-239892 CDD-005.133

Índices para catálogo sistemático:

1. Python : Linguagem de programação : Computadores : Processamento de dados 005.133

Aline Graziele Benitez - Bibliotecária - CRB-1/3129

Python para Processamento de Dados

Instituições responsáveis

Universidade Federal de Goiás (UFG)

Centro de Competência Embrapii em Tecnologias Imersivas, denominado AKCIT (Advanced Knowledge Center for Immersive Technologies)

Centro de Excelência em Inteligência Artificial (CEIA)

Instituições financiadoras

Empresa Brasileira de Pesquisa e Inovação Industrial (Embrapii)

Governo do Estado de Goiás

Empresas parceiras do AKCIT

Apoio

Universidade Federal de Goiás (UFG)

Pró-Reitoria de Pesquisa e Inovação (PRPI-UFG)

Instituto de Informática (INF-UFG)

Lista de Abreviaturas e Siglas

API	*Application Programming Interface* - Interface de Programação de Aplicações
CSV	*Comma-Separated Values* - Valores Separados por Vírgula
EDA	*Exploratory Data Analysis* - Análise Exploratória de Dados
IDE	*Integrated Development Environment* - Ambiente de Desenvolvimento Integrado
JSON	*JavaScript Object Notation* - Notação de Objetos JavaScript
KDE	*Kernel Density Estimation - Estimativa de Densidade do Kernel*
NumPy	*Numerical Python - Python* Numérico
SQL	*Structured Query Language* - Linguagem de Consulta Estruturada
UFG	Universidade Federal de Goiás
ufunc	*Universal Function* - Função Universal

Lista de Figuras

Lista de Códigos

Sumário

Apresentação

Prezado(a) Participante,

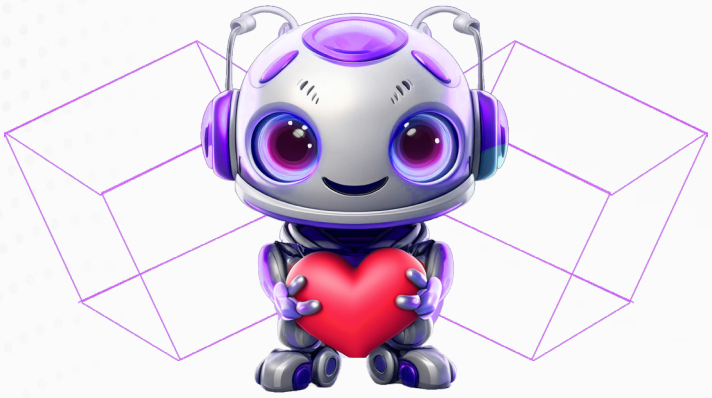
Seja bem-vindo(a) ao Microcurso ***Python* para Processamento de Dados**! Esse Microcurso faz parte da Coleção Formação e Capacitação do Centro de Competências Imersivas, uma parceria entre a Embrapii e a Universidade Federal de Goiás (UFG).

O objetivo desse Microcurso é capacitar os(as) participantes a utilizarem eficazmente o *Python*, uma das linguagens de programação mais populares e versáteis para processamento e análise de dados. Com o aumento do volume de dados gerados diariamente e a necessidade crescente de tomar decisões baseadas em dados reais, a habilidade de manipular e entender esses dados se torna essencial em várias áreas de atuação.

Ao longo deste *ebook*, exploraremos como as bibliotecas *Python* como *NumPy*, *Pandas*, *Matplotlib* e *Seaborn* podem ser utilizadas para realizar desde operações básicas de manipulação de dados até análises estatísticas complexas e criação de visualizações impactantes. Discutiremos também conceitos fundamentais de processamento de dados, manipulação e análise de dados tabulares, e apresentaremos visualizações informativas que auxiliarão na interpretação e comunicação dos resultados de análises.

A oferta deste Microcurso foi motivada pela crescente demanda por profissionais capazes de transformar dados brutos em percepções valiosas. Assim, fornece uma base sólida para quem deseja aprofundar-se no campo da ciência de dados ou otimizar processos em suas áreas de atuação.

Esperamos que este Microcurso não apenas aumente sua proficiência em *Python* para análise de dados, mas também inspire você a aplicar esses conhecimentos em sua carreira e em estudos futuros.



Desejamos um excelente estudo!

Python para Processamento de Dados

Unidade I

Processamento de Dados Estruturados Multidimensionais com *NumPy*

Unidade I - Processamento de Dados Estruturados Multidimensionais com *NumPy*

1.1 Introdução ao *NumPy*

NumPy, que significa *Numerical Python*, é uma das bibliotecas mais fundamentais para a computação numérica em *Python*. Desenvolvida inicialmente em 2005 por Travis Oliphant, a biblioteca é projetada para efetuar operações matemáticas complexas de maneira eficiente e com sintaxe relativamente simples. É um pilar para várias outras bibliotecas de ciência de dados, incluindo *Pandas*, Matplotlib, Scikit-learn, entre outras.

O coração do *NumPy* é o *array* multidimensional ou `ndarray`. Esse objeto é uma coleção de elementos, tipicamente números, do mesmo tipo, indexados por uma tupla de inteiros positivos que podem ser manipulados de forma vetorizada. A vetorização de operações, que elimina *loops* explícitos e usa operações aplicadas diretamente a *arrays*, é uma das razões pela qual o NumPy é tão rápido e eficaz.

NumPy não só otimiza a execução de cálculos numéricos com alto desempenho por meio de suas estruturas de dados internas, mas também serve como uma interface para rotinas em C e Fortran. Isso permite que operações computacionalmente intensivas sejam executadas fora do ambiente *Python*, interpretadas e de alto nível, aumentando significativamente a velocidade de processamento.

No contexto da ciência de dados, o *NumPy* é frequentemente utilizado para tarefas de preparação de dados. Isso inclui limpar, transformar e normalizar os dados — passos essenciais em qualquer processo de análise de dados. Usando o *NumPy*, essas tarefas são executadas de forma eficaz, preparando os dados para análises mais detalhadas e precisas.

O uso de *NumPy* se estende além da ciência de dados, beneficiando engenheiros, físicos e matemáticos. Ele é usado em quase todos os aspectos da análise científica e técnica, desde simulações até processamento de imagens, mostrando sua versatilidade e capacidade de lidar com uma ampla gama de desafios numéricos.

Um dos pontos fortes do *NumPy* é a vasta gama de funcionalidades matemáticas que ele proporciona. Possibilita não somente a realização de operações matemáticas básicas, como soma e multiplicação, mas suporta também processos complexos como transformadas de *Fourier* e operações de álgebra linear. Essa diversidade de ferramentas torna o *NumPy* uma excelente escolha para qualquer tarefa que exija cálculos matemáticos intensivos.

A comunidade de desenvolvedores e usuários do *NumPy* é grande e ativa. Isso garante que a biblioteca esteja constantemente sendo atualizada e melhorada. Também fornece um recurso rico para novos(as) usuários(as) aprenderem por meio de exemplos e tutoriais disponíveis publicamente.

Antes de começarmos a explorar as funcionalidades do *NumPy*, é essencial que tenhamos a biblioteca instalada em nosso ambiente *Python*. A instalação do *NumPy* é simples e pode ser feita por meio do `pip`, o gerenciador de pacotes do *Python*. Abra o terminal ou *prompt* de comando do *Python* e digite o seguinte comando: `pip install numpy`.

Esse comando baixará e instalará a última versão do *NumPy*, juntamente com suas dependências, se necessário. Após a instalação, você pode verificar se o *NumPy* foi instalado corretamente, importando a biblioteca em seu interpretador Python ou em um *script*, conforme o Código 1.

Código 1 - Verificando a instalação do *NumPy*

```
import numpy as np

print("NumPy versão:", np.__version__)
```

Fonte: autoria própria.

NumPy é notável não apenas por sua velocidade e eficiência, mas também por sua interface intuitiva. No Código 2, há um exemplo de criação de uma variável do tipo *array* (a) e, logo em seguida, cada elemento dele é multiplicado por um escalar (2), resultando em outro *array*, atribuído à variável b.

Código 2 - Exemplos de utilização da biblioteca *NumPy* com *arrays* unidimensionais

```
# Criando um array de uma dimensão
a = np.array([1, 2, 3, 4, 5])
print("Array a:", a)
# Saída esperada: Array a: [1 2 3 4 5]


# Realizando operações matemáticas simples
b = a * 2
print("Array b (a multiplicado por 2):", b)
# Saída esperada: Array b (a multiplicado por 2): [ 2  4  6  8 10]
```

Fonte: autoria própria.

Até mesmo operações um pouco mais complexas, envolvendo *arrays* multidimensionais, são intuitivas. Mesmo novos(as) usuários(as) em *Python* podem aprender a usar *arrays* e a realizar operações complexas com uma curva de aprendizado relativamente suave. Um exemplo disso é o Código 3, no qual é definido um *array* bidimensional (2×2) e, em seguida, soma-se o valor 3 a cada um de seus elementos, calculando-se a média aritmética de todos os elementos (após a soma) ao final.

Código 3 - Exemplos de utilização da biblioteca *NumPy* com *arrays* multidimensionais e funções

```
# Criando um array bidimensional (matriz)
c = np.array([[1, 2], [3, 4]])
print("Matriz c:\n", c)
# Saída esperada:
# Matriz c:
# [[1 2]
#  [3 4]]

# Somando um número a cada elemento da matriz
d = c + 3
print("Matriz d (c mais 3):\n", d)
# Saída esperada:
# Matriz d (c mais 3):
# [[4 5]
#  [6 7]]

# Calculando a média dos elementos do array a
media = np.mean(a)
print("Média dos elementos de a:", media)
# Saída esperada: Média dos elementos de a: 3.0
```

Fonte: autoria própria.

1.2 Vetores e Matrizes

Como dito anteriormente, um dos principais componentes do *NumPy* é a capacidade de criar e manipular *arrays*. Um *array* é uma estrutura de dados que armazena valores de maneira eficiente, sendo ideal para operações matemáticas. A criação de *arrays* no *NumPy* pode ser feita a partir de listas ou utilizando funções específicas da biblioteca. Para criar um *array* unidimensional a partir de uma lista, basta usar a função `np.array()`. No exemplo do Código 4, tal função é utilizada para criar um *array* do *NumPy* a partir de uma lista contendo os valores de 1 a 5.

Código 4 - Exemplo de criação de um *array* a partir de uma lista

```
import numpy as np


a = np.array([1, 2, 3, 4, 5])
print("Array a:", a)
# Saída esperada: Array a: [1 2 3 4 5]
```

Fonte: autoria própria.

Além da criação de *arrays* a partir de listas, o *NumPy* oferece várias funções para gerar *arrays* de maneira prática e eficiente. Algumas dessas funções incluem `np.zeros()`, `np.ones()`, `np.arange()`, e `np.linspace()`. Essas funções permitem a criação de *arrays* preenchidos com zeros, uns ou com uma sequência de números em um intervalo especificado. No Código 5, é apresentada a função `np.zeros((3, 3))` para criar uma matriz 3x3 de zeros.

Código 5 - Criando *array* de zeros com *np.zeros*

```
zeros_array = np.zeros((3, 3))
print("Matriz de zeros:\n", zeros_array)
# Saída esperada:
# Matriz de zeros:
# [[0. 0. 0.]
#  [0. 0. 0.]
#  [0. 0. 0.]]
```

Fonte: autoria própria.

A função `np.arange(start, stop, step)` é útil para criar *arrays* com valores sequenciais com um passo definido. Por exemplo, no Código 6, é utilizado `np.arange(0, 10, 2)` para criar um *array* que começa em 0 e vai até 10, excluindo o último valor, com um incremento de 2.

Código 6 - Exemplo de utilização da função *np.arange*

```
range_array = np.arange(0, 10, 2)
print("Array com np.arange:", range_array)
# Saída esperada: Array com np.arange: [0 2 4 6 8]
```

Fonte: autoria própria.

Para criar *arrays* com valores espaçados uniformemente em um intervalo, pode ser utilizada a função `np.linspace(start, stop, num)`. Por exemplo, no Código 7, `np.linspace(0, 1, 5)` gera cinco números igualmente espaçados entre 0 e 1.

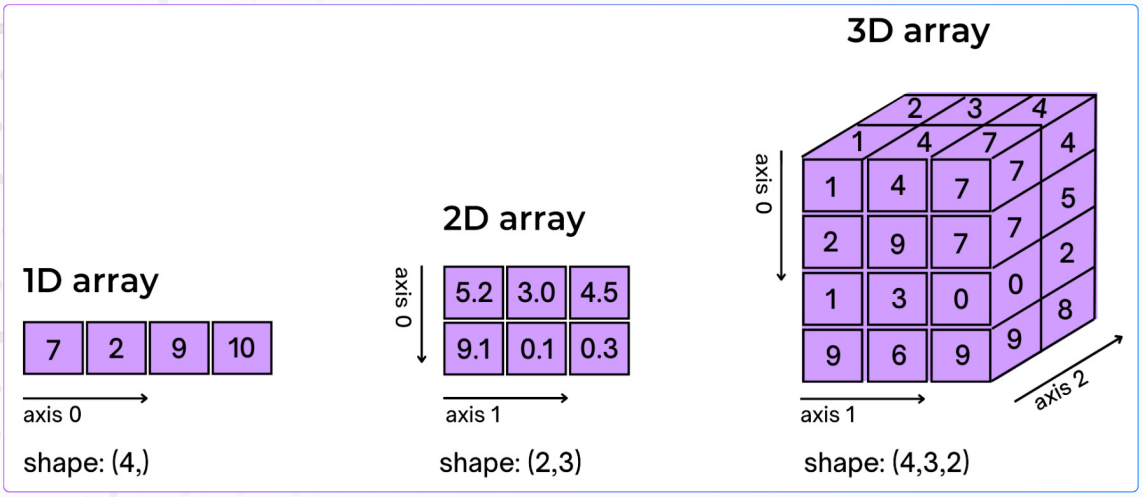
Código 7 - Exemplo de utilização da função np.linspace

```
linspace_array = np.linspace(0, 1, 5)
print("Array com np.linspace:", linspace_array)
# Saída esperada: Array com np.linspace: [0.   0.25 0.5  0.75 1.  ]
```

Fonte: autoria própria.

Arrays em *NumPy* possuem atributos que permitem fácil acesso às suas propriedades. Como pode ser visto no Código 8, `array.shape` retorna a forma (dimensões) do *array*, enquanto `array.ndim` retorna o número de dimensões. Esses atributos são essenciais para entender e manipular a estrutura dos dados armazenados nos *arrays*. Na Figura 1, constam exemplos de diferentes formas de *arrays* com uma, duas ou três dimensões. Nota-se que vetores são representados por *arrays* de uma dimensão e matrizes como *arrays* de duas dimensões. Os *arrays* com mais de duas dimensões são chamados de **tensores**.

Figura 1 - Formas e dimensões de diferentes *arrays*



Fonte: Kumavat (2024).

Código 8 - Atributos *shape* e *ndim* de um *array NumPy*

```
print("Forma do array:", linspace_array.shape)
# Saída esperada: Forma do array: (5,)
print("Número de dimensões:", linspace_array.ndim)
# Saída esperada: Número de dimensões: 1
```

Fonte: autoria própria.

Arrays multidimensionais podem ser criados passando listas aninhadas como argumentos para `np.array()`. A profundidade das listas aninhadas determina a dimensão do *array* resultante, o que é crucial para aplicações que requerem matrizes ou tensores de alta dimensão. No Código 9, veja um exemplo de criação de uma matriz 2x3 a partir de uma lista de listas.

Código 9 - Exemplo de criação de *array* multidimensional (matriz 2x3) a partir de listas

```
matriz = np.array([[1, 2, 3], [4, 5, 6]])
print("Matriz:\n", matriz)
# Saída esperada:
# Matriz:
# [[1 2 3]
#  [4 5 6]]
```

Fonte: autoria própria.

O atributo `array.dtype` é particularmente importante por informar o tipo de dados dos elementos armazenados no *array*. Conhecer o tipo de dados é vital para otimizar o desempenho e garantir que as operações matemáticas sejam realizadas corretamente. Um exemplo de utilização do atributo `array.dtype` pode ser visto no Código 10.

Código 10 - Exemplo de utilização do atributo *dtype* de um *array* do *NumPy*

```
print("Tipo de dados de 'a':", a.dtype)
# Saída esperada: Tipo de dados de 'a': int32 (ou int64, dependendo do sistema operacional)
```

Fonte: autoria própria.

Arrays NumPy são mutáveis, o que significa que os valores em um *array* podem ser alterados após sua criação. Isso permite a modificação de *arrays* existentes sem a necessidade de criar novos *arrays*, otimizando o uso de memória e processamento. No Código 11, é feita a alteração do valor na posição 0 do *array*.

Código 11 - Alterando o valor da posição 0 de um *array* previamente definido

```
a[0] = 10
print("Array 'a' modificado:", a)
# Saída esperada: Array 'a' modificado: [10  2  3  4  5]
```

Fonte: autoria própria.

Entender e manipular as formas dos *arrays* é fundamental para operações avançadas, incluindo *broadcasting* (a capacidade de realizar operações aritméticas em *arrays* de diferentes formas, ajustando automaticamente as dimensões) e *reshaping* (a alteração da forma de um *array* para diferentes formatos e estruturas), que serão abordadas em detalhes nas próximas seções. A manipulação da forma de um *array* permite a reorganização dos dados sem alterar seu conteúdo, facilitando operações como transposições e reestruturação dos dados.

1.3 Arrays do *NumPy vs.* Estruturas de Dados Nativas do *Python*

Listas, tuplas e conjuntos (*sets*) são estruturas de dados fundamentais em *Python*, cada uma com suas características e usos específicos. A título de revisão, no Código 12 consta um exemplo para cada uma dessas estruturas de dados, destacando suas características específicas.

Código 12 - Exemplos com listas, tuplas e *sets*

```
# Lista: Mutável, pode conter elementos de diferentes tipos
lista = [1, 'dois', 3.0]
lista.append(4)  # Adiciona um elemento ao final
print("Lista modificada:", lista)
# Saída esperada: Lista modificada: [1, 'dois', 3.0, 4]

# Tupla: Imutável, pode conter elementos de diferentes tipos
tupla = (1, 'dois', 3.0)
# tupla[0] = 2  # Isto resultaria em um erro
print("Tupla:", tupla)
# Saída esperada: Tupla: (1, 'dois', 3.0)

# Conjunto: Não ordenado, sem duplicatas
conjunto = {1, 2, 2, 3, 4, 5}
print("Conjunto (sem duplicatas):", conjunto)
# Saída esperada: Conjunto (sem duplicatas): {1, 2, 3, 4, 5}
```

Fonte: autoria própria.

Os *arrays* do *NumPy* oferecem diversas vantagens em relação às listas, tuplas e *sets* nativos do *Python*, especialmente quando se trata de computação numérica e científica. Enquanto listas e tuplas são estruturas de dados gerais que podem armazenar objetos de diferentes tipos, os *arrays* do *NumPy* são homogêneos. Isso significa que todos os

elementos devem ser do mesmo tipo. Essa homogeneidade permite que o *NumPy* utilize operações altamente otimizadas ao nível de *hardware*, acelerando a computação.

No Código 13, é apresentado um exemplo de cenário onde realizar uma operação em um *array* do *NumPy* é bem mais performático que iterar em uma lista. Nesse exemplo, são criados um *array* e uma lista com uma sequência de um milhão de números inteiros iniciando em 0 (por meio da função *range*). A seguir, é somado 1 em cada elemento da lista e do *array*. Os tempos são medidos. O tempo para executar tal operação em um *array* será notoriamente menor que aquele para uma lista.

Código 13 - Comparando o tempo de execução entre listas e *arrays* do *NumPy*

```
import time


lista = list(range(1000000))
inicio = time.time()
lista = [x + 1 for x in lista]
fim = time.time()
print("Tempo com lista:", fim - inicio)


array = np.arange(1000000)
inicio = time.time()
array += 1
fim = time.time()
print("Tempo com array NumPy:", fim - inicio)
# Saida Esperada (os resultados podem variar pois dependem das
configurações de velocidade de processador e memória):
# Tempo com lista: 0.23276233673095703
# Tempo com array NumPy: 0.002035856246948242
```

Fonte: autoria própria.

Em termos de funcionalidade, as listas são dinâmicas e podem ser facilmente modificadas (elementos podem ser adicionados ou removidos), enquanto os *arrays* do *NumPy* têm tamanho fixo após sua criação. Modificar o tamanho de um *array NumPy* implica na criação de um novo *array* e, consequentemente, na realocação de memória. Essa característica torna os *arrays* do *NumPy* menos flexíveis que as listas para operações que envolvem mudanças frequentes no tamanho dos dados, mas muito mais rápidos para operações matemáticas sobre elementos fixos.

Tuplas, sendo imutáveis, não oferecem a flexibilidade das listas ou dos *arrays* do *NumPy* para modificar conteúdo após a criação, o que as torna menos úteis para computação científica, na qual a manipulação dos dados é comum. No entanto, a imutabilidade das

tuplas as torna ideais como chaves de dicionários ou como elementos de conjuntos, o que não é possível com *arrays* do *NumPy* devido à sua mutabilidade.

Os conjuntos (*sets*) são estruturas únicas em *Python* que não permitem elementos duplicados e não mantêm uma ordem específica dos itens. Essa característica os torna inadequados para operações numéricas que dependem da ordenação dos elementos, mas úteis para testes de pertencimento e remoção de duplicatas. *Arrays* do *NumPy*, por outro lado, mantêm a ordem dos elementos e permitem duplicatas, características essenciais para muitas operações matemáticas e análises de dados.

A escolha entre usar *arrays* do *NumPy ou* estruturas de dados nativas do *Python* geralmente depende do contexto específico da aplicação. Para análises numéricas, simulações e grandes volumes de dados, os *arrays* do *NumPy* são geralmente preferidos devido à sua eficiência em termos de desempenho e à vasta biblioteca de funções matemáticas e estatísticas. Para aplicações que requerem estruturas de dados mutáveis com tipos variados ou que não envolvem operações numéricas intensivas, listas e tuplas podem ser mais apropriadas.

1.4 Operações com Arrays

Os *arrays* do *NumPy* são projetados para facilitar operações numéricas complexas com sintaxe simples e eficiente. Uma das características mais notáveis do *NumPy* é que todas as operações aritméticas básicas são aplicadas elemento a elemento. Isso significa que se você adicionar, subtrair, multiplicar ou dividir dois *arrays*, cada operação será realizada entre os elementos correspondentes de ambos os *arrays*.

Por exemplo, ao trabalhar com dois *arrays*, a e b, que contêm os valores [1, 2, 3] e [4, 5, 6], respectivamente, a soma de a e b resultará em um novo *array* onde cada elemento é a soma dos elementos correspondentes dos *arrays* originais. No Código 14, isso é ilustrado, juntamente com outras operações aritméticas básicas.

Código 14 - Exemplo de operações aritméticas com *arrays*

```
import numpy as np


a = np.array([1, 2, 3])
b = np.array([4, 5, 6])


# Adição
c = a + b
print("Adição:", c)
# Saída esperada: Adição: [5 7 9]
```

continua

```
# Subtração
d = b - a
print("Subtração:", d)
# Saída esperada: Subtração: [3 3 3]

# Multiplicação
e = a * b
print("Multiplicação:", e)
# Saída esperada: Multiplicação: [ 4 10 18]

# Divisão
f = b / a
print("Divisão:", f)
# Saída esperada: Divisão: [4. 2.5 2. ]
```

Fonte: autoria própria.

Além das operações básicas, o *NumPy* suporta o conceito de *broadcasting*, extremamente útil quando se deseja realizar operações entre *arrays* de diferentes tamanhos. O *broadcasting* permite que você realize operações aritméticas em *arrays* que, à primeira vista, não pareceriam compatíveis para tal. Por exemplo, se você tem um *array* e um escalar, o *NumPy* permite adicionar o escalar a cada elemento do *array* sem a necessidade de replicar o escalar para corresponder ao tamanho do *array*. Esse exemplo é apresentado no Código 15.

O *broadcasting* também se estende a *arrays* de diferentes dimensões. Considere um *array* h de tamanho 3 e um *array* i de forma 3x1. O *NumPy* ajusta automaticamente cada *array* para que suas formas correspondam, permitindo operações elemento a elemento entre eles. Nesse caso, h é um vetor linha (considerado com a forma (1, 3)) e i é um vetor coluna (considerado com a forma (3, 1)). O *NumPy* "estica" h ao longo da dimensão vertical e i ao longo da dimensão horizontal, resultando em um *array* final de forma (3, 3), onde cada elemento de h é somado ao correspondente de i, produzindo a matriz resultante (Código 15).

Código 15 - Exemplo de *broadcasting* entre *arrays*

```
h = np.array([1, 2, 3])
i = np.array([[0], [1], [2]])

# Broadcasting entre diferentes tamanhos
```

continua

```
j = h + i
print("Broadcasting entre diferentes tamanhos:\n", j)
# Saída esperada:
# Broadcasting entre diferentes tamanhos:
# [[1 2 3]
#  [2 3 4]
#  [3 4 5]]
```

Fonte: autoria própria.

A indexação e o *slicing* são técnicas essenciais para acessar e modificar partes de um *array*. Com a indexação, você pode extrair elementos específicos de um *array* usando índices. Por exemplo, a[1] retorna o segundo elemento do *array* a. O *slicing*, por outro lado, permite acessar subconjuntos do *array*. Usando a sintaxe `start:stop:step`, você pode extrair partes do *array* conforme a necessidade. Por exemplo, a[0 : 2] retorna os dois primeiros elementos do *array* em a, conforme apresentado no Código 16.

Código 16 - Exemplos de indexação e *slicing*

```
# Indexação simples
k = a[1]
print("Elemento no índice 1 –", k)
# Saída esperada: Elemento no índice 1 - 2

# Slicing
l = a[0 –2]
print("Primeiros dois elementos:", l)
# Saída esperada: Primeiros dois elementos: [1 2]
```

Fonte: autoria própria.

O *slicing* é uma técnica poderosa que vai além do acesso básico a elementos, permitindo manipulações mais complexas de *arrays*. Além de selecionar um segmento de um *array*, o *slicing* pode ser usado para modificar partes de um *array*. Ao atribuir um valor a uma fatia de um *array*, esse valor é propagado ao longo de toda a seção selecionada. Por exemplo, no Código 17, parte de um *array* é zerada de forma simples utilizando *slicing*.

Código 17 - Exemplo de *zeroing* com *slicing*

```
m = np.array([1, 2, 3, 4, 5])
m[1 -4] = 0
print(“Array após zeroing:”, m)
# Saída esperada: Array após zeroing: [1 0 0 0 5]
```

Fonte: autoria própria.

Uma característica importante do *slicing* é que as alterações feitas em *subarrays* por meio do *slicing* são refletidas no *array* original. Isso ocorre porque o *slicing* em *NumPy* retorna uma visão do *array* original, não uma cópia. Essa característica torna o *slicing* uma ferramenta eficiente, mas os(as) usuários(as) devem ser cautelosos(as) para não modificar dados inadvertidamente.

Outro aspecto importante do *slicing* é a inclusão do *step* que permite especificar o intervalo entre os elementos no segmento selecionado do *array*. Isso adiciona uma camada extra de flexibilidade ao acessar múltiplos elementos. Por exemplo, se você deseja selecionar elementos alternados de um *array*, pode definir o *step* como 2, o que significa que o *slicing* “pulará” cada segundo elemento. O uso do *step* é particularmente útil em situações nas quais você precisa de uma amostragem regular de dados em um conjunto maior ou deseja inverter a ordem dos elementos de um *array*.

No Código 18, é demonstrado o uso de *slicing* com *step* em *arrays NumPy*. No primeiro exemplo, m[1 -8 -2] seleciona elementos do índice 1 ao 7 (exclusivo), pegando a cada 2 elementos, resultando nos valores [1, 3, 5, 7]. No segundo exemplo, m[::-1] usa um *step* negativo para inverter o *array*, resultando em [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]. Esses exemplos, bem como os apresentados na Figura 2, mostram como o *slicing* pode ser utilizado para extrair *sub-arrays* e manipular a ordem dos elementos.

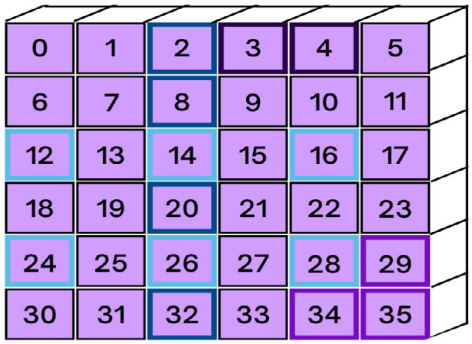
Código 18 - *Slicing* com *step*

```
# Slicing com step
m = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
step_slice = m[1 : 8 ? 2]
print(“Elementos selecionados com step de 2 –”, step_slice)
# Saída esperada: Elementos selecionados com step de 2 - [1 3 5 7]

# Usando step negativo para inverter o array
reverse_slice = m[::-1]
print(“Array invertido:”, reverse_slice)
# Saída esperada: Array invertido: [9 8 7 6 5 4 3 2 1 0]
```

Fonte: autoria própria.

Figura 2 - Exemplos de uso de *slicing*

```
>>> a[0, 3:5]
array ( [3, 4] )

>>> a[4:, 4:]
array ( [28, 29],
        [ 34, 35]] )

>>> a [:, 2]
array ( [2, 8, 14, 20, 26, 32] )

>>> a [2 : : 2, : : 2]
array ( [ 12, 14, 16],
        [24, 26, 28]])
```



Fonte: Stack Overflow.

Além de operações básicas e *slicing*, os *arrays NumPy* suportam indexação avançada, que permite selecionar elementos não contíguos e em padrões complexos. Um exemplo, como visto no Código 19, é a indexação com *arrays* de inteiros, na qual é possível especificar os índices dos elementos que deseja acessar.

Código 19 - Exemplo de indexação com lista de índices

```
n = np.array([10, 20, 30, 40, 50])

indices = [1, 3, 4]

o = n[indices]

print("Elementos selecionados:", o)

# Saída esperada: Elementos selecionados: [20 40 50]
```

Fonte: autoria própria.

A indexação booleana é outra forma poderosa de manipular *arrays*. Essa técnica usa um *array* booleano do mesmo tamanho que o *array* a ser indexado para selecionar elementos. Por exemplo, no Código 20, consta uma seleção com todos os elementos de um *array* que são maiores que um valor específico (p > 3, no exemplo).

Código 20 - Exemplo de indexação booleana

```
p = np.array([1, 2, 3, 4, 5])

mask = p > 3

q = p[mask]

print("Elementos maiores que 3 -", q)

# Saída esperada: Elementos maiores que 3 - [4 5]
```

NumPy também oferece a função `np.where()`, uma ferramenta versátil para realizar seleções condicionais. Essa função é particularmente útil para substituir valores em um *array* com base em alguma condição. No exemplo do Código 21, todos os valores menores que 3 foram substituídos por zero de forma fácil com `np.where()`. Como é possível observar, a função `np.where()` recebe três parâmetros: o primeiro é a condição para realização da filtragem, o segundo trata-se do valor a ser utilizado para substituição (caso a condição seja verdadeira para o elemento correspondente). Por fim, o terceiro parâmetro é o vetor de entrada, no qual a operação será realizada.

Código 21 - Exemplo usando *np.where*

```
r = np.array([1, 2, 3, 4, 5])
s = np.where(r < 3, 0, r)
print("Substituição condicional:", s)
# Saída esperada: Substituição condicional: [0 0 3 4 5]
```

Fonte: autoria própria.

O *broadcasting* e as operações aritméticas que discutimos são apenas a ponta do iceberg. *NumPy* suporta uma ampla gama de operações matemáticas e científicas mais complexas, como operações de álgebra linear, estatística e até transformações geométricas. Além disso, as operações com *arrays* não se limitam a *arrays* de uma ou duas dimensões. *NumPy* é capaz de manipular *arrays* de alta dimensão (tensores), cruciais em aplicações avançadas como aprendizado de máquina e processamento de imagens.

1.5 Funções Universais (*ufunc*)

As Funções Universais, ou `ufuncs`, são uma característica central do *NumPy* que permite a execução de operações vetorizadas (operações aplicadas a todos os elementos de um *array* simultaneamente). Essas funções operam elemento a elemento, proporcionando uma execução altamente otimizada e rápida, fundamental para o processamento de grandes volumes de dados. As *ufuncs* são implementadas em C, o que significa que elas podem operar com a velocidade de código compilado, evitando os laços de interação típicos do *Python* puro.

Existem dois tipos principais de `ufuncs` no *NumPy*: as `ufuncs` unárias, que operam em uma única entrada, e as `ufuncs` binárias, que operam em dois *inputs*. Exemplos de `ufuncs` unárias incluem funções como np.sqrt e np.exp, que calculam a raiz quadrada e o exponencial de cada elemento de um *array*, respectivamente. As `ufuncs` binárias incluem operações como np.add e `np.multiply`, que realizam adição e multiplicação elemento a elemento entre dois *arrays*.

Além de operações básicas, as `ufuncs` também suportam operações mais complexas essenciais em campos científicos e de engenharia. Por exemplo, operações de álgebra linear como produto de matrizes (`np.dot`), determinante (`np.linalg.det`) e autovalores (`np.`

`linalg.eigvals`) são suportadas por meio de submódulos específicos como `np.linalg`, como apresentado no Código 22. Essas funções permitem realizar cálculos matemáticos complexos que são a base para muitos algoritmos de ciência de dados e aprendizagem de máquina.

Código 22 - Exemplo de operações de álgebra linear

```
A = np.array([[1, 2], [3, 4]])
B = np.array([[2, 0], [1, 2]])

# Produto de matrizes
produto = np.dot(A, B)
print("Produto das matrizes:\n", produto)
# Saída esperada:
# Produto das matrizes:
# [[ 4  4]
#  [10  6]]

# Determinante
determinante = np.linalg.det(A)
print("Determinante:", determinante)
# Saída esperada: Determinante: -2.0

# Autovalores
autovalores = np.linalg.eigvals(A)
print("Autovalores:", autovalores)
# Saída esperada: Autovalores: [-0.37228132  5.37228132]
```

Fonte: autoria própria.

Em estatística, o *NumPy* oferece `ufuncs` que podem calcular médias, medianas, desvios padrão e outras medidas estatísticas diretamente de *arrays* numéricos. Isso é extremamente útil para análise de dados, onde tais cálculos são frequentes. Funções como `np.mean`, `np.median`, e `np.std`, apresentadas no Código 23, são exemplos de como as operações estatísticas podem ser realizadas de maneira eficiente e direta.

Código 23 - Exemplo de operações estatísticas

```
dados = np.array([1, 2, 3, 4, 5])

# Média
media = np.mean(dados)
print("Média:", media)
# Saída esperada: Média: 3.0

# Mediana
mediana = np.median(dados)
print("Mediana:", mediana)
# Saída esperada: Mediana: 3.0

# Desvio padrão
desvio_padrao = np.std(dados)
print("Desvio padrão:", desvio_padrao)
# Saída esperada: Desvio padrão: 1.4142135623730951
```

Fonte: autoria própria.

As transformações geométricas também são uma aplicação importante das `ufuncs`. O *NumPy* pode ser usado para aplicar transformações como rotações, escalonamentos e translações em objetos geométricos representados em *arrays*. Isso é particularmente valioso em áreas como processamento de imagens e gráficos computacionais, onde tais transformações são fundamentais.

Como foi possível observar, as funções universais do *NumPy* são ferramentas poderosas que proporcionam não apenas a velocidade, mas também a flexibilidade para executar uma ampla gama de operações matemáticas e científicas. A capacidade de executar essas operações de forma vetorizada, sem a necessidade de *loops* explícitos em *Python*, não só economiza tempo de desenvolvimento, mas também melhora significativamente o desempenho dos programas.

1.6 Visualização de Dados com NumPy e Matplotlib

Embora o *NumPy* seja uma ferramenta poderosa para a manipulação de dados numéricos, a visualização desses dados é igualmente importante para a análise e comunicação eficaz dos resultados. Para isso, combinamos frequentemente o *NumPy* com *Matplotlib*, uma biblioteca de plotagem em *Python* que oferece uma variedade de

funções para a criação de gráficos estáticos, animados e interativos. Nesta seção, será explorado o uso do *NumPy* em conjunto com *Matplotlib* para visualizar dados simples, preparando o terreno para uma exploração mais profunda de *Matplotlib* na Unidade III.

Um exemplo básico de visualização de dados envolve a criação de um gráfico de linhas. Exemplos comuns são a visualização da evolução de uma variável ao longo do tempo ou a relação funcional entre duas variáveis. Utilizando *arrays* do NumPy para representar os dados, pode-se facilmente plotar esses dados com *Matplotlib*.

Código 24 - Visualização de dados com *NumPy* e *Matplotlib*

```
import numpy as np
import matplotlib.pyplot as plt
# Dados
x = np.linspace(0, 10, 100)
y = np.sin(x)



# Plotando
plt.figure(figsize=(8, 4))
plt.plot(x, y, label='sin(x)')
plt.title('Gráfico de Linha Simples')
plt.xlabel('x')
plt.ylabel('sin(x)')
plt.legend()
plt.grid(True)
plt.show()
```

Fonte: autoria própria.

O Código 24 gera um gráfico de linha que mostra a função seno entre 0 e 10. O uso de `np.linspace` para criar o *array* x é um exemplo perfeito de como o *NumPy* gera dados para visualizações. A função `plt.plot()` de *Matplotlib* é usada aqui para criar o gráfico de linha, com *labels* (rótulos) para o eixo x e y, um título e uma legenda, essenciais para entender o gráfico, como pode ser visto na Figura 3.

Figura 3 - Gráfico gerado a partir do Código 24

Fonte: autoria própria.

Outro exemplo útil é a visualização de distribuições de dados. Histogramas são uma ferramenta comum para explorar a distribuição de conjuntos de dados. *NumPy* pode ser usado para calcular os histogramas e *Matplotlib* para plotá-los.

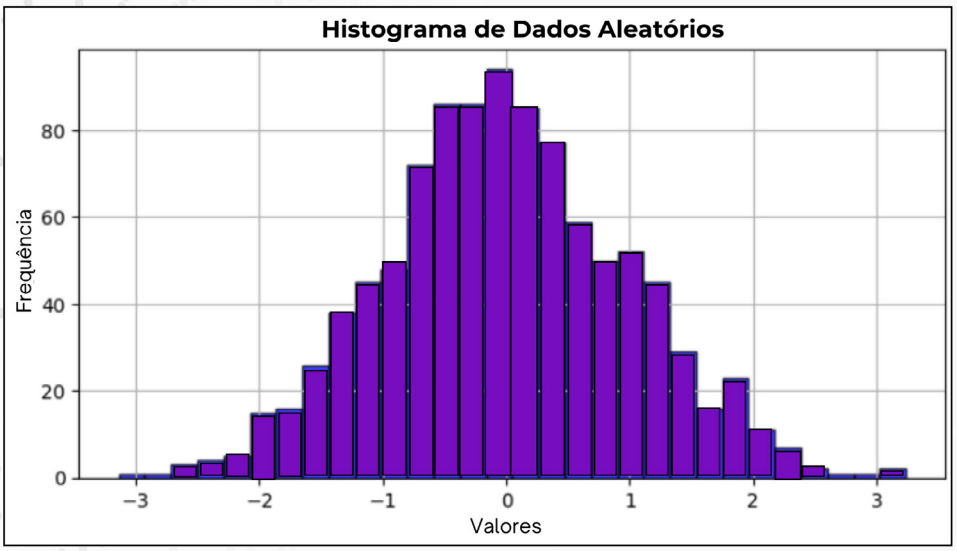
Código 25 - Exemplo de histograma com *NumPy* e *Matplotlib*

```
# Dados aleatórios
data = np.random.normal(size=1000)


# Histograma
plt.figure(figsize=(8, 4))
plt.hist(data, bins=30, alpha=0.75, color='blue', edgecolor='black')
plt.title('Histograma de Dados Aleatórios')
plt.xlabel('Valores')
plt.ylabel('Frequência')
plt.grid(True)
plt.show()
```

Fonte: autoria própria.

O Código 25 produz um histograma (`plt.hist`) que ajuda a visualizar como os dados (gerados aleatoriamente, utilizando a função `np.random.normal`) estão distribuídos em torno da média. Esse tipo de gráfico, como apresentado na Figura 4, é extremamente útil para a análise estatística inicial e para verificar a normalidade dos dados experimentais.

Figura 4 - Histograma gerado a partir do Código 25



Fonte: autoria própria.

Embora esta seção seja apenas uma introdução à combinação de *NumPy* com *Matplotlib* para visualização de dados, ela destaca a facilidade com que essas duas poderosas ferramentas podem ser usadas juntas. Aprofundaremos mais no uso de *Matplotlib* na Unidade III, na qual uma variedade maior de tipos de gráficos e técnicas de visualização avançadas serão exploradas. Essa introdução possibilita estabelecer a base sobre como o *NumPy* alimenta as visualizações com representações eficientes de dados e como o *Matplotlib* pode ser usado para transformar esses dados em percepções visuais compreensíveis.

1.7 *Notebook Colab*

Introdução ao NumPy

NumPy é uma biblioteca essencial para a computação numérica em *Python*. Destaca-se sua eficiência em operações matemáticas graças à vetorização que elimina a necessidade de *loops* explícitos (estruturas de repetição explícitas). Desenvolvida inicialmente em 2005 por Travis Oliphant, a biblioteca é crucial não apenas para a ciência de dados, mas também em campos como engenharia e física devido à sua capacidade de realizar operações complexas de forma eficiente. *NumPy* é a base para outras ferramentas de ciência de dados e se integra bem com linguagens de baixo nível como C e Fortran, o que melhora significativamente o desempenho das operações computacionais.

Sempre que precisar, consulte a documentação oficial do *NumPy*: https://numpy.org/doc/stable/reference/index.html

O *NumPy* já vem instalado nesse *Google Colab*. Mas, para instalá-lo em seu ambiente local, você pode usar o comando a seguir:

```
!pip install numpy
```

```
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages
(1.26.4)
```

Verificando Instalação do *NumPy*:

```
import numpy as np

print("Versão do NumPy:", np.__version__)
```

```
Versão do NumPy: 1.26.4
```

A seguir, são apresentados exemplos de utilização da biblioteca *NumPy* com *arrays* unidimensionais.

```
# Criando um array de uma dimensão

a = np.array([1, 2, 3, 4, 5])

print("Array a:", a)
```

```
Array a: [1 2 3 4 5]
```

```
# Realizando operações matemáticas simples

b = a * 2

print("Array b (a multiplicado por 2):", b)
```

```
Array b (a multiplicado por 2): [ 2  4  6  8 10]
```

Também pode-se trabalhar com *arrays* multidimensionais com *NumPy*. Veja alguns exemplos a seguir:

```
# Criando um array bidimensional (matriz)

c = np.array([[1, 2], [3, 4]])

print("Matriz c:\n", c)
```

```
Matriz c:
 [[1 2]
 [3 4]]
```

```
# Somando um número a cada elemento da matriz
d = c + 3
print("Matriz d (c mais 3):\n", d)
```

```
Matriz d (c mais 3):
 [[4 5]
 [6 7]]
```

```
# Calculando a média dos elementos do array a
media = np.mean(a)
print("Média dos elementos de a:", media)
```

```
Média dos elementos de a: 3.0
```

Arrays e Matrizes

Nesta seção, é explorada a importância central dos *arrays* no *NumPy*, uma estrutura de dados que armazena valores de forma eficiente e que é essencial para operações matemáticas. É detalhado como criar *arrays* utilizando várias funções do *NumPy*, como np.array() para conversões diretas de listas e np.zeros(), np.ones(), np.arange(), e np.linspace() para geração rápida de *arrays* com características específicas. Também abordamos atributos críticos de *arrays* como *shape* e *ndim*, que revelam as dimensões e a quantidade de dimensões do *array*, respectivamente, permitindo aos/às usuários/as manipularem e entenderem mais profundamente suas estruturas de dados.

Podem ser utilizadas algumas funções interessantes para criação de *arrays*, como zeros, *arange* e *linspace*.

```
zeros_array = np.zeros((3, 3))
print("Matriz de zeros:\n", zeros_array)
```

```
Matriz de zeros:
 [[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
```

```
range_array = np.arange(0, 10, 2)
print("Array com np.arange:", range_array)
```

```
Array com np.arange: [0 2 4 6 8]
```

```
linspace_array = np.linspace(0, 1, 5)
print("Array com np.linspace:", linspace_array)
```

```
Array com np.linspace: [0.   0.25 0.5  0.75 1.  ]
```

Para saber a forma e dimensões do *array*, pode-se usar os atributos *shape* e *ndim*:

```
print("Forma do array:", linspace_array.shape)
print("Número de dimensões:", linspace_array.ndim)
```

```
Forma do array: (5,)
Número de dimensões: 1
```

É possivel definir *arrays* com listas de listas:

```
matriz = np.array([[1, 2, 3], [4, 5, 6]])
print("Matriz:\n", matriz)
```

```
Matriz:
 [[1 2 3]
 [4 5 6]]
```

Os elementos do *array* em *NumPy* devem ter o mesmo tipo de dados. Para saber qual o tipo de dados de um *array*, utiliza-se o atributo *dtype*:

```
print("Tipo de dados de 'a':", a.dtype)
```

```
Tipo de dados de 'a': int64
```

Para acessar uma posição do *array*, utiliza-se colchetes ([]):

```
a[0] = 10
print("Array 'a' modificado:", a)

matriz[0][1] = 10
print("Matriz 'matriz' modificada:", matriz)
```

```
Array 'a' modificado: [10  2  3  4  5]
Matriz 'matriz' modificada: [[ 1 10  3]
 [ 4  5  6]]
```

Arrays do *NumPy* vs. Estruturas de Dados Nativas do *Python*

Os *arrays* do *NumPy* podem ser comparados com as estruturas de dados nativas do *Python*, como listas, tuplas e conjuntos. Ao contrário dessas estruturas, os *arrays* do *NumPy* são homogêneos e permitem operações matemáticas eficientes ao nível do *hardware*, o que os torna ideais para computação numérica e científica. As listas são flexiveis para operações que requerem alterações frequentes de tamanho, as tuplas são imutáveis, o que as torna úteis como chaves de dicionários e a unicidade dos conjuntos elimina duplicatas. Contrastando, os *arrays* do *NumPy*, com sua capacidade de realizar operações complexas de forma vetorizada e manter a ordem dos elementos, apresentam-se como a escolha preferida para análises numéricas e manipulações de grandes volumes de dados.

Lista: mutável, pode conter elementos de diferentes tipos:

```
lista = [1, 'dois', 3.0]
lista.append(4)  # Adiciona um elemento ao final
print("Lista modificada:", lista)
```

```
Lista modificada: [1, 'dois', 3.0, 4]
```

Tupla: imutável, pode conter elementos de diferentes tipos:

```
tupla = (1, 'dois', 3.0)
tupla[0] = 2  # Isto resultaria em um erro, uma vez que as tuplas são imutáveis.
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-3-795970560b0c> in <cell line: 2>()
      1 tupla = (1, 'dois', 3.0)
----> 2 tupla[0] = 2  # Isto resultaria em um erro

TypeError: 'tuple' object does not support item assignment
```

```
print("Tupla:", tupla)
```

```
Tupla: (1, 'dois', 3.0)
```

Conjunto: não ordenado, sem duplicatas:

```
conjunto = {1, 2, 2, 3, 4, 5}
print("Conjunto (sem duplicatas):", conjunto)
```

```
Conjunto (sem duplicatas): {1, 2, 3, 4, 5}
```

A execução com *NumPy* geralmente será significativamente mais rápida:

```
import time


lista = list(range(1000000))
inicio = time.time()
lista = [x + 1 for x in lista]
fim = time.time()
print("Tempo com lista:", fim - inicio)


array = np.arange(1000000)
inicio = time.time()
array += 1
fim = time.time()
print("Tempo com array NumPy:", fim - inicio)
```

```
Tempo com lista: 0.08141207695007324
Tempo com array NumPy: 0.0006694793701171875
```

Exercício Prático

Crie um *array* com o formato e valores a seguir, utilizando as funções aprendidas nesta seção.

Array:

```
 [[ 0  2  4  6  8]
 [10 12 14 16 18]]
```

```
array = # SEU CÓDIGO AQUI
print("Array:\n", array)
```

```
Array:
 [[ 0  2  4  6  8]
 [10 12 14 16 18]]
```

Operações com *Arrays*

Nessa seção, discuti-se como os *arrays* do *NumPy* facilitam a realização de operações numéricas complexas por meio de uma interface intuitiva e eficiente. As operações básicas como adição, subtração, multiplicação e divisão são aplicadas elemento a elemento entre *arrays*. Introduzimos conceitos avançados como *broadcasting*, que permitem operações entre *arrays* de diferentes tamanhos, sem replicação manual de dados, e exploramos técnicas de indexação e *slicing*, que são fundamentais para acessar e manipular partes específicas de *arrays*. Também abordamos operações mais sofisticadas, incluindo funções universais (ufuncs) que operam em um nível de vetorização, oferecendo desempenho substancialmente melhorado em comparação com *loops* tradicionais em *Python*.

```
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])


# Adição
c = a + b
print("Adição:", c)

# Subtração
d = b - a
print("Subtração:", d)

# Multiplicação
e = a * b



print("Multiplicação:", e)

# Divisão
f = b / a
print("Divisão:", f)
```

Adição: [5 7 9]
Subtração: [3 3 3]
Multiplicação: [4 10 18]
Divisão: [4. 2.5 2.]

```
h = np.array([1, 2, 3])
i = np.array([[0], [1], [2]])
# Broadcasting entre diferentes tamanhos
j = h + i
print("Broadcasting entre diferentes tamanhos:\n", j)
```

```
Broadcasting entre diferentes tamanhos:
 [[1 2 3]
 [2 3 4]
 [3 4 5]]
```

```
# Indexação simples
k = a[1]
print("Elemento no índice 1:", k)
# Slicing
l = a[0:2]
print("Primeiros dois elementos:", l)
```

```
Elemento no índice 1: 2
Primeiros dois elementos: [1 2]
```

```
m = np.array([1, 2, 3, 4, 5])
m[1:4] = 0
print("Array após zeroing:", m)
```

```
Array após zeroing: [1 0 0 0 5]
```

```
# Slicing com step
m = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
step_slice = m[1:8:2]
print("Elementos selecionados com step de 2:", step_slice)

# Usando step negativo para inverter o array
reverse_slice = m[::-1]
print("Array invertido:", reverse_slice)
```

```
Elementos selecionados com step de 2: [1 3 5 7]
Array invertido: [9 8 7 6 5 4 3 2 1 0]
```

```
n = np.array([10, 20, 30, 40, 50])
indices = [1, 3, 4]
o = n[indices]
print("Elementos selecionados:", o)
```

Elementos selecionados: [20 40 50]

```
p = np.array([1, 2, 3, 4, 5])
mask = p > 3
q = p[mask]
print("Elementos maiores que 3:", q)

# Também é possível usar a mascara diretamente entre os colchetes:
print("Elementos maiores que 3 (usando mascara diretamente):", p[p > 3])
```

Elementos maiores que 3: [4 5]
Elementos maiores que 3 (usando mascara diretamente): [4 5]

Exercício Prático

Substitua os valores pares por zero e retorne apenas os valores a partir da posição 50.

```
array = np.arange(0, 100, 1)
# SEU CÓDIGO AQUI
print("Array modificado:", array)
```

Array modificado: [0 51 0 53 0 55 0 57 0 59 0 61 0 63 0 65 0 67 0 69 0 71 0 73
 0 75 0 77 0 79 0 81 0 83 0 85 0 87 0 89 0 91 0 93 0 95 0 97
 0 99]

Funções Universais (*ufunc*)

Esta seção destaca o papel central das ufuncs no *NumPy*: funções otimizadas para operar sobre *arrays* elemento a elemento, proporcionando execução rápida e eficiente. As ufuncs podem ser unárias, operando sobre um único array, ou binárias, envolvendo dois *arrays*, e incluem operações como adição, multiplicação, raiz quadrada e exponencial. Essas funções são implementadas em C para desempenho máximo e são cruciais para a realização de cálculos matemáticos e científicos complexos, como operações de álgebra linear e estatísticas. A *ufunc* np.where, por exemplo, é versátil para substituições condicionais dentro de *arrays*, demonstrando a capacidade do *NumPy* de facilitar manipulações complexas de dados de forma eficaz e direta.

```
r = np.array([1, 2, 3, 4, 5])
s = np.where(r < 3, 0, r)
print("Substituição condicional:", s)
```

Substituição condicional: [0 0 3 4 5]

```
A = np.array([[1, 2], [3, 4]])
B = np.array([[2, 0], [1, 2]])


# Produto de matrizes
produto = np.dot(A, B)
print("Produto das matrizes:\n", produto)
```

Produto das matrizes:
[[4 4]
[10 8]]

```
# Determinante
determinante = np.linalg.det(A)
print("Determinante:", determinante)
```

Determinante: -2.0000000000000004

```
# Autovalores
autovalores = np.linalg.eigvals(A)
print("Autovalores:", autovalores)
```

Autovalores: [-0.37228132 5.37228132]

Operações Estatísticas:

```
dados = np.array([1, 2, 3, 4, 5])

# Média
media = np.mean(dados)
print("Média:", media)
```

Média: 3.0

```
# Mediana
mediana = np.median(dados)
print("Mediana:", mediana)
```

Mediana: 3.0

```
# Desvio padrão
desvio_padrao = np.std(dados)
print("Desvio padrão:", desvio_padrao)
```

Desvio padrão: 1.4142135623730951

Exercício Prático

São dadas notas de 20 alunos para dois bimestres (`notasB1` e `notasB2`). Calcule a média das notas de cada aluno em um novo *array*. Supondo que a média para aprovação seja 6.0, gere um *array* de valores booleanos com True, caso aprovado, e False, caso reprovado. Utilize a função `np.where`.

```
notasB1 = np.random.uniform(1, 10, 20)
notasB2 = np.random.uniform(1, 10, 20)

# Calcule a média das notas
media_notas = # SEU CÓDIGO AQUI
print("Média das notas:", media_notas)

aprovados = # SEU CÓDIGO AQUI
print("Alunos aprovados:", aprovados)
```

Visualização de Dados com *NumPy* e *Matplotlib*

Nesta seção, exploramos como combinar o *NumPy* com a biblioteca de plotagem *Matplotlib* para criar visualizações de dados eficazes e informativas. Demonstramos o uso de *NumPy* para gerar dados numéricos que são visualizados utilizando as funcionalidades de plotagem do *Matplotlib*. Um exemplo prático envolve a criação de um gráfico de linha para visualizar a função seno usando *arrays* gerados pelo *NumPy* e plotados com *Matplotlib*, destacando a simplicidade e eficácia dessa abordagem. Também introduzimos técnicas para a visualização de distribuições de dados, como histogramas, que são essenciais para

análises estatísticas iniciais e para verificar a normalidade dos dados. Esta seção serve como uma introdução essencial à interação entre *NumPy* e *Matplotlib*, preparando o terreno para explorações mais profundas na visualização de dados avançada, apresentada na **Unidade III**.

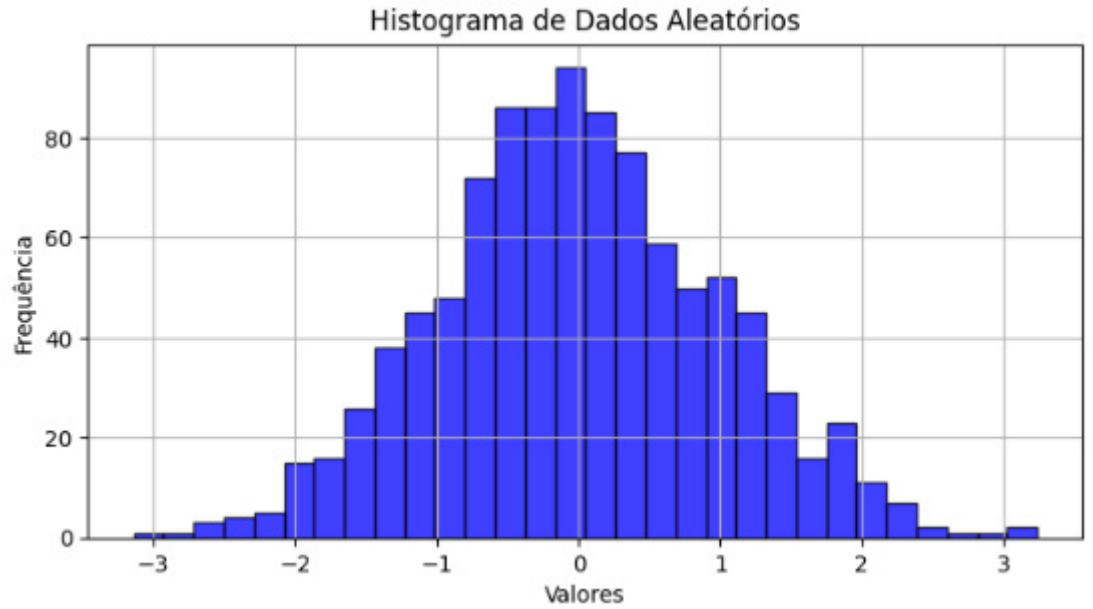
```
import numpy as np
import matplotlib.pyplot as plt


# Dados
x = np.linspace(0, 10, 100)
y = np.sin(x)


# Plotando
plt.figure(figsize=(8, 4))
plt.plot(x, y, label='sin(x)')
plt.title('Gráfico de Linha Simples')
plt.xlabel('x')
plt.ylabel('sin(x)')
plt.legend()
plt.grid(True)
plt.show()
```

Figura 5 - Gráfico de Linha Simples

Fonte: autoria própria

```
# Dados aleatórios
data = np.random.normal(size=1000)


# Histograma
plt.figure(figsize=(8, 4))
plt.hist(data, bins=30, alpha=0.75, color='blue', edgecolor='black')
plt.title('Histograma de Dados Aleatórios')
plt.xlabel('Valores')
plt.ylabel('Frequência')
plt.grid(True)
plt.show()
```

Figura 6 - Histograma de Dados Aleatórios



Fonte: autoria própria.

1.8 Saiba Mais...

NumPy Team. *NumPy documentation.*

Python para Processamento de Dados

Unidade II

Manipulação e Análise de Dados Bidimensionais com Pandas

Unidade II - Manipulação e Análise de Dados Bidimensionais com *Pandas*

2.1 Introdução ao Pandas e DataFrames

Pandas é uma biblioteca de *software* essencial para manipulação e análise de dados em *Python*, desenvolvida por Wes McKinney, em 2008. A biblioteca foi criada para resolver problemas de análise de dados em *Python* de forma rápida e fácil. Ela oferece estruturas de dados flexíveis e expressivas, projetadas para trabalhar com dados relacionais, ou rotulados, de maneira intuitiva.

A capacidade de *Pandas* para lidar com estruturas de dados grandes e complexas torna-o uma ferramenta indispensável em diversos campos da ciência de dados, economia, biologia, entre outros. As funcionalidades de *Pandas* estendem-se desde simples leituras de dados até operações de agrupamento e estatísticas avançadas.

Pandas introduz dois tipos de estruturas de dados fundamentais: ***Series*** e ***DataFrame***. Essas estruturas são construídas sobre *arrays* do *NumPy*, o que significa que elas são rápidas e eficientes para operações numéricas, mesmo com grandes volumes de dados.

Uma ***Series*** (traduzido para **série** para facilitar a leitura) é uma estrutura de dados unidimensional que pode armazenar qualquer tipo de dado (**inteiros**, ***strings***, ***floats***, objetos *Python*, etc.). Cada elemento em uma série possui um índice associado, o que torna essa estrutura semelhante a um *array* do *NumPy*, mas com capacidades de indexação mais flexíveis. No Código 26, é apresentado um exemplo de inicialização de uma série com 6 elementos (a partir de uma lista). Nota-se o valor np.nan, útil para representar um dado faltante ou que estava fora do tipo correto para a série.

Código 26 - Exemplo de criação de uma *Series* do *Pandas*

```
import pandas as pd
import numpy as np

# Criando uma Series
s = pd.Series([1, 3, 5, np.nan, 6, 8])
print(s)
# Saída esperada:
# 0    1.0
# 1    3.0
# 2    5.0
```

continua

```
# 3    NaN
# 4    6.0
# 5    8.0
# dtype: float64
```

Fonte: autoria própria.

O ***DataFrame*** é uma estrutura de dados bidimensional, semelhante a uma tabela de dados, que suporta colunas de diversos tipos. Essa estrutura é ideal para o manuseio de conjuntos de dados de maneira prática e eficiente, oferecendo funcionalidades semelhantes às encontradas em linguagens de programação estatística como R e em *softwares* de planilhas. No Código 27, é apresentado um exemplo de inicialização de um *DataFrame* a partir de um dicionário do *Python*. Nota-se que as chaves do dicionário viram os nomes das colunas no *DataFrame*.

Código 27 - Exemplo de criação de um *DataFrame* do *Pandas*

```
data = {'Nome': ['João', 'Ana', 'Pedro', 'Maria'],
        'Idade': [28, 22, 35, 42],
        'Cidade': ['Goiânia', 'São Paulo', 'Salvador', 'Curitiba']}
df = pd.DataFrame(data)
print(df)
# Saída esperada:
#     Nome  Idade    Goiânia
# 0   João     28        Rio
# 1    Ana     22  São Paulo
# 2  Pedro     35   Salvador
# 3  Maria     42   Curitiba
```

Fonte: autoria própria.

Cada *DataFrame* possui um índice (ou "*row label*") e colunas, ambos podendo ter nomes atribuídos. Isso possibilita acessar dados de forma rápida e eficaz usando esses rótulos, em vez de localizações numéricas.

A instalação de *Pandas* é simples e pode ser realizada por meio do **pip**, o gerenciador de pacotes do *Python*. Utiliza-se o seguinte comando no terminal para instalar a biblioteca: pip install pandas.

Assim como feito para o *NumPy*, após a instalação, é importante verificar a versão da biblioteca para assegurar que a instalação foi bem sucedida, o que pode ser feito com os comandos, como apresentado no Código 28.

Código 28 - Verificando a instalação do *Pandas*

```
import pandas as pd

print("Versão do Pandas:", pd.__version__)

# Saída esperada: Versão do Pandas: [versão instalada]
```

Fonte: autoria própria.

Trabalhar com *Pandas* começa geralmente com a carga de um conjunto de dados em um *DataFrame*. O *Pandas* suporta uma variedade de formatos de arquivo, como CSV[1], Excel[2], SQL[3], JSON[4], entre outros, facilitando a leitura e escrita de dados. O Código 29 apresenta um exemplo de leitura de dados em arquivo CSV. Nele é utilizada a função head() para exibir algumas linhas iniciais do *DataFrame* criado a partir do CSV.

Código 29 - Lendo dados de um arquivo CSV

```
df_csv = pd.read_csv('dados.csv')

print(df_csv.head())  # Exibe as primeiras cinco linhas do DataFrame
```

Fonte: autoria própria.

Após carregar os dados, a próxima etapa envolve frequentemente a limpeza desses dados. *Pandas* oferece várias funções para tratar dados faltantes, remover duplicatas, filtrar linhas ou colunas e fazer transformações nos dados.

As operações em *Pandas* são incrivelmente eficientes. Por exemplo, alterações em dados, agregações e fusões são operações que podem ser executadas de forma rápida e com sintaxe intuitiva. Isso torna *Pandas* uma ferramenta poderosa não só para análise de dados exploratória, mas também para transformações de dados. Ele pode ser utilizado antes de carregar os dados em um sistema de banco de dados ou para preparação de dados para análise posterior.

A capacidade de aplicar funções a colunas ou linhas inteiras de um *DataFrame* sem a necessidade de estruturas de repetição (*loops*) explícitas é uma das características mais valiosas de *Pandas*. Isso é possível por meio de métodos como apply e map, que permitem a aplicação de uma função a cada elemento de uma série ou *DataFrame* de maneira vetorizada. No exemplo do Código 30, a função apply é utilizada para definir uma

1 CSV - *Comma Separated Version* - arquivo texto com dados separados por vírgula ou outro caractere especial.

2 Excel, XLS ou XLSX é um dos formatos mais comuns para criação de planilhas.

3 SQL - *Structured Query Language* - linguagem de consulta e manipulação de dados mais utilizada em bancos de dados relacionais.

4 JSON - *JavaScript Object Notation* - o formato de dados mais popular atualmente para transferência de dados entre sistemas.

nova série a partir da série da coluna “Idade” do *DataFrame* em **df**. Para cada elemento da série original (Idade) é aplicada a função que dobra seu valor, resultando em uma nova série então armazenada na coluna “Idade dobrada” do *DataFrame*.

Código 30 - Aplicando uma função a uma coluna usando o método *apply*

```
df['Idade_dobrada'] = df['Idade'].apply(lambda x: x * 2)
print(df)
# Saída esperada:
#     Nome  Idade     Cidade  Idade_dobrada
# 0   João     28        Rio             56
# 1    Ana     22  São Paulo             44
# 2  Pedro     35   Salvador             70
# 3  Maria     42   Curitiba             84
```

Fonte: autoria própria.

A manipulação de índices é outra característica poderosa de *Pandas*. Permite configurar uma ou mais colunas como índice de um *DataFrame*, facilitando operações de subconjuntos e preparando dados para visualizações ou análises mais detalhadas.

Pandas também é altamente eficiente para análises complexas. Ele oferece funções como groupby para agrupamento de dados. Por exemplo, no Código 31, a função groupby é utilizada para agrupar todos os elementos pela coluna “Cidade”. Em seguida, é calculada a média dos valores na coluna “Idade”. Assim, o *DataFrame* resultante em media_idade_por_cidade contém as cidades como índices e uma única coluna com a média das idades para cada cidade. Além da média, outras funções de agregação como soma, moda, mediana, etc. são possíveis de serem aplicadas a cada grupo separadamente. Isso é especialmente útil em análises estatísticas e econômicas.

Código 31 - Agrupando dados e calculando a média da idade por cidade

```
media_idade_por_cidade = df.groupby('Cidade')['Idade'].mean()
print(media_idade_por_cidade)
# Saída esperada:
# Cidade
# Curitiba    42.0
# Rio         28.0
# Salvador    35.0
```

continua

```
# São Paulo    22.0
# Name: Idade, dtype: float64
```

Fonte: autoria própria.

Para análises mais complexas, o *Pandas* pode ser integrado com outras bibliotecas, como *NumPy*, *Matplotlib* e *Seaborn*, permitindo a criação de visualizações de dados avançadas diretamente a partir dos *DataFrames*.

Em resumo, *Pandas* é uma ferramenta essencial no kit de ferramentas de qualquer analista de dados. Com a sua rica suíte de funcionalidades para manipulação de dados e integração com outras bibliotecas de análise e visualização de dados, *Pandas* não só simplifica a manipulação de dados, mas também acelera o processo de conversão de dados brutos em percepções acionáveis.

Ao longo desta Unidade, esses conceitos e funcionalidades serão explorados em mais detalhes, proporcionando uma base sólida que permitirá aos estudantes aproveitar ao máximo o que o *Pandas* tem a oferecer para análise de dados.

2.2 Manipulação e Operações Básicas com Series e DataFrames

Antes de aprofundar nas manipulações com *DataFrames*, é essencial compreender a estrutura de dados mais simples em *Pandas*: a Series. Como dito anteriormente, uma série é um *array* unidimensional que pode armazenar dados de qualquer tipo, sendo cada elemento associado a um rótulo ou índice. Dada uma série, todos os dados nela contidos são de um mesmo tipo.

A criação de uma série é bastante flexível, permitindo a utilização de listas, *arrays* do *NumPy* ou dicionários como fonte de dados. Cada método de criação fornece características únicas à série, como a definição automática de índices quando criada a partir de um dicionário, exemplificado no Código 32.

Código 32 - Criação de *series* com *Pandas*

```
import pandas as pd

# Criando uma Series a partir de uma lista
s1 = pd.Series([1, 3, 5, None, 6, 8])
print(s1)


# Criando uma Series a partir de um dicionário
s2 = pd.Series({'a': 1, 'b': 2, 'c': 3})
print(s2)
```

Fonte: autoria própria.

Uma característica chave das séries é a capacidade de acessar e modificar seus elementos por meio dos índices. *Pandas* oferece várias formas de indexação:

- **Indexação direta:** que utiliza colchetes para acessar valores por índices numéricos ou rótulos;
- **loc**: baseado em rótulos e permite *slicing* (acessar subconjuntos do *array)* ao incluir os rótulos inicial e final;
- **iloc**: baseado na posição numérica dos elementos, semelhante ao *slicing* de listas e *arrays* em *Python*.

Essas formas de indexação proporcionam flexibilidade e precisão no acesso e manipulação dos dados. No Código 33, é possível observar diferentes exemplos de utilização das três formas de indexação citadas anteriormente. A indexação direta com s['a'] ou s[0] pode tanto utilizar um rótulo quanto a posição sequencial do elemento (nesse exemplo, o primeiro elemento da série). A indexação com loc utiliza exclusivamente os rótulos, como em s.loc['b'] para o segundo elemento ou s.loc['a':'c'] para uma faixa de valores do primeiro ao terceiro elemento da série. Por fim, a indexação com iloc utiliza exclusivamente a posição sequencial do elemento na série, assim, s.iloc[2] retornará o terceiro elemento e s.iloc[1 : 3] uma série com o segundo e o terceiro elementos.

Código 33 - Diferentes formas de indexação de séries

```
import pandas as pd

s = pd.Series([1, 3, 5, None, 6, 8], index=['a', 'b', 'c', 'd', 'e', 'f'])

# Indexação direta
print(s['a'])  # Saída: 1.0
print(s[0])    # Saída: 1.0

# Indexação com loc
print(s.loc['b'])  # Saída: 3.0

# Slicing com loc
print(s.loc['a':'c'])
# Saída esperada:
# a    1.0
# b    3.0
# c    5.0
# dtype: float64
```

continua

```
# Indexação com iloc
print(s.iloc[2])  # Saída: 5.0


# Slicing com iloc
print(s.iloc[1 : 3])
# Saída esperada:
# b    3.0
# c    5.0
# dtype: float64
```

Fonte: autoria própria.

Além da indexação, as séries também suportam *slicing*, permitindo acessar uma faixa de dados. O *slicing* em séries é feito de uma maneira muito parecida com aquele realizado em *arrays* do *NumPy*. A diferença é que, nas séries, ele pode ser realizado tanto com rótulos quanto com índices inteiros, como apresentado no Código 34, no qual há uma impressão dos elementos da série **s** do índice (rótulo) **'a'** até o **'c'** ou da posição **0** até a **3**.

Código 34 - *Slicing* de séries com rótulos e índices

```
print(s['a':'c'])  # Slicing com rótulos
print(s[0 : 3])      # Slicing com índices inteiros
# Saída esperada:
# a    1.0
# b    3.0
# c    5.0
# dtype: float64
```

Fonte: autoria própria.

As séries possuem atributos úteis como index, values, e dtype, apresentados no Código 35. Eles fornecem informações sobre os índices, os dados armazenados, e o tipo de dado, respectivamente.

Código 35 - Atributos úteis de séries em *Pandas*

```
print(s.index)    # Exibe os índices
print(s.values)   # Exibe os valores
print(s.dtype)    # Exibe o tipo de dados
```

continua

```
# Saída esperada:
# Index(['a', 'b', 'c', 'd', 'e', 'f'], dtype='object')
# [ 1.  3.  5. nan  6.  8.]
# float64
```

Fonte: autoria própria.

Operações aritméticas com séries são vetorizadas, o que significa que operações como adição, subtração ou multiplicação são aplicadas a cada elemento da série, sem a necessidade de *loops* explícitos, como pode ser visto no Código 36, no qual é adicionado o escalar 10 a cada elemento da série em **s**.

Código 36 - Operações aritméticas em séries

```
print(s + 10)
# Saída esperada:
# 0    11.0
# 1    13.0
# 2    15.0
# 3     NaN
# 4    16.0
# 5    18.0
# dtype: float64
```

Fonte: autoria própria.

A filtragem em séries também é uma operação vetorizada. Usando condições booleanas, pode-se facilmente filtrar elementos com base em critérios específicos, uma ferramenta poderosa para análise de dados. Outro método poderoso para realizar filtragem condicional é a função where. Ela retorna uma série onde as condições especificadas são atendidas; para as outras, pode substituir por um valor específico. No Código 37, são apresentadas as duas formas de filtragem. Na primeira, é retornada uma série contendo apenas os elementos da série **s** maiores que a média de seus elementos. Na segunda, a função where substitui os valores que não atendem à condição do filtro (a condição resulta em "false") por um valor padrão "*below threshold*".

Código 37 - Filtragem com condição booleana e *where*

```
# Filtrando elementos maiores que 3
print(s[s > s.mean()])
# Saída esperada:
# 2    5.0
# 4    6.0
# 5    8.0
# dtype: float64

print(s.where(s > 3, 'below threshold'))
# Saída esperada:
# a    below threshold
# b    below threshold
# c                5.0
# d                NaN
# e                6.0
# f                8.0
# dtype: object
```

Fonte: autoria própria.

Métodos de transformação como sort_values, drop_duplicates, e métodos condicionais como isnull e notnull são essenciais para preparar os dados para análise. Esses métodos permitem ordenar dados, remover duplicatas e filtrar valores nulos ou não nulos, respectivamente. Esses métodos são apresentados no Código 38.

Código 38 - Métodos de transformação de uma série

```
print(s.sort_values())  # Ordena os valores
print(s.drop_duplicates())  # Remove duplicatas
print(s.isnull())  # Detecta valores nulos
```

Fonte: autoria própria.

O método reindex permite alterar, adicionar ou remover índices de uma série, o que pode servir para alinhar séries para operações de dados. No Código 39, s.reindex reorganiza os índices já existentes na série **s** e adiciona um novo índice **g**, ao qual é atribuído o valor **NaN** (*not a number*) indicando que o valor não foi informado (faltante).

Código 39 - Utilizando *reindex* para refazer os índices

```
print(s.reindex(['g', 'f', 'e', 'd', 'c', 'b', 'a']))
# Saída esperada:
# g    NaN
# f    8.0
# e    6.0
# d    NaN
# c    5.0
# b    3.0
# a    1.0
# dtype: float64
```

Fonte: autoria própria.

A compreensão completa dessas funcionalidades básicas das séries é fundamental antes de avançar para os *DataFrames*, sendo essencialmente coleções de séries alinhadas por índices compartilhados.

Como dito anteriormente, o *DataFrame* é uma estrutura de dados bidimensional que se assemelha a uma tabela de banco de dados ou uma planilha do Excel®, composta por linhas e colunas. Cada coluna em um *DataFrame* pode ser considerada uma série e todos os dados numa coluna são do mesmo tipo.

DataFrames podem ser criados de várias formas, um exemplo, visto no Código 40, é a partir de dicionários de listas ou *arrays*, onde as chaves se tornam os nomes das colunas e os valores são as listas representando os dados. No exemplo, o *DataFrame* em df é criado a partir do dicionário em data.

Código 40 - Criando *DataFrame* a partir de um dicionário de listas

```
import pandas as pd

data = {'ID': [1, 2, 3],
        'Name': ['Alice', 'Bob', 'Charlie'],
        'Age': [25, 30, 35]}
df = pd.DataFrame(data)
print(df)
# Saída esperada:
#    ID     Name  Age
# 0   1    Alice   25
```

continua

```
# 1   2      Bob   30
# 2   3  Charlie   35
```

Fonte: autoria própria.

Acessar colunas por seus nomes retorna uma série e acessar elementos específicos pode ser realizado por meio dos métodos loc e iloc, como mostrado no Código 41.

Código 41 - Acessando colunas e elementos do *DataFrame*

```
# Acessando uma coluna
print(df['Name'])
# Saída esperada:
# 0      Alice
# 1        Bob
# 2    Charlie
# Name: Name, dtype: object

# Acessando um elemento específico
print(df.loc[0, 'Name'])
# Saída esperada:
# Alice
```

Fonte: autoria própria.

Adicionar uma coluna é simples como atribuir dados a um novo nome de coluna. Remover uma coluna ou linha pode ser feito utilizando o método drop. O parâmetro axis determina se a operação afeta linhas (axis=0) ou colunas (axis=1). O parâmetro inplace, quando definido como "True", aplica as mudanças diretamente no *DataFrame* original, modificando-o sem criar uma nova cópia. Se inplace=False (o padrão), a operação retorna um novo *DataFrame* com a modificação, deixando o original inalterado. Essa funcionalidade permite uma manipulação precisa e eficiente dos dados, conforme necessário para a análise. No exemplo do Código 42, a coluna "*Department*" é adicionada para posteriormente ser removida no próprio *DataFrame* em df, utilizando a função drop.

Código 42 - Adição e exclusão de uma coluna a um *DataFrame*

```
# Adicionando uma nova coluna
df['Department'] = ['HR', 'Tech', 'Marketing']
```

continua

```
print(df)
# Saída esperada:
#    ID     Name  Age Department
# 0   1    Alice   25         HR
# 1   2      Bob   30       Tech
# 2   3  Charlie   35  Marketing

# Excluindo uma coluna
df.drop('Department', axis=1, inplace=True)
print(df)
# Saída esperada:
#    ID     Name  Age
# 0   1    Alice   25
# 1   2      Bob   30
# 2   3  Charlie   35
```

Fonte: autoria própria.

Adicionar e remover linhas também segue um procedimento direto com append[5] e drop, como pode ser visto no Código 43. No exemplo, uma nova linha está sendo adicionada a partir de um dicionário contendo como chaves as colunas já definidas no *DataFrame*. O parâmetro ignore_index=True é informado para que o *Pandas* se encarregue de atribuir um índice sequencial para a nova linha (no exemplo, o índice **3**) em vez de tentar preservar os índices da entrada (new_row, no exemplo). Esse parâmetro é particularmente necessário quando é informado um único elemento em forma de dicionário. Para adicionar múltiplas linhas, basta utilizar uma lista de dicionários.

Código 43 - Utilização de *append* e *drop* para manipulação de linhas

```
# Adicionando uma nova linha
new_row = {'ID': 4, 'Name': 'Dave', 'Age': 28}
df = df._append(new_row, ignore_index=True)
print(df)
# Saída esperada:
#    ID     Name  Age
# 0   1    Alice   25
```

continua

5 A função *append* foi renomeada no *Pandas* a partir da versão 1.4.0. É necessário usar a função *_append* em versões mais recentes.

```
# 1   2      Bob   30
# 2   3  Charlie   35
# 3   4     Dave   28

# Excluindo uma linha
df.drop(3, inplace=True)
print(df)

# Saída esperada:
#    ID     Name  Age
# 0   1    Alice   25
# 1   2      Bob   30
# 2   3  Charlie   35
```

Fonte: autoria própria.

Além das operações básicas, o *Pandas* oferece uma variedade de métodos e atributos essenciais para a análise e manipulação de *DataFrames*. Entre eles, .shape, .index, .columns, e .dtypes fornecem informações estruturais sobre o *DataFrame*, enquanto .info() oferece um resumo conciso incluindo o tipo de dado e o número de valores não nulos em cada coluna.

Métodos como .head() e .tail(), apresentados no Código 44, são utilizados para visualizar rapidamente as primeiras ou últimas linhas do *DataFrame*, respectivamente, facilitando uma inspeção inicial dos dados ou das manipulações aplicadas.

Código 44 - Mostrando algumas linhas do *DataFrame* com *head* e *tail*

```
print(df.head(2))  # Mostra as duas primeiras linhas
print(df.tail(2))  # Mostra as duas últimas linhas
```

Fonte: autoria própria.

Para análises estatísticas rápidas, .describe() oferece um resumo que inclui contagem, média, desvio padrão, mínimo, máximo e os quartis de colunas numéricas. Outros métodos como .value_counts() e .sort_values() ajudam na exploração e ordenação dos dados.

Código 45 - Análises estatísticas rápidas do *DataFrame*

```
print(df['Age'].describe())  # Resumo estatístico da coluna 'Age'
print(df['Name'].value_counts())  # Contagem de valores únicos na coluna 'Name'
```

Fonte: autoria própria.

O método query permite filtrar linhas de um *DataFrame* com base em uma expressão de *string* condicional, o que é particularmente útil para consultas complexas e pode tornar o código mais legível. A filtragem condicional também pode ser realizada utilizando condições booleanas. Essa operação é realizada aplicando condições diretamente nas colunas, que retornam uma *Series* de valores booleanos usada para indexar o *DataFrame*. O Código 46 mostra as duas formas de filtragem. Neste exemplo, o *DataFrame* resultante deve conter apenas as pessoas com mais de 30 anos (Age > 30) e que o nome seja Charlie (Name == 'Charlie'). No comentário, há o mesmo filtro utilizando expressão booleana.

Código 46 - Filtragem condicional com método *query* e expressão booleana

```
# Filtrando dados usando query
# Com expressão booleana: df[(df['Age'] > 30) & (df['Name'] == 'Charlie')]
result = df.query("Age > 30 & Name == 'Charlie'")
print(result)
# Saída esperada:
#    ID     Name  Age
# 2   3  Charlie   35
```

Fonte: autoria própria.

Métodos como .set_index() e .reset_index() são fundamentais para manipular índices em *DataFrames*. O primeiro permite definir uma ou mais colunas como índices, enquanto o segundo restaura o índice para a configuração padrão, adicionando possivelmente o índice atual como uma coluna. No Código 47, é exemplificada a utilização de ambos os métodos. Em um primeiro momento, a coluna "*Name*" é transformada em índice do *DataFrame* (set_index). Em seguida, o índice é convertido para um número sequencial (reset_index) e a coluna "*Name*" volta a fazer parte do *DataFrame*.

Código 47 - Utilizando *set_index* e *reset_index*

```
# Alterando o índice para a coluna 'Name'
df.set_index('Name', inplace=True)
print(df)
# Saída esperada:
```

continua

```
#          ID  Age
# Name
# Alice     1   25
# Bob       2   30
# Charlie   3   35
# Dave      4   28



# Resetando para o índice padrão
df.reset_index(inplace=True)
print(df)
# Saída esperada:
#       Name  ID  Age
# 0    Alice   1   25
# 1      Bob   2   30
# 2  Charlie   3   35
# 3     Dave   4   28
```

Fonte: autoria própria.

Esses métodos e atributos formam a base para uma manipulação eficiente de *DataFrames*, permitindo que os(as) usuários(as) realizem desde tarefas simples de visualização até análises estatísticas complexas e manipulações de dados avançadas. A familiaridade com essas ferramentas é crucial para qualquer pessoa que trabalha com dados em *Pandas*.

2.3 Operações Avançadas e Análise Exploratória de Dados

O *Pandas* fornece várias funções avançadas para manipular *DataFrames*, permitindo análises complexas e agregações de dados. Funções como groupby, merge, join, e pivot são essenciais para tarefas que envolvem a reorganização, combinação e resumo de grandes conjuntos de dados.

A função groupby no *Pandas* permite agrupar dados em subconjuntos segundo os valores de uma ou mais colunas. Esse agrupamento facilita realizar operações como somas, médias, ou outras agregações específicas para cada grupo, fornecendo percepções

valiosas sobre padrões e tendências. No código 48, há um exemplo de utilização de groupby. Os dados do *DataFrame* df são agrupados pela coluna "*Department*" sendo feita a média das idades (coluna "*Age*") para cada departamento.

Código 48 - Exemplo de *groupby* para calcular a média de idade por departamento

```
grouped = df.groupby('Department')['Age'].mean()
print(grouped)
```

Fonte: autoria própria.

As operações merge e join são cruciais para combinar dados de diferentes *DataFrames* de forma eficaz, seja por meio de chaves comuns ou índices. O método merge é usado para combinações baseadas em colunas específicas, enquanto join é geralmente utilizado para combinações baseadas em índices. No exemplo do Código 49, merge é utilizado para combinar os dados de df1 com df2 utilizando a coluna "*EmployeeID*", presente em ambos os *DataFrames*. O parâmetro how='inner' configura para que o *DataFrame* resultante contenha apenas registros que aparecem nos dois *DataFrames* de entrada. Já o método join combina os dois *DataFrames* utilizando os índices como referência. Assim, caso existam colunas em comum aos dois *DataFrames* de entrada (como a "*EmployeeID*"), elas serão renomeadas adicionando os sufixos informados nos parâmetros lsuffix e rsuffix. O parâmetro how='outer' configura para que o *DataFrame* resultante contenha todos os registros que aparecem em pelo menos um *DataFrame*, de acordo com o índice.

Código 49 - Exemplo de *merge* e *join* entre *DataFrames*

```
# Exemplo de merge
merged_df = df1.merge(df2, on='EmployeeID', how='inner')
print(merged_df)

# Exemplo de join
joined_df = df.join(df2, lsuffix='_df', rsuffix='_df2', how='outer')
print(joined_df)
```

Fonte: autoria própria.

A função pivot_table do *Pandas* possibilita criar tabelas *pivot* (também chamadas de tabelas dinâmicas, principalmente em sistemas de planilhas). Tais tabelas são uma excelente maneira de resumir dados. Elas podem transformar dados de um formato de lista para uma forma tabular mais complexa, permitindo a análise multidimensional dos dados. No exemplo do Código 50, é criada uma tabela dinâmica (*pivot*) tendo como índice os produtos (*Product*) e as colunas as regiões (*Region*). Os valores em cada célula são a soma (aggfunc='sum') dos valores das vendas (*Sales*).

Código 50 - Criando uma tabela *pivot* para visualizar as vendas por produto e região

```
pivot = pd.pivot_table(df, values='Sales', index='Product', columns='Region', aggfunc='sum')
print(pivot)
```

Fonte: autoria própria.

A Análise Exploratória de Dados (*Exploratory Data Analysis* [EDA]) é reforçada por meio do uso de técnicas avançadas que ajudam a compreender a profundidade e a complexidade dos dados. Operações como groupby e tabelas *pivot* desempenham um papel crucial na visualização de agregações e na identificação de padrões ou discrepâncias nos dados.

Identificar e tratar dados faltantes é crucial para manter a precisão das análises. Funções como **.**isna() para identificar e .fillna() podem ser utilizadas para substituir dados faltantes, importante para a preparação dos dados. No Código 51, é apresentado um exemplo de utilização do método fillna(). Nesse exemplo, os valores faltantes da coluna "*Age*" estão sendo preenchidos com a média dos valores dessa coluna. O parâmetro inplace=True informa que o preenchimento deverá ser realizado no próprio *DataFrame* df.

Código 51 - Tratando dados faltantes substituindo-os pela média

```
df['Age'].fillna(df['Age'].mean(), inplace=True)
```

Fonte: autoria própria.

Pandas também permite a concatenação de *DataFrames* ao longo de um eixo especificado usando o método concat. Assim, é possível combinar dados de fontes similares ou adicionar novos dados a um conjunto existente. No Código 52, um novo *DataFrame* é criado e atribuído para variável new_data_rows. Em seguida, o *DataFrame* previamente existente em df é concatenado ao new_data_rows resultando em um novo *DataFrame* em concatenated_df. O parâmetro axis=1 é utilizado para informar que a concatenação é na dimensão das colunas (os dois *DataFrames* possuem as mesmas colunas).

Código 52 - Concatenando *DataFrames* vertical e horizontalmente

```
# Concatenando DataFrames verticalmente (adição de linhas)
new_data_rows = pd.DataFrame({'ID': [6, 7], 'Name': ['Lucas', 'Daniela'], 'Age': [31, 29]})
concatenated_df = pd.concat([df, new_data_rows], ignore_index=True)
print(concatenated_df)


# Concatenando DataFrames horizontalmente (adição de colunas)
new_data_columns = pd.DataFrame({'Department':
```

continua

```
['HR', 'Tech', 'Marketing']})

concatenated_df = pd.concat([df, new_data_columns], axis=1)
print(concatenated_df)
```

Fonte: autoria própria.

Outra característica importante do *Pandas* são as funcionalidades robustas para ler e escrever dados em diversos formatos como CSV, XLS, JSON, e SQL. Isso permite uma fácil integração com diferentes fontes de dados e facilita o compartilhamento de resultados analíticos. No Codigo 53, é exemplificado o uso do método to_csv para salvar dados em arquivo e read_csv para carregar dados de um arquivo em um *DataFrame*. O parâmetro index=False é utilizado para evitar que os índices sejam salvos no arquivo.

Código 53 - Lendo e escrevendo em formato CSV

```
# Escrevendo para CSV
df.to_csv('output.csv', index=False)

# Lendo de CSV
df_from_csv = pd.read_csv('output.csv')
print(df_from_csv)
```

Fonte: autoria própria.

A compreensão dessas operações avançadas não apenas enriquece o conjunto de ferramentas de um analista de dados, mas também assegura uma manipulação de dados eficaz e uma análise exploratória profunda. Essas habilidades são essenciais para realizar análises complexas e para preparar dados para tarefas mais avançadas como modelagem estatística ou aprendizagem de máquina.

2.4 Serialização com Pickle

A serialização é o processo de transformar objetos em um formato que pode ser restaurado mais tarde. No contexto do *Pandas*, utilizar *Pickle* para serializar *DataFrames* é extremamente útil para vários propósitos. Podem ser citados a persistência de estados de dados, o compartilhamento de modelos de aprendizagem de máquina (como os utilizados para classificação, regressão e recomendação, após treinados) entre diferentes tecnologias ou plataformas, e a otimização do desempenho por meio do *caching* (armazenamento temporário de dados frequentemente acessados para reduzir a latência em sistemas computacionais) de estados de dados.

A serialização com *Pickle* oferece benefícios significativos. Ela permite salvar o estado completo de *DataFrames* ou qualquer objeto *Python*, garantindo que todos os dados sejam carregados exatamente como foram salvos. Isso é crucial para a continuidade em ambientes de produção ou análise de dados. Integrado nativamente com *Python* e *Pandas*, *Pickle* simplifica a serialização e desserialização de dados complexos com poucas linhas de código, eliminando a necessidade de desenvolver *parsers* (convertedores de dados) complicados. Além disso, sua capacidade de processar rapidamente grandes volumes de dados o torna ideal para armazenar estados de dados de maneira eficiente, especialmente em casos onde recriar esses estados seria computacionalmente oneroso.

No Código 54, é exemplificada a utilização de serialização e desserialização utilizando *Pickle*. Inicialmente é criado um *DataFrame* (df) que posteriormente é serializado em um arquivo dataframe.pkl (to_pickle). Em seguida, o mesmo arquivo é carregado em uma nova variável (loaded_df) utilizando o método read_pickle.

Código 54 - Serialização com *Pickle* e *Pandas*

```
import pandas as pd


# Suponha que df seja um DataFrame Pandas complexo
df = pd.DataFrame({
    'A': range(1, 6),
    'B': ['A', 'B', 'C', 'D', 'E'],
    'C': pd.date_range('20230101', periods=5)
})


# Serializando o DataFrame usando Pickle
df.to_pickle('dataframe.pkl')


# Desserializando o DataFrame
loaded_df = pd.read_pickle('dataframe.pkl')
print(loaded_df)
# Saída esperada:
#    A  B          C
# 0  1  A 2023-01-01
# 1  2  B 2023-01-02
# 2  3  C 2023-01-03
# 3  4  D 2023-01-04
# 4  5  E 2023-01-05
```

Enquanto *Pickle* é poderoso, é importante notar que nunca se deve desserializar dados *Pickle* de fontes não confiáveis. *Pickle* não é seguro contra códigos maliciosos e pode executar código arbitrário durante a desserialização.

Além do *Pickle*, existem várias outras opções eficazes para serializar *DataFrames* no *Pandas*, cada uma com características próprias que as tornam adequadas para diferentes contextos e necessidades[6].

Parquet é uma opção popular para ambientes de *big data* (grandes volumes de dados), oferecendo armazenamento eficiente em um formato de arquivo colunar ideal para operações de leitura eficiente em grandes *datasets*. A integração com *Pandas* se dá por meio dos métodos to_parquet e read_parquet do *DataFrame*.

Feather proporciona uma leitura e escrita extremamente rápidas e é especialmente útil para transferências de dados entre R e *Python*, mantendo a integridade do tipo de dados. Esse formato pode ser manipulado em *Pandas* usando os métodos to_feather, e read_feather do *DataFrame*.

Para aqueles que trabalham com armazenamento hierárquico de grandes conjuntos de dados científicos, HDF5 oferece uma solução robusta, suportando uma variedade de cargas de dados com sua capacidade de armazenar *arrays* multidimensionais. A integração com *Pandas* pode ser realizada por meio dos métodos to_hdf e read_hdf do *DataFrame*.

Por fim, a integração com bancos de dados SQL é facilitada por meio dos métodos to_sql e read_sql do *DataFrame*. Tais métodos permitem interações diretas com bases de dados SQL por meio da biblioteca *SQLAlchemy*, proporcionando uma maneira conveniente de integrar análises de dados com sistemas de gerenciamento de banco de dados tradicionais.

2.5 Visualização de Dados e Integração com Ferramentas de Análise

Pandas oferece excelente suporte para visualizações diretamente a partir de *DataFrames*, com integração direta com bibliotecas de visualização como *Matplotlib* e *Seaborn*. Como dito anteriormente, *Matplotlib* é a base para a plotagem em *Python*, permitindo a criação de uma ampla variedade de gráficos estáticos, animados e interativos. Por exemplo, pode-se usar para criar um simples gráfico de linhas ou histograma a partir de um *DataFrame*.

Seaborn, construído sobre *Matplotlib*, adiciona uma camada de abstração mais alta, ideal para criar gráficos estatísticos complexos com menos código. *Seaborn* trabalha diretamente com *DataFrames Pandas* e pode ser usado para criar gráficos que resumem as relações nos dados.

6 Uma documentação completa sobre serialização com *Pandas* pode ser acessada em: https://pandas.pydata.org/docs/reference/frame.html#serialization-io-conversion

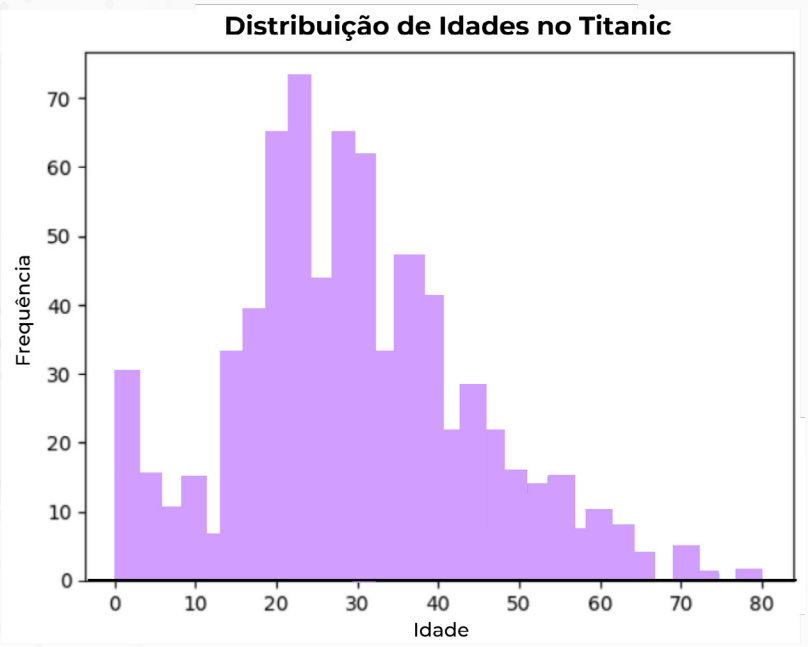
Código 55 - Geração de gráfico a partir de *DataFrame*

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Carregando o dataset do Titanic diretamente do GitHub
url = 'https://raw.githubusercontent.com/mwaskom/seaborn-data/master/titanic.csv'
titanic = pd.read_csv(url)

# Histograma da distribuição de idades
titanic['age'].dropna().plot(kind='hist', bins=30, color='c',
title='Distribuição de Idades no Titanic')
plt.xlabel('Idade')
plt.ylabel('Frequência')
plt.show()
```

Fonte: autoria própria.

O Código 55 carrega os dados diretamente do *GitHub* e gera um histograma da distribuição de idades dos indivíduos no *dataset*, apresentado na Figura 5. Esses gráficos são ferramentas eficazes para explorar visualmente os dados e extrair percepções**.** O uso de URLs para carregar dados facilita o acesso e manipulação de *datasets* em análises exploratórias.

Figura 7 - Histograma para mostrar a distribuição das idades dos passageiros



Fonte: autoria própria.

Na prática, *Pandas* é amplamente utilizado para análise de dados em muitos setores, manipulando desde dados financeiros até *datasets* científicos. A capacidade de ler e escrever dados em uma variedade de formatos facilita o acesso a *datasets* reais. *Pandas* pode importar dados diretamente de arquivos CSV, Excel, ou bases de dados SQL, o que permite trabalhar com dados reais de forma eficiente.

Integrar *Pandas* com bibliotecas como *NumPy, SciPy* e *Scikit-learn* permite aos(às) cientistas de dados preparar, analisar e modelar dados complexos para percepções profundas e previsões. Além disso, dados reais frequentemente disponíveis em repositórios públicos, como *UCI Machine Learning Repository, Kaggle* ou *APIs* públicas, podem ser facilmente manipulados e analisados com *Pandas*.

2.6 Notebook Colab

Introdução ao Pandas e DataFrames

Antes de iniciarmos o conteúdo deste Notebook Colab é importante reforçar que os códigos disponíveis abaixo, bem como os métodos e os seus parâmetros de entrada, estão explicados e comentados nas Seções do Ebook. Realizaremos neste ambiente, testes de vários comandos para que você experimente na prática os conceitos descritos anteriormente.

Nesta seção é apresentado o Pandas, uma biblioteca de software fundamental para a manipulação e análise de dados em Python, criada por Wes McKinney em 2008. Destaca-se pela capacidade de lidar eficientemente com grandes e complexas estruturas de dados, sendo amplamente utilizada em campos como ciência de dados, economia e biologia.

A biblioteca introduz estruturas como Series e DataFrames, construídas sobre arrays do NumPy, que facilitam o trabalho com dados relacionais de forma intuitiva e eficaz. Essas estruturas permitem desde simples operações de leitura de dados até análises estatísticas avançadas, tornando o Pandas uma ferramenta indispensável para profissionais da área.

O Pandas já vem instalado nesse Google Colab. Mas, para instalá-lo em seu ambiente local, você pode usar o comando abaixo:

```
!pip install pandas
```

```
Requirement already satisfied: pandas in /usr/local/lib/python3.10/dist-packages (2.1.4)
Requirement already satisfied: numpy<2,>=1.22.4 in /usr/local/lib/python3.10/dist-packages (from pandas) (1.26.4)
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/python3.10/dist-packages (from pandas) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-packages (from pandas) (2024.1)
Requirement already satisfied: tzdata>=2022.1 in /usr/local/lib/python3.10/dist-packages (from pandas) (2024.1)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from python-dateutil>=2.8.2->pandas) (1.16.0)
```

Verificando instalação do Pandas

```
import pandas as pd
print("Versão do Pandas:", pd.__version__)
```

Versão do Pandas: 2.1.4

Inicialmente vamos criar e trabalhar com Series:

```
import pandas as pd
import numpy as np


# Criando uma Series
s = pd.Series([1, 3, 5, np.nan, 6, 8])
s
```

	0
0	1.0
1	3.0
2	5.0
3	NaN
4	6.0
5	8.0

dtype: float64

Um DataFrame é composto de várias séries:

```
data = {'Nome': ['João', 'Ana', 'Pedro', 'Maria'],
        'Idade': [28, 22, 35, 42],
        'Cidade': ['Goiânia', 'São Paulo', 'Salvador', 'Curitiba']}
df = pd.DataFrame(data)
df
```

	Nome	Idade	Cidade
0	João	28	Goiânia
1	Ana	22	São Paulo
2	Pedro	35	Salvador
3	Maria	42	Curitiba

Manipulação e Operações Básicas com Series e DataFrames

Esta seção detalha como manipular as estruturas de dados fundamentais do *Pandas*, enfocando em *Series* e *DataFrames*. *Series*, descritas como *arrays* unidimensionais com indexação flexível, permitem armazenar diversos tipos de dados, cada um com um índice associado. *DataFrames*, por outro lado, são estruturas bidimensionais semelhantes a tabelas, que suportam colunas de vários tipos. Esta seção abrange desde a criação dessas estruturas até operações básicas como adição e remoção de dados, indexação e *slicing*, que são essenciais para a manipulação eficiente dos dados. A capacidade de realizar operações aritméticas vetorizadas e filtragem condicional também é enfatizada, mostrando como o *Pandas* simplifica a análise de dados sem a necessidade de *loops* explícitos.

```
# Criando uma Series a partir de uma lista
s1 = pd.Series([1, 3, 5, None, 6, 8])
s1
```

	0
0	1.0
1	3.0
2	5.0
3	NaN
4	6.0
5	8.0

dtype: float64

```
# Criando uma Series a partir de um dicionário
s2 = pd.Series({'a': 1, 'b': 2, 'c': 3})
s2
```

	0
a	1
b	2
c	3

dtype: int64

```
s = pd.Series([1, 3, 5, None, 6, 8], index=['a', 'b', 'c', 'd', 'e', 'f'])


print("Indexação direta")
print(s['a'])
```

continua

```
print(s[0])



print("\nIndexação com loc")
print(s.loc['a'])



print("\nIndexação com iloc")
print(s.iloc[0])
```

```
Indexação direta
1.0
1.0

Indexação com loc
1.0

Indexação com iloc
1.0
<ipython-input-7-397b4298c8bb>:6: FutureWarning: Series.__getitem__ treating keys as positions is deprecated. In a future version, integer keys will always be treated as labels (consistent with DataFrame behavior). To access a value by position, use `ser.iloc[pos]`
  print(s[0])
```

```
print("Slicing com rótulos")
print(s['a':'c'])

print("\nSlicing com índices inteiros")
print(s[0:3])
```

```
Slicing com rótulos
a    1.0
b    3.0
c    5.0
dtype: float64

Slicing com índices inteiros
a    1.0
b    3.0
c    5.0
dtype: float64
```

```
print(s.index)    # Exibe os índices
print(s.values)   # Exibe os valores
print(s.dtype)    # Exibe o tipo de dados
```

```
RangeIndex(start=0, stop=6, step=1)
[ 1.  3.  5. nan  6.  8.]
float64
```

```
# Somando 10 em todos os elementos não nulos da série
print(s + 10)
```

```
a    11.0
b    13.0
c    15.0
d     NaN
e    16.0
f    18.0
dtype: float64
```

```
print("Filtrando elementos maiores que a média:")
print(s[s > s.mean()])

print("\nSubstituindo valores menores ou iguais a 3:")
print(s.where(s > 3, 'abaixo do limite'))
```

```
Filtrando elementos maiores que a média:
c    5.0
e    6.0
f    8.0
dtype: float64

Substituindo valores menores ou iguais a 3:
a    abaixo do limite
b    abaixo do limite
c                 5.0
d    abaixo do limite
e                 6.0
f                 8.0
dtype: object
```

```
print(s.sort_values())  # Ordena os valores
print(s.drop_duplicates())  # Remove duplicatas
print(s.isnull())  # Detecta valores nulos
```

```
0    1.0
1    3.0
2    5.0
4    6.0
5    8.0
3    NaN
```

```
dtype: float64
0    1.0
1    3.0
2    5.0
3    NaN
4    6.0
5    8.0
dtype: float64
0    False
1    False
2    False
3     True
4    False
5    False
dtype: bool
```

```
print(s.reindex(['g', 'f', 'e', 'd', 'c', 'b', 'a']))
```

```
g    NaN
f    8.0
e    6.0
d    NaN
c    5.0
b    3.0
a    1.0
dtype: float64
```

DataFrames

```
data = {'ID': [1, 2, 3],
        'Name': ['Alice', 'Bob', 'Charlie'],
        'Age': [25, 30, 35]}
df = pd.DataFrame(data)
df
```

	ID	Name	Age
0	1	Alice	25
1	2	Bob	30
2	3	Charlie	35

```
# Acessando uma coluna
df['Name']
```

```
    Name
0  Alice
1    Bob
```

```
2    Charlie
dtype: object
```

```
# Acessando um elemento específico
df.loc[0, 'Name']
```

```
"Alice"
```

```
# Adicionando uma nova coluna
df['Department'] = ['HR', 'Tech', 'Marketing']
df
```

	ID	Name	Age	Department
0	1	Alice	25	HR
1	2	Bob	30	Tech
2	3	Charlie	35	Marketing

```
# Excluindo uma coluna
df.drop('Department', axis=1, inplace=True)
df
```

	ID	Name	Age
0	1	Alice	25
1	2	Bob	30
2	3	Charlie	35

```
# Adicionando uma nova linha
new_row = {'ID': 4, 'Name': 'Dave', 'Age': 28}
df = df._append(new_row, ignore_index=True)
df
```

	ID	Name	Age
0	1	Alice	25
1	2	Bob	30
2	3	Charlie	35
3	4	Dave	28

```
# Excluindo uma linha
df.drop(3, inplace=True)
df
```

	ID	Name	Age
0	1	Alice	25
1	2	Bob	30
2	3	Charlie	35

```
# Mostra as duas primeiras linhas
df.head(2)
```

	ID	Name	Age
0	1	Alice	25
1	2	Bob	30

```
# Mostra as duas últimas linhas
df.tail(2)
```

	ID	Name	Age
1	2	Bob	30
2	3	Charlie	35

```
# Resumo estatístico da coluna 'Age'
df.describe()
```

	ID	Age
count	3.0	3.0
mean	2.0	30.0
std	1.0	5.0
min	1.0	25.0
25%	1.5	27.5
50%	2.0	30.0
75%	2.5	32.5
max	3.0	35.0

```
# Contagem de valores únicos na coluna 'Name'
df['Name'].value_counts()
```

Name	count
Alice	1
Bob	1
Charlie	1

dtype: int64

```
# Filtrando dados usando query
result = df.query("Age > 30 & Name == 'Charlie'")
result
```

	ID	Name	Age
2	3	Charlie	35

```
# Com expressão booliana:
df[(df['Age'] > 30) & (df['Name'] == 'Charlie')]
```

	ID	Name	Age
2	3	Charlie	35

```
# Alterando o índice para a coluna 'Name'
df.set_index('Name', inplace=True)
df
```

Name	ID	Age
Alice	1	25
Bob	2	30
Charlie	3	35
Dave	4	28

```
# Resetando para o índice padrão
df.reset_index(inplace=True)
df
```

	Name	ID	Age
0	Alice	1	25
1	Bob	2	30
2	Charlie	3	35
3	Dave	4	28

Operações Avançadas e Análise Exploratória de Dados

Esta seção explora funcionalidades mais complexas do *Pandas* que permitem uma manipulação sofisticada e análise detalhada de grandes conjuntos de dados. São abordados métodos como groupby, que agrupa dados para operações de agregação; merge e join, para combinar diferentes *DataFrames*; e pivot_table, para criar resumos multidimensionais e visualizações. Tais ferramentas são cruciais para a Análise Exploratória de Dados (EDA), facilitando a identificação de padrões, tendências e anomalias. A seção também enfatiza a importância de tratar dados faltantes e a habilidade de realizar consultas complexas e filtros usando expressões condicionais, equipando os/as usuários/as para lidar efetivamente com desafios de análise de dados complexos no mundo real.

A função *groupby* no *Pandas* permite agrupar dados em subconjuntos segundo os valores de uma ou mais colunas.

```
df = pd.DataFrame({
    'ID': [1, 2, 4, 5, 6],
    'Nome': ['João Silva', 'Maria Souza', 'Ana Costa', 'Carlos Andrade', 'Eduardo Pereira'],
    'Telefone': ['(62) 98123-4567', '(62) 97654-3210', '(62) 99876-5432', '(62) 91234-5678', '(62) 94567-8901'],
    'Idade': [25, 30, 28, 32, 40],
    'Departamento': ['RH', 'Financeiro', 'Vendas', 'Financeiro', 'Vendas']
})
df
```

	ID	Nome	Telefone	Idade	Departamento
0	1	João Silva	(62) 98123-4567	25	RH
1	2	Maria Souza	(62) 97654-3210	30	Financeiro
2	4	Ana Costa	(62) 99876-5432	28	Vendas
3	5	Carlos Andrade	(62) 91234-5678	32	Financeiro
4	6	Eduardo Pereira	(62) 94567-8901	40	Vendas

```
df.groupby('Departamento')['Idade'].mean()
```

Departamento	Idade
Financeiro	31.0
RH	25.0
Vendas	34.0

dtype: float64

```
df2 = pd.DataFrame({
    'ID': [1, 2, 3, 4, 5],
    'Username': ['joao.silva', 'maria.souza', 'pedro.oliveira', 'ana.costa', 'carlos.andrade'],
    'Email': ['joao.silva@email.com', 'maria.souza@email.com', 'pedro.oliveira@email.com', 'ana.
costa@email.com', 'carlos.andrade@email.com']
})

# Exemplo de merge
pd.merge(df, df2, on='ID', how='inner')
```

	ID	Nome	Telefone	Idade	Departamento	Username	Email
0	1	João Silva	(62) 98123-4567	25	RH	joao.silva	joao.silva@email.com
1	2	Maria Souza	(62) 97654-3210	30	Financeiro	maria.souza	maria.souza@email.com
2	4	Ana Costa	(62) 99876-5432	28	Vendas	ana.costa	ana.costa@email.com
3	5	Carlos Andrade	(62) 91234-5678	32	Financeiro	carlos.andrade	carlos.andrade@email.com

```
# Exemplo de join
df.join(df2, lsuffix='_df', rsuffix='_df2', how='outer')
```

D_df	Nome	Telefone	Idade	Departamento	ID_df2	Username	Email
1	João Silva	(62) 98123-4567	25	RH	1	joao.silva	joao.silva@email.com
2	Maria Souza	(62) 97654-3210	30	Financeiro	2	maria.souza	maria.souza@email.com
4	Ana Costa	(62) 99876-5432	28	Vendas	3	pedro.oliveira	pedro.oliveira@email.com
5	Carlos Andrade	(62) 91234-5678	32	Financeiro	4	ana.costa	ana.costa@email.com
6	Eduardo Pereira	(62) 94567-8901	40	Vendas	5	carlos.andrade	carlos.andrade@email.com

Tabelas dinâmicas ou pivô:

```
df['Salário'] = [8000, 7000, 3000, 4000, 5000]
df['Cidade'] = ['Goiânia', 'Goiânia', 'Trindade', 'Trindade', 'Anápolis']
df
```

	ID	Nome	Telefone	Idade	Departamento	Salário	Cidade
0	1	João Silva	(62) 98123-4567	25	RH	8000	Goiânia
1	2	Maria Souza	(62) 97654-3210	30	Financeiro	7000	Goiânia
2	4	Ana Costa	(62) 99876-5432	28	Vendas	3000	Trindade
3	5	Carlos Andrade	(62) 91234-5678	32	Financeiro	4000	Trindade
4	6	Eduardo Pereira	(62) 94567-8901	40	Vendas	5000	Anápolis

```
salarios = pd.pivot_table(df, values='Salário', index='Cidade',
columns='Departamento', aggfunc='mean')
salarios
```

Departamento Cidade	Financeiro	RH	Vendas
Anápolis	NaN	NaN	5000.0
Goiânia	7000.0	8000.0	NaN
Trindade	4000.0	NaN	3000.0

```
# Tratando dados faltantes substituindo-os pela média.
salarios.fillna(salarios.mean(), inplace=True)
salarios
```

Departamento Cidade	Financeiro	RH	Vendas
Anápolis	5500.0	8000.0	5000.0
Goiânia	7000.0	8000.0	4000.0
Trindade	4000.0	8000.0	3000.0

```
# Escrevendo para CSV
df.to_csv('output.csv', index=False)
```

```
# Lendo de CSV
df_from_csv = pd.read_csv('output.csv')
df_from_csv
```

	ID	Nome	Telefone	Idade	Departamento	Salário	Cidade
0	1	João Silva	(62) 98123-4567	25	RH	8000	Goiânia
1	2	Maria Souza	(62) 97654-3210	30	Financeiro	7000	Goiânia
2	4	Ana Costa	(62) 99876-5432	28	Vendas	3000	Trindade
3	5	Carlos Andrade	(62) 91234-5678	32	Financeiro	4000	Trindade
4	6	Eduardo Pereira	(62) 94567-8901	40	Vendas	5000	Anápolis

Serialização com *Pickle*

O *Pandas* utiliza a serialização para preservar o estado dos *DataFrames*, utilizando a biblioteca *Pickle* do *Python*. Este processo é crucial para tarefas como a persistência de dados, transferência de estados de dados entre plataformas ou tecnologias e otimização do desempenho por meio do armazenamento de estados de dados. O uso de *Pickle* permite a serialização e desserialização eficiente de *DataFrames*, garantindo que os dados sejam restaurados exatamente como foram salvos. A integração nativa com *Pandas* simplifica esse processo, permitindo a manipulação de grandes volumes de dados de forma eficiente. No entanto, é destacada a precaução de nunca desserializar dados de fontes não confiáveis com *Pickle*, devido ao risco de execução de código arbitrário.

```
# Suponha que df seja um DataFrame Pandas complexo
df
```

	ID	Nome	Telefone	Idade	Departamento	Salário	Cidade
0	1	João Silva	(62) 98123-4567	25	RH	8000	Goiânia
1	2	Maria Souza	(62) 97654-3210	30	Financeiro	7000	Goiânia
2	4	Ana Costa	(62) 99876-5432	28	Vendas	3000	Trindade
3	5	Carlos Andrade	(62) 91234-5678	32	Financeiro	4000	Trindade
4	6	Eduardo Pereira	(62) 94567-8901	40	Vendas	5000	Anápolis

```
# Serializando o DataFrame usando Pickle (salva o DataFrame no arquivo dataframe.pkl)
df.to_pickle('dataframe.pkl')
```

```
# Desserializando o DataFrame (carrega o DataFrame disponivel no arquivo dataframe.pkl)
loaded_df = pd.read_pickle('dataframe.pkl')
loaded_df
```

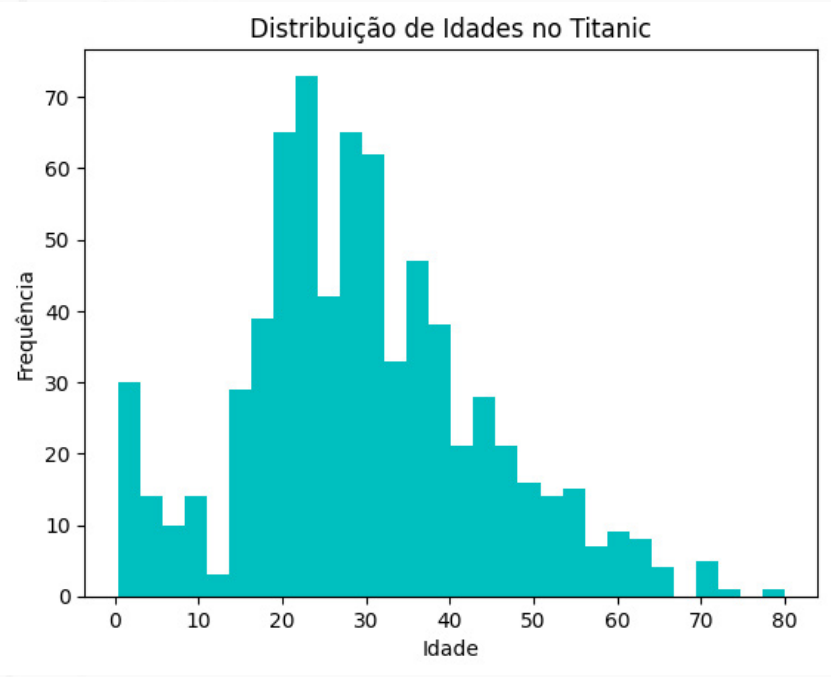
	ID	Nome	Telefone	Idade	Departamento	Salário	Cidade
0	1	João Silva	(62) 98123-4567	25	RH	8000	Goiânia
1	2	Maria Souza	(62) 97654-3210	30	Financeiro	7000	Goiânia
2	4	Ana Costa	(62) 99876-5432	28	Vendas	3000	Trindade
3	5	Carlos Andrade	(62) 91234-5678	32	Financeiro	4000	Trindade
4	6	Eduardo Pereira	(62) 94567-8901	40	Vendas	5000	Anápolis

Visualização de Dados e Integração com Ferramentas de Análise

O *Pandas* consegue se integrar com poderosas bibliotecas de visualização como *Matplotlib* e *Seaborn* para facilitar a visualização direta de dados a partir de *DataFrames*. Essa integração permite aos usuários criar uma variedade de gráficos estatísticos complexos de forma simplificada e eficiente, aproveitando a estrutura de dados do *Pandas* para maximizar a clareza e o impacto visual das análises. Por exemplo, é possível gerar rapidamente gráficos de linhas, histogramas e outras visualizações que resumem relações e distribuições nos dados, proporcionando *insights* valiosos por meio de visualizações intuitivas e acessíveis.

```
import matplotlib.pyplot as plt
import seaborn as sns


# Carregando o dataset do Titanic diretamente do GitHub
url = 'https://raw.githubusercontent.com/mwaskom/seaborn-data/master/titanic.csv'
titanic = pd.read_csv(url)


# Histograma da distribuição de idades
titanic['age'].dropna().plot(kind='hist', bins=30, color='c', title='Distribuição de Idades no
Titanic')
plt.xlabel('Idade')
plt.ylabel('Frequência')
plt.show()
```

Figura 8 - Distribuição de Idades no Titanic



Fonte: autoria própia

2.7 Saiba Mais...

Pandas Development Team. *Pandas documentation.*

Python para Processamento de Dados

Unidade III

Visualização de Informações com *Matplotlib* e *Seaborn*

Unidade III - Visualização de Informações com Matplotlib e Seaborn

3.1 Fundamentos da Visualização de Dados

A visualização de dados é uma faceta crucial da análise de dados, essencial para transformar conjuntos de dados complexos em representações visuais que são imediatamente compreensíveis e informativas. Ela permite que analistas, cientistas de dados e partes interessadas descubram padrões, identifiquem tendências e comuniquem resultados de forma eficaz. A visualização também facilita a tomada de decisões ao apresentar dados de uma maneira que pode ser rapidamente absorvida e interpretada.

No ecossistema *Python*, *Matplotlib* e *Seaborn* são duas das bibliotecas mais populares para a realização de tarefas de visualização de dados. *Matplotlib* é amplamente reconhecida por sua versatilidade e capacidade de customização. Ela oferece uma vasta gama de tipos de gráficos, desde os mais simples gráficos de linhas e barras até visualizações complexas de dados em três dimensões. A biblioteca é projetada para ser intuitiva para quem vem de ambientes de programação que utilizam *MATLAB*, facilitando a transição para *Python*.

Seaborn é construído sobre a base do *Matplotlib* e oferece uma interface de alto nível, mais acessível, com configurações padrão esteticamente mais agradáveis e a capacidade de criar visualizações estatísticas complexas com menos código. *Seaborn* é especialmente forte quando se trata de gráficos que envolvem a análise de variáveis categóricas. Tem integração nativa com as estruturas de dados do *Pandas*, facilitando ainda mais a análise e visualização de grandes conjuntos de dados.

Essas bibliotecas são ferramentas indispensáveis no arsenal de qualquer analista de dados, proporcionando os meios para visualizar informações de maneiras que não apenas destacam percepções importantes, mas também as comunicam claramente. Ao utilizar *Matplotlib* e *Seaborn*, os(as) usuários(as) podem elaborar narrativas visuais fundamentais para a análise exploratória de dados, a apresentação de resultados e a tomada de decisões baseada em dados.

Para os exemplos dessa Unidade, será utilizado o *dataset* Iris[7]. Trata-se de um dos conjuntos de dados mais conhecidos na área de estatística e aprendizado de máquina, frequentemente utilizado para testar algoritmos de classificação e visualização de dados. Ele contém 150 amostras de três espécies diferentes da flor íris (*Iris setosa*, *Iris virgínica* e *Iris versicolor*), com 50 amostras para cada espécie. Cada amostra tem quatro características medidas: o comprimento e a largura das sépalas e das pétalas, todas expressas em centímetros.

7 O Dataset Iris está disponível oficialmente em https://archive.ics.uci.edu/dataset/53/iris. Ele também está presente na maioria das bibliotecas de análise de dados, como o Scikit-Learn.

O *dataset* Iris é valorizado por sua simplicidade e conveniência, sendo adequado para tarefas iniciais de aprendizado de máquina e análises exploratórias de dados. Para carregar o *dataset* em um *DataFrame* do *Pandas*, será utilizado o Código 56.

Código 56 - Carregando o *dataset* do Iris diretamente do *GitHub*

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

url = 'https://raw.githubusercontent.com/mwaskom/seaborn-data/master/iris.csv'
iris = pd.read_csv(url)
```

Fonte: autoria própria.

3.2 Visualizações com Matplotlib

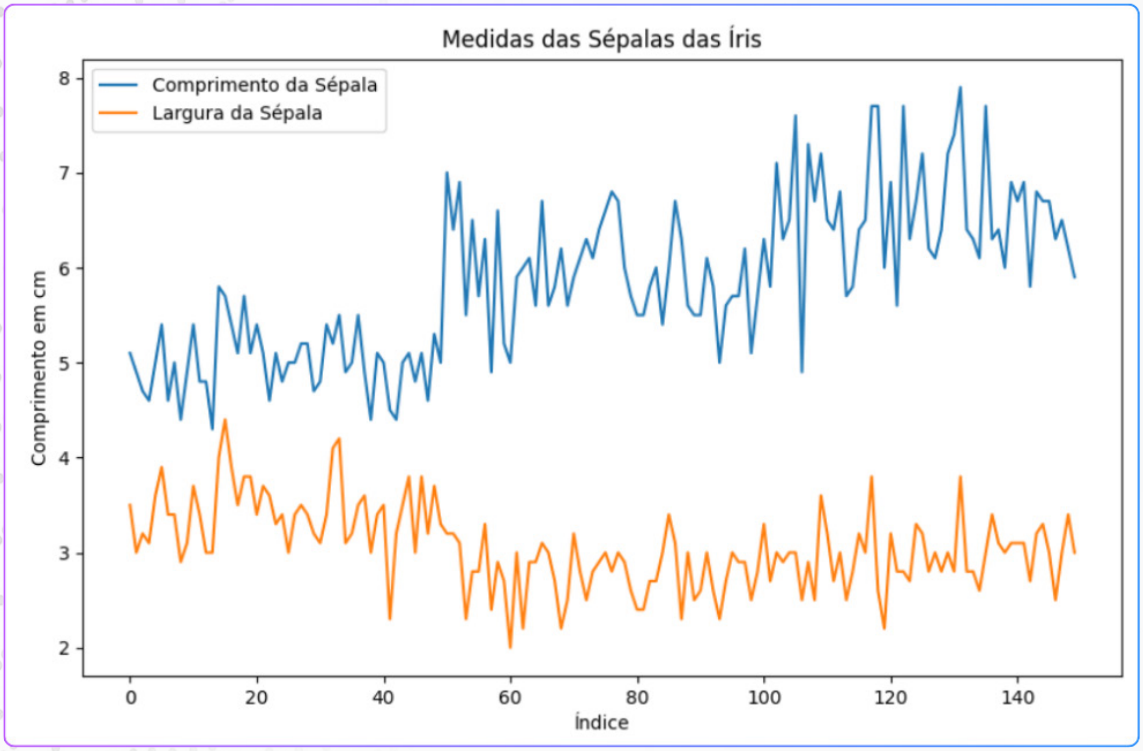
Nessa seção, será explorado como utilizar o *Matplotlib* para criar visualizações básicas e avançadas, apresentando exemplos práticos que ilustram essas técnicas. Começando com gráficos básicos, *Matplotlib* simplifica a criação de visualizações comuns, como gráficos de linhas, barras e histogramas. Por exemplo, para visualizar as tendências de vendas ao longo do tempo, um gráfico de linhas, como apresentado na Figura 6 é ideal. O Código 57 é responsável por gerá-lo. O método plt.plot() do *Matplotlib* é fundamental para isso. Ele aceita os eixos **x** e **y** como argumentos e opcionalmente marcadores e estilos de linha.

Código 57 - Criando um gráfico de linhas simples

```
# Define o tamanho da figura (largura, altura) em polegadas
plt.figure(figsize=(10, 6))

plt.plot(iris['sepal_length'], label='Comprimento da Sépala')
plt.plot(iris['sepal_width'], label='Largura da Sépala')
plt.title('Medidas das Sépalas das Íris')
plt.xlabel('Índice')
plt.ylabel('Comprimento em cm')
plt.legend()
plt.show()
```

Fonte: autoria própria.

Figura 9 - Gráfico de linhas com medidas de comprimento e largura das sépalas das íris



Fonte: autoria própria.

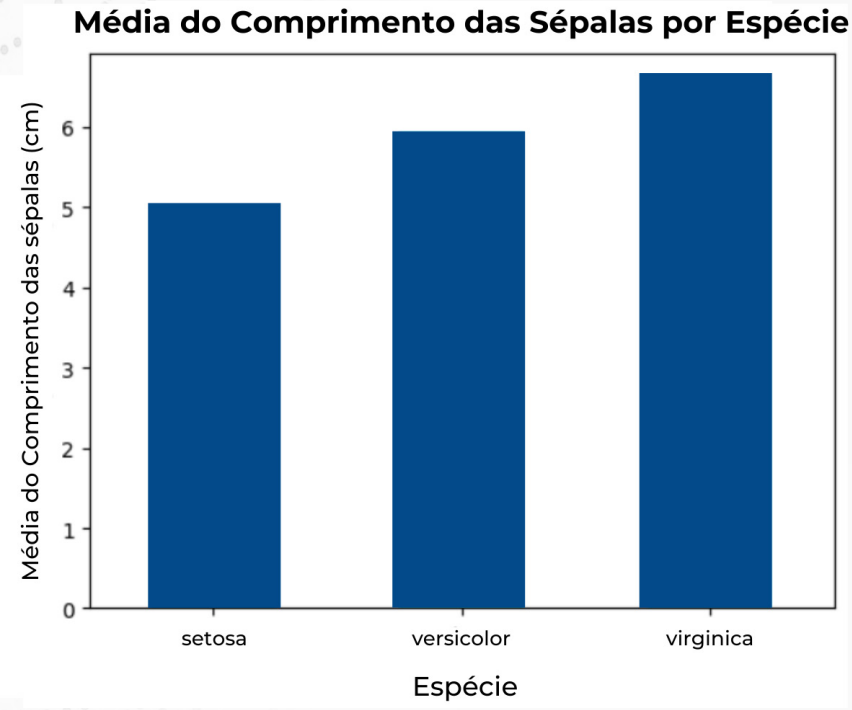
Usados para comparações de categorias, os gráficos de barras são gerados pelo método plt.bar(). Há ainda a possibilidade de utilizar o método plot do *DataFrame*, como apresentado na Figura 7 e gerado pelo Código 58. Nesse exemplo, é inicialmente realizado um agrupamento pelas espécies (species). Para cada espécie, é calculada a média (mean()) do comprimento das sépalas (['sepal_length']), resultando em um novo *DataFrame* em sepal_length_means. A partir desse *DataFrame*, o gráfico de barras (kind='bar') é plotado pelo método plot utilizando a cor ciano (color='cyan') com transparência de 70% (alpha=0.7).

Código 58 - Criando um gráfico de barras para a média do comprimento das sépalas

```
# Agrupando os dados por espécie e calculando a média do comprimento das sépalas
sepal_length_means = iris.groupby('species')['sepal_length'].mean()

# Criando um gráfico de barras
sepal_length_means.plot(kind='bar', color='cyan', alpha=0.7)

plt.title('Média do Comprimento das Sépalas por Espécie')
plt.xlabel('Espécie')

plt.ylabel('Média do Comprimento das Sépalas (cm)')
plt.show()
```

Fonte: autoria própria.

Figura 10 - Gráfico de barras com média do comprimento das sépalas por espécie



Fonte: autoria própria.

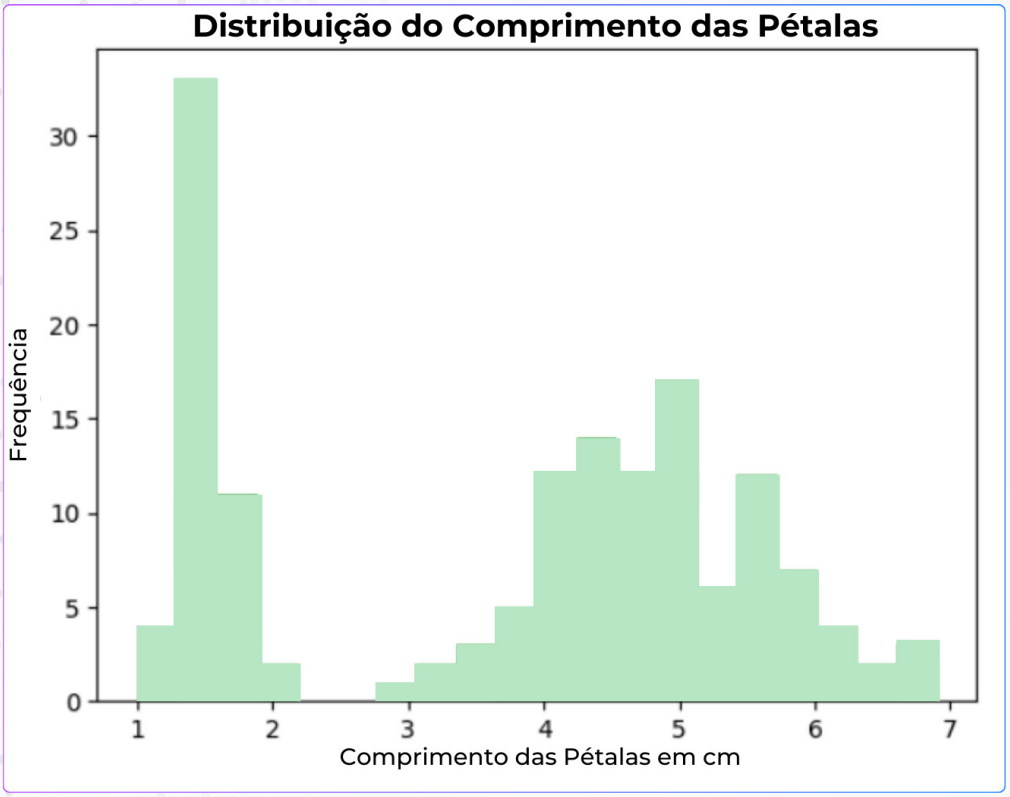
Histogramas são utilizados para observar a distribuição de dados numéricos. O método plt.hist() facilita isso, permitindo definir o número de compartimentos (bins) e a transparência (alpha). No Código 59, é apresentado um exemplo de uso, gerando o histograma da Figura 8. Nesse exemplo, iris['petal_length'] possui os dados de entrada, bins=20 configura a quantidade de barras no histograma, color='green' define a cor (verde) e alpha=0.7 uma porcentagem de transparência (70%).

Código 59 - Criando um histograma para uma variável

```
# Define a cor, a transparência (alpha) e o número de barras (bins)
plt.hist(iris['petal_length'], bins=20, color='green', alpha=0.7)


plt.title('Distribuição do Comprimento das Pétalas')
plt.xlabel('Comprimento das Pétalas em cm')
plt.ylabel('Frequência')
plt.show()
```

Fonte: autoria própria.

Figura 11 - Distribuição do comprimento das pétalas



Fonte: autoria própria.

Gráficos de área são ideais para mostrar quantidades acumuladas. Utiliza-se plt.stackplot() para criar gráficos de área, como apresentado no Código 60, gerando o gráfico da Figura 9. No exemplo, há uma variável **x** que recebe uma sequencia de números com o tamanho das amostras presente no DataFrame **iris**. Os valores de comprimento e largura das sépalas são atribuídos às variáveis y_sepal_length e y_sepal_width, respectivamente, substituindo valores nulos por zero (fillna(0)). Utilizando o método stackplot as duas variáveis são desenhadas no gráfico.

Código 60 - Criando um gráfico de área com a distribuição acumulada das medidas das sépalas

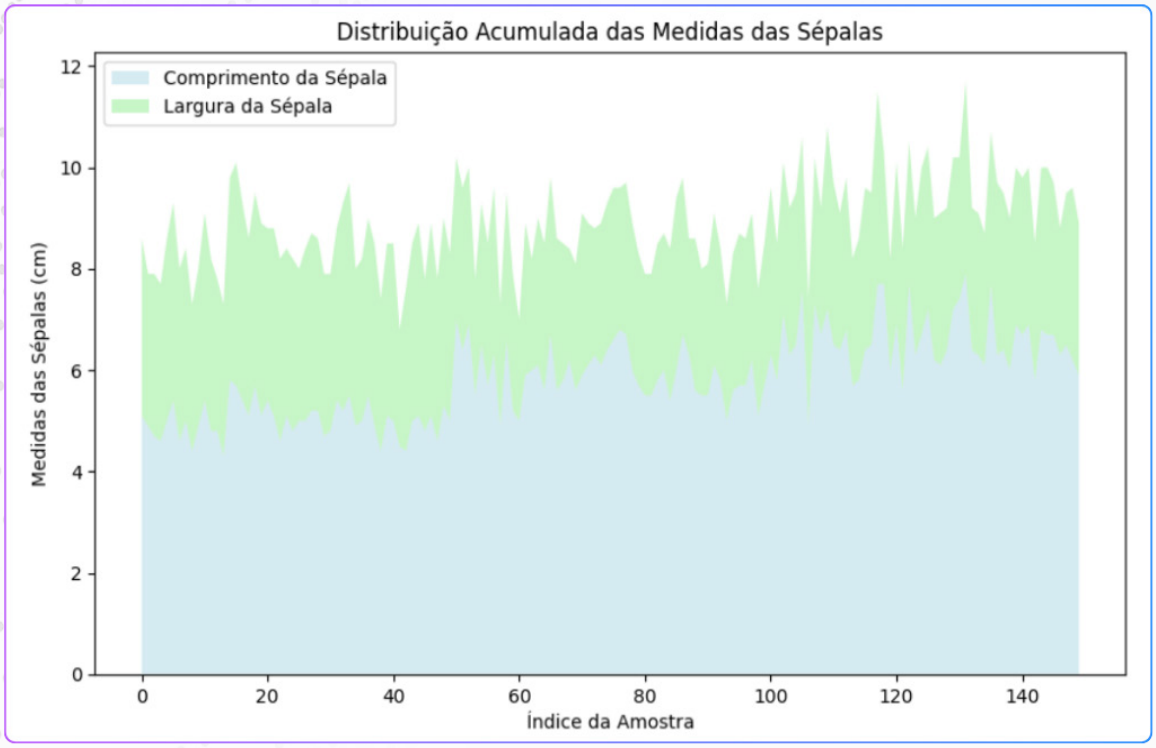
```
# Preparando os dados para o gráfico de área
x = range(len(iris))  # Índice para cada amostra

# Comprimento e largura da sépala, tratando possíveis dados faltantes
y_sepal_length = iris['sepal_length'].fillna(0)
y_sepal_width = iris['sepal_width'].fillna(0)

# Criando um gráfico de área
```

continua

```
plt.figure(figsize=(10, 6))
plt.stackplot(x, y_sepal_length, y_sepal_width, labels=['Comprimento da Sépala',
'Largura da Sépala'], colors=['lightblue', 'lightgreen'], alpha=0.5)
plt.title('Distribuição Acumulada das Medidas das Sépalas')
plt.xlabel('Índice da Amostra')
plt.ylabel('Medidas das Sépalas (cm)')
plt.legend(loc='upper left')
plt.show()
```

Fonte: autoria própria.

Figura 12 - Gráfico de área com a distribuição acumulada das medidas das sépalas



Fonte: autoria própria.

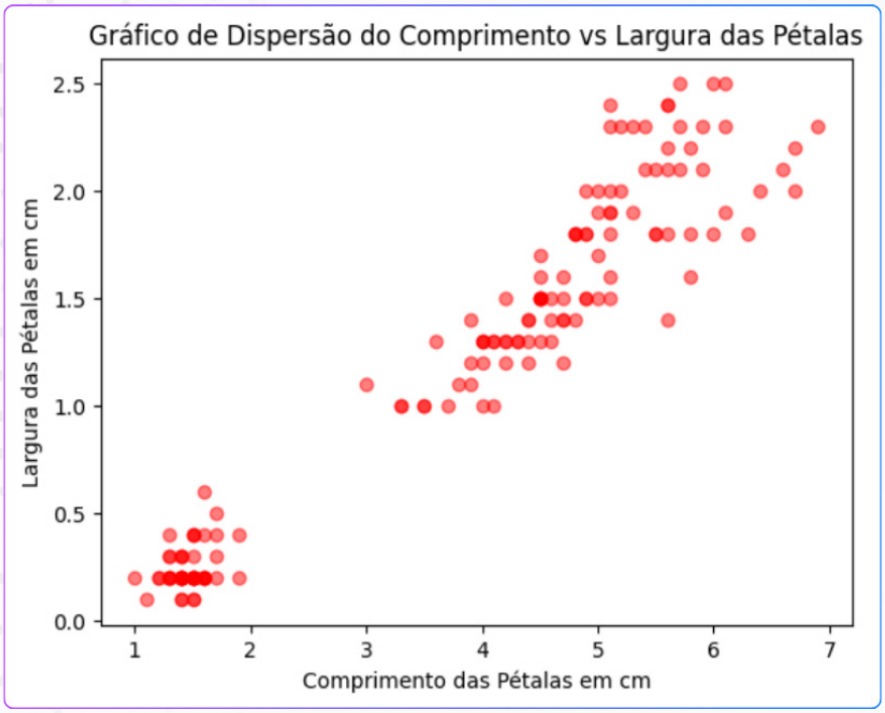
Gráficos de dispersão são importantes para visualizar a relação entre duas variáveis numéricas. O método plt.scatter() é usado para isso, como apresentado no Código 61, gerando o gráfico da Figura 10. Nesse exemplo, iris['petal_length'] possui os valores para o eixo X e iris['petal_width'] os valores para o eixo Y, c='red' configura a cor a ser utilizada (vermelho) e alpha=0.5, o grau de transparência.

Código 61 - Criando um gráfico de dispersão do comprimento vs. largura das pétalas

```
plt.scatter(iris['petal_length'], iris['petal_width'], c='red', alpha=0.5)
plt.title('Gráfico de Dispersão do Comprimento vs Largura das Pétalas')
plt.xlabel('Comprimento das Pétalas em cm')
plt.ylabel('Largura das Pétalas em cm')
plt.show()
```

Fonte: autoria própria.

Figura 13 - Gráfico de dispersão do comprimento *vs.* largura das pétalas



Fonte: autoria própria.

Os gráficos de pizza são úteis para mostrar proporções. Como apresentado no Código 62, plt.pie() é o método para criar esses gráficos, e você pode especificar etiquetas, tamanhos e cores, como no gráfico da Figura 11. No exemplo do Código 62, species_counts representa os valores, species_counts.index os rótulos, e autopct='%1.1f%%' as porcentagens em cada segmento. A opção startangle=90 ajusta o ângulo inicial do gráfico para facilitar a visualização e as cores são escolhidas manualmente para cada segmento, melhorando a clareza e a distinção entre as categorias.

Código 62 - Criando um gráfico de pizza com a distribuição das espécies de íris

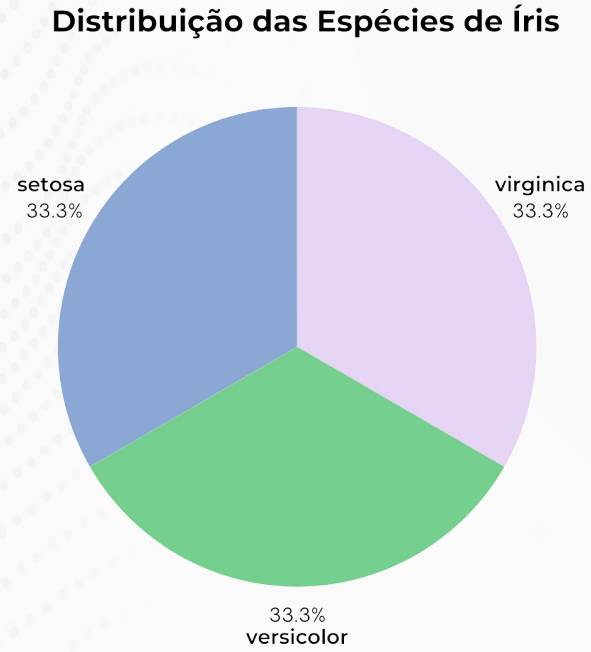
```
# Contando quantas amostras existem de cada espécie
species_counts = iris['species'].value_counts()


# Criando um gráfico de pizza
```

continua

```
plt.figure(figsize=(8, 8))
plt.pie(species_counts, labels=species_counts.index, autopct='%1.1f%%',
startangle=90, colors=['lightblue', 'lightgreen', 'lavender'])
plt.title('Distribuição das Espécies de Íris')
plt.show()
```

Fonte: autoria própria.

Figura 14 - Gráfico de pizza com distribuição das espécies de íris



Fonte: autoria própria.

Para criar visualizações mais complexas que necessitam da combinação de múltiplos gráficos em uma única figura, o *Matplotlib* oferece uma funcionalidade extremamente útil chamada *subplots*. O uso de *subplots* permite aos(às) usuários(as) de dados apresentar várias perspectivas dos dados simultaneamente, facilitando comparações diretas e análises mais ricas.

O método plt.subplots() cria uma figura e um conjunto de *subplots*. Esse método é flexível e pode ser configurado para criar várias linhas e colunas de gráficos numa única figura. No Código 63, há um exemplo de como usar *subplots* para combinar um gráfico de linhas e um histograma.

Código 63 - Exemplo de gráficos múltiplos usando *subplots*

```
fig, axs = plt.subplots(1, 2, figsize=(14, 7))
axs[0].plot(iris['sepal_length'], iris['sepal_width'], 'b^')
axs[1].hist(iris['sepal_width'], bins=15, color='r', alpha=0.7)
```

continua

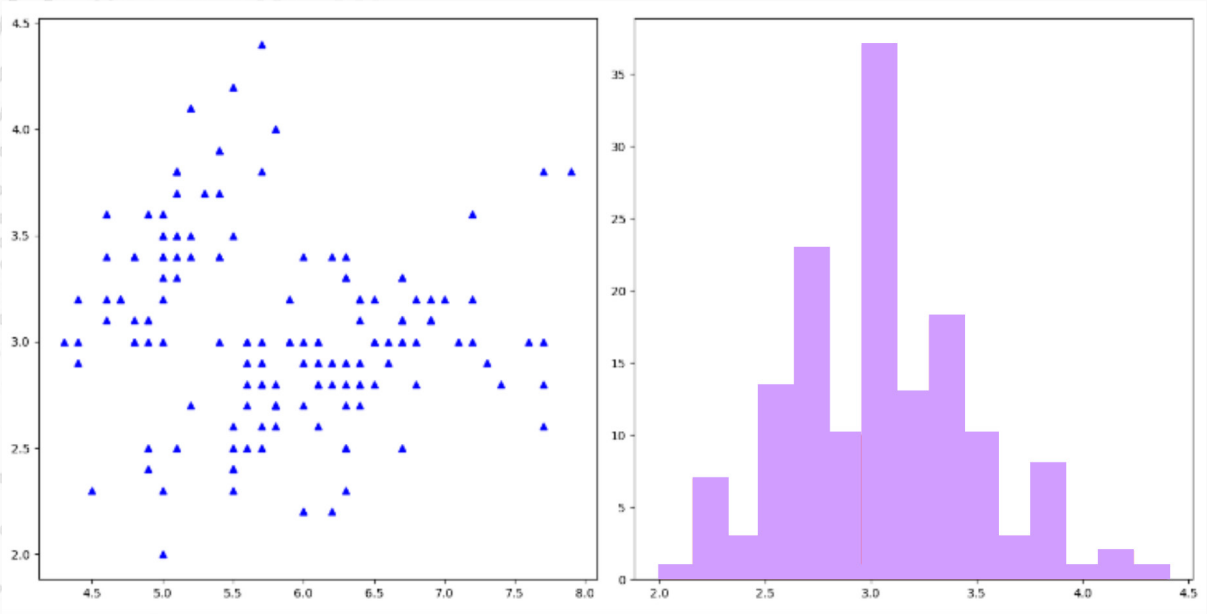
```
plt.tight_layout()
plt.show()
```

Fonte: autoria própria.

No exemplo do Código 63, resulta a Figura 12:

- `fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 7))` cria uma figura e dois *subplots* lado a lado;
- `ax1` e `ax2` são usados para criar gráficos específicos em cada *subplot*;
- `plt.tight_layout()` é uma chamada útil que ajusta automaticamente os parâmetros do *subplot* para que se encaixem bem na figura.

Figura 15 - Exemplo de gráficos múltiplos usando *subplots*



Fonte: autoria própria.

Utilizar *subplots* permite uma análise visual detalhada de diferentes aspectos dos dados, tornando-os uma ferramenta indispensável para apresentações de dados complexas e relatórios detalhados.

3.3 Visualizações com *Seaborn*

Como dito anteriormente, *Seaborn* é uma biblioteca de visualização de dados em *Python* que se baseia no *Matplotlib* e fornece uma interface de alto nível para desenhar gráficos estatísticos atraentes e informativos. Ele é excepcionalmente útil para visualizações estatísticas que exploram relações e distribuições entre variáveis numéricas.

Com ferramentas como `pairplot` (Código 64 e Figura 13), que cria matrizes de gráficos de dispersão, e `jointplot` (Código 65 e Figura 14), que combina gráficos de dispersão e histogramas para detalhar a relação entre duas variáveis, *Seaborn* simplifica a visualização de múltiplas dimensões dos dados simultaneamente. O `kdeplot` (Código 66 e Figura 15) ou gráfico de densidade kernel oferece uma maneira suave de visualizar a distribuição de dados, destacando áreas de maior densidade e tendências nos dados.

Como é possível observar nos exemplos, os métodos para construção de gráficos apresentados utilizam uma assinatura semelhante, tendo vários parâmetros em comum. A seguir, está uma lista dos mais utilizados, para referência:

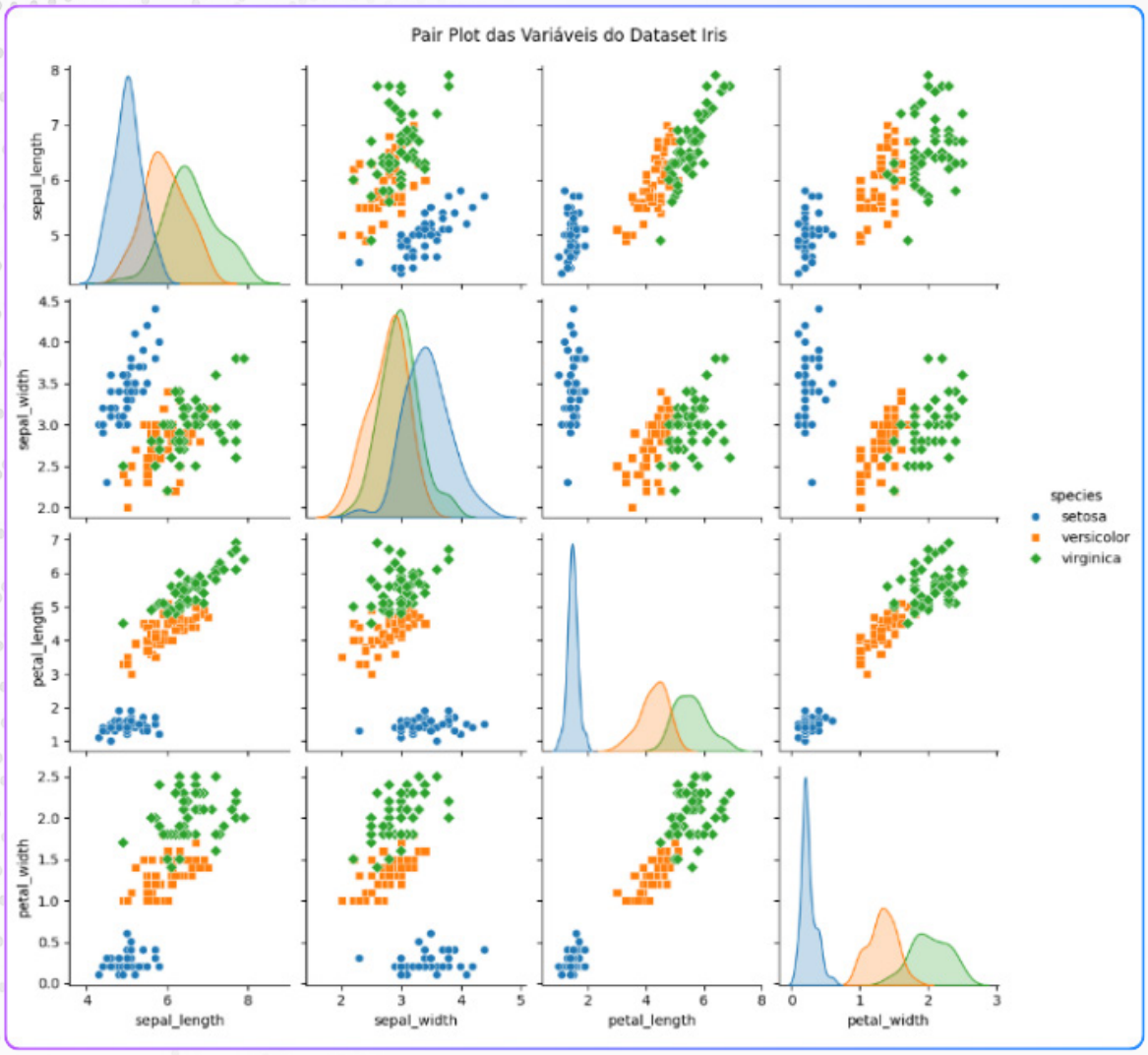
- ***data***: o *DataFrame* onde estão os dados. Costumeiramente, é também o primeiro parâmetro do método, como observado no Código 64, em que é passado o *DataFrame* **iris**.
- ***hue***: nome de uma coluna no *DataFrame* para mapear aspectos do gráfico para cores diferentes.
- ***markers***: o marcador a ser usado para todos os pontos do gráfico de dispersão ou uma lista de marcadores com comprimento igual ao número de níveis no parâmetro `hue`, para que pontos de cores diferentes também tenham marcadores de gráfico de dispersão diferentes.
- ***color***: especificação de cor única para quando o mapeamento de matiz (`hue`) não é usado. Caso contrário, o gráfico tentará utilizar o ciclo de cores padrão do *Matplotlib*.
- **x** e **y**: parâmetros que especificam posições nos eixos x e y do gráfico, respectivamente.
- ***kind***: O tipo de gráfico a ser utilizado em uma imagem. Usado em *plots* que combinam múltiplos tipos de gráficos, como o `jointplot`.

Código 64 - Pair *plot* das variáveis do *dataset* Iris

```
sns.pairplot(iris, hue='species', markers=["o", "s", "D"])
plt.suptitle('Pair Plot das Variáveis do Dataset Iris', y=1.02)
plt.show()
```

Fonte: autoria própria.

Figura 16 - *Pair plot* das variáveis do *dataset* íris



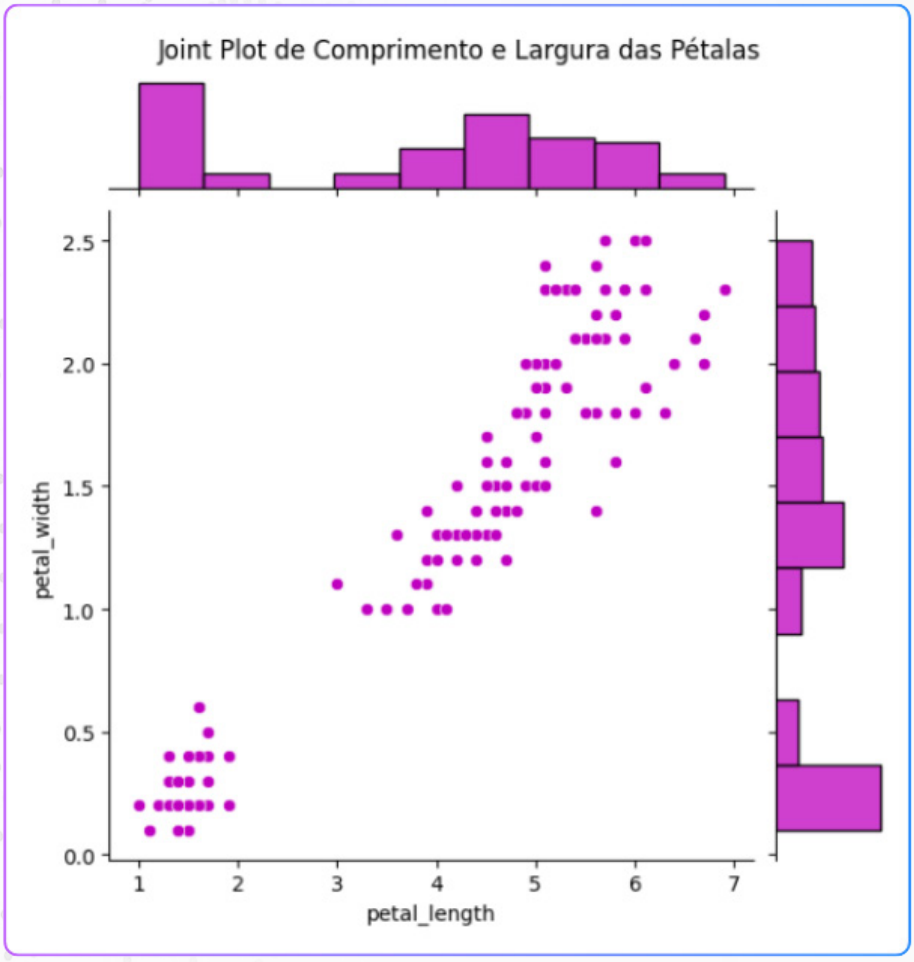
Fonte: autoria própria.

Código 65 - Criando um *joint plot* entre comprimento e largura das pétalas

```
sns.jointplot(x='petal_length', y='petal_width', data=iris, kind='scatter', color='m')
plt.suptitle('Joint Plot de Comprimento e Largura das Pétalas', y=1.02)
plt.show()
```

Fonte: autoria própria.

Figura 17 - *Joint plot* de comprimento e largura das pétalas

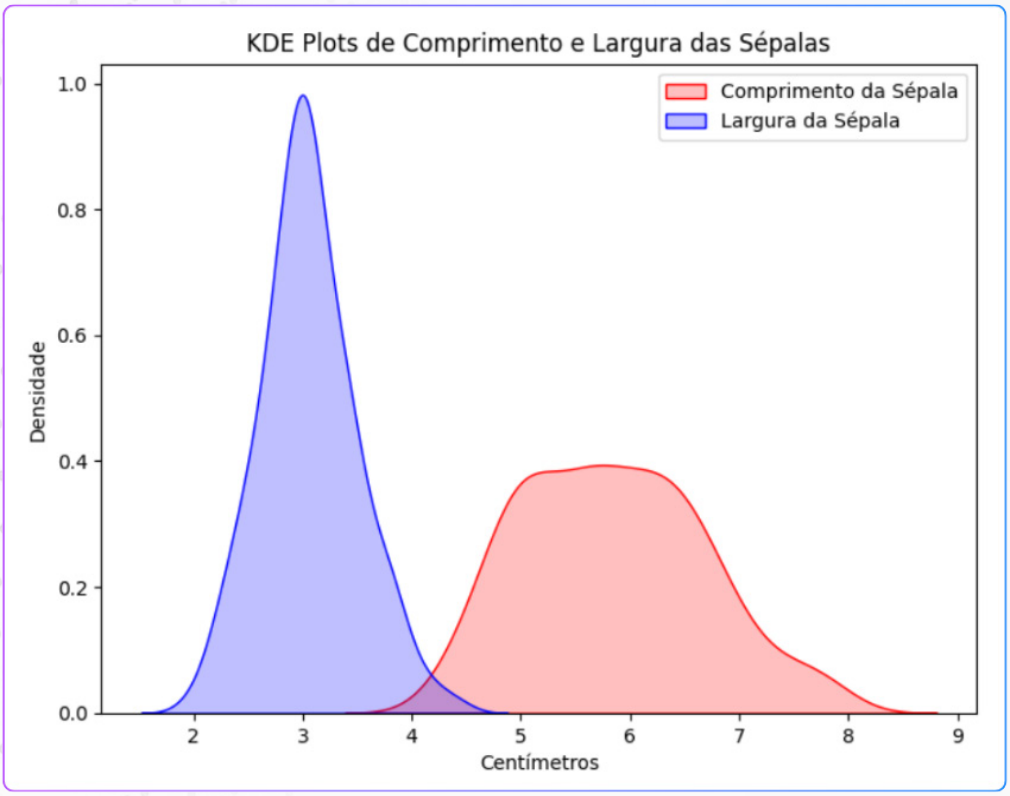


Fonte: autoria própria.

Código 66 - Criando *KDE plots* para o comprimento das sépalas

```
plt.figure(figsize=(8, 6))
sns.kdeplot(iris['sepal_length'], fill=True, color="r", label="Comprimento da Sépala")
sns.kdeplot(iris['sepal_width'], fill=True, color="b", label="Largura da Sépala")
plt.title('KDE Plots de Comprimento e Largura das Sépalas')
plt.xlabel('Centímetros')
plt.ylabel('Densidade')
plt.legend()
plt.show()
```

Figura 18 - *KDE plots* de comprimento e largura das sépalas



Fonte: autoria própria.

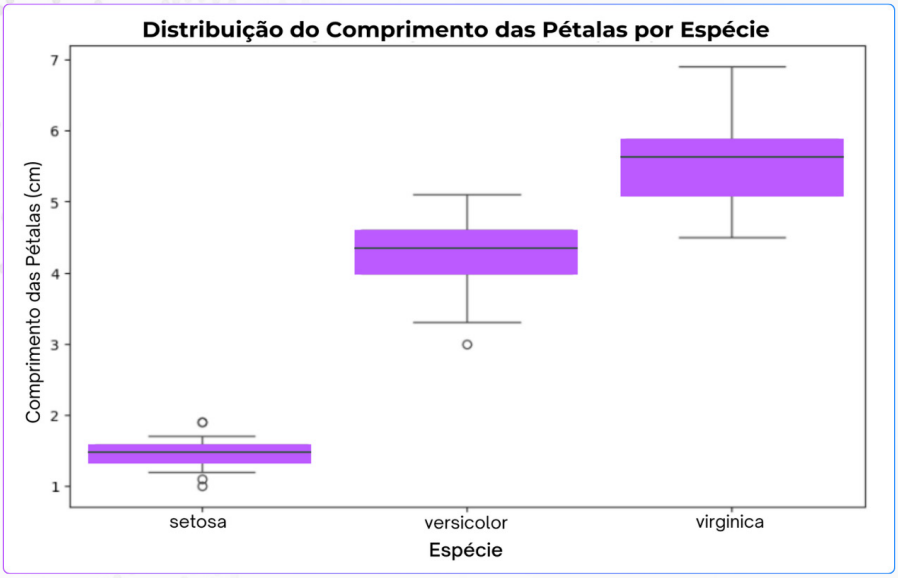
Para dados categóricos, *Seaborn* oferece gráficos como `boxplot` (Código 67 e Figura 16) e `violinplot` (Código 68 e Figura 17), eficazes para comparar distribuições e identificar *outliers* (elementos que estão a uma distância anormal de outros valores no conjunto de dados) em diferentes categorias. Os gráficos de barras (Código 69 e Figura 18) também são facilmente configuráveis e permitem visualizar as contagens ou medidas estatísticas associadas a diferentes categorias. Esses tipos de visualizações são essenciais para analisar como as características se comportam entre grupos distintos, revelando padrões ou anomalias que podem não ser aparentes em visualizações mais tradicionais.

Código 67 - Criando um *box plot* para visualizar a distribuição do comprimento das pétalas por espécie

```
plt.figure(figsize=(10, 6))
sns.boxplot(x='species', y='petal_length', data=iris)
plt.title('Distribuição do Comprimento das Pétalas por Espécie')
plt.xlabel('Espécie')
plt.ylabel('Comprimento das Pétalas (cm)')
plt.show()
```

Fonte: autoria própria.

Figura 19 - *Boxplot* com a distribuição do comprimento das pétalas por espécie



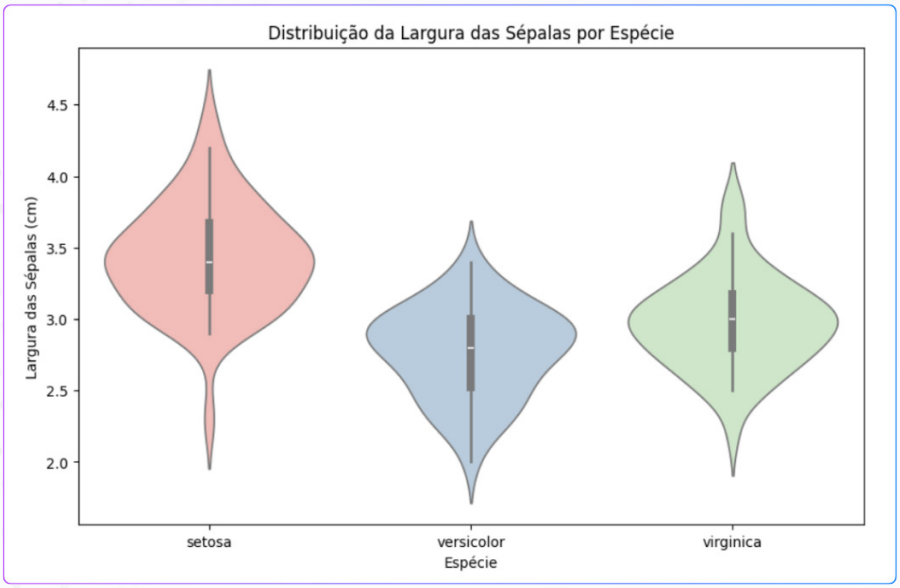
Fonte: autoria própria.

Código 68 - Criando um *violin plot* para visualizar a distribuição da largura das sépalas por espécie

```
plt.figure(figsize=(10, 6))
sns.violinplot(x='species', y='sepal_width', data=iris,
palette='Pastel1' , hue='species', legend=False)
plt.title('Distribuição da Largura das Sépalas por Espécie')
plt.xlabel('Espécie')
plt.ylabel('Largura das Sépalas (cm)')
plt.show()
```

Fonte: autoria própria.

Figura 20 - *Violin plot* com a distribuição da largura das sépalas por espécie



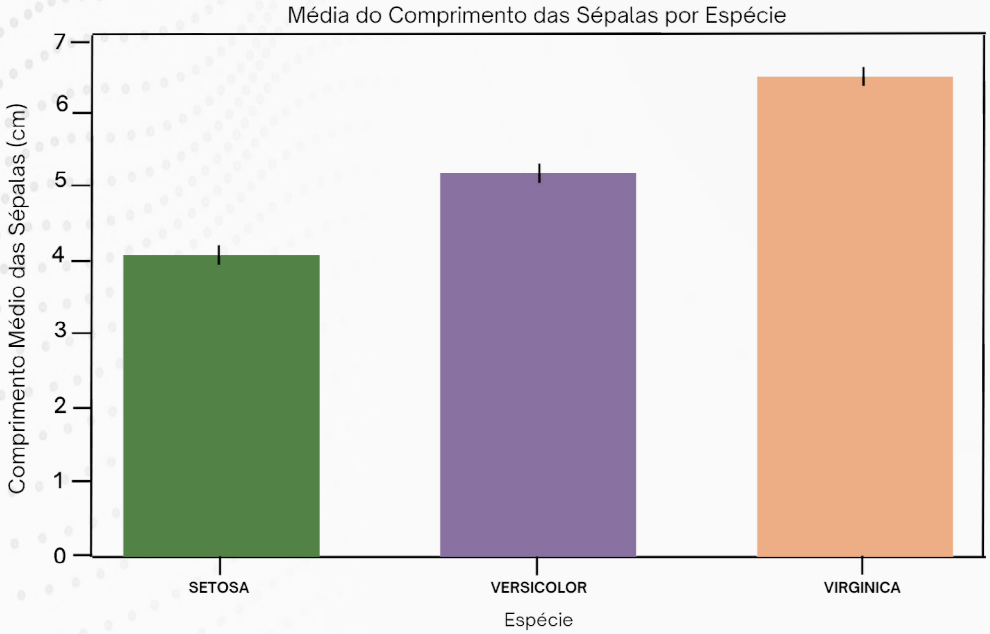
Fonte: autoria própria.

Código 69 - Criando um *bar plot* para comparar a média do comprimento das sépalas entre as espécies

```
plt.figure(figsize=(10, 6))
sns.barplot(x='species', y='sepal_length', data=iris, palette='Accent')
plt.title('Média do Comprimento das Sépalas por Espécie')
plt.xlabel('Espécie')
plt.ylabel('Comprimento Médio das Sépalas (cm)')
plt.show()
```

Fonte: autoria própria.

Figura 21 - Gráfico de barras do *Seaborn* com a média do comprimento das sépalas por espécie



Fonte: autoria própria.

A personalização é uma das principais vantagens do *Seaborn*. A biblioteca permite ajustar detalhadamente o estilo, cores e *layout* dos gráficos para melhor se adequar ao contexto de apresentação dos dados. Comandos como `set_style` e `set_palette` permitem modificar globalmente o estilo visual dos gráficos para se alinharem com a estética desejada. A forma de uso é apresentada no Código 70. Na documentação do *Seaborn* é possível obter a lista de estilos[8] (`set_style`) e paletas de cores[9] (`set_palette`) disponíveis. Essas personalizações não só aumentam a legibilidade dos gráficos como também reforçam a narrativa visual dos dados apresentados.

8 Documentação oficial do método *set_style*: https://seaborn.pydata.org/generated/seaborn.set_style.html.

9 Documentação oficial do *Seaborn* sobre paletas de cores: https://seaborn.pydata.org/tutorial/color_palettes.html.

Código 70 - Definindo estilo e paleta de cores no *Seaborn*

```
# Definindo o estilo do gráfico
sns.set_style("whitegrid")

# Personalizando a paleta de cores
sns.set_palette("husl")

# Definição do Gráfico ...
```

Fonte: autoria própria.

3.4 Notebook Colab

Fundamentos da Visualização de Dados

Esta seção destaca a importância crucial da visualização na análise de dados, servindo como uma ferramenta essencial para transformar conjuntos de dados complexos em representações visuais claras e informativas. Visualizações eficazes permitem aos/às analistas e partes interessadas descobrir padrões, identificar tendências e comunicar resultados de maneira eficaz, apoiando a tomada de decisões rápida e informada. Além disso, introduz Matplotlib e Seaborn como as bibliotecas predominantes no ecossistema Python para visualização de dados, com Matplotlib oferecendo versatilidade e capacidade de customização extensiva e Seaborn proporcionando uma interface mais acessível e esteticamente agradável para criar visualizações estatísticas complexas com menos esforço.

O *Matplotlib* e o *Seaborn* já vêm instalados nesse *Google Colab*. Mas, para instalá-los em seu ambiente local, você pode usar os comandos a seguir:

```
!pip install matplotlib
!pip install seaborn
```

```
Requirement already satisfied: matplotlib in /usr/local/lib/python3.10/dist-packages
(3.7.1)
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.10/dist-
packages (from matplotlib) (1.2.1)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.10/dist-packages
(from matplotlib) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.10/dist-
packages (from matplotlib) (4.53.1)
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.10/dist-
packages (from matplotlib) (1.4.5)
Requirement already satisfied: numpy>=1.20 in /usr/local/lib/python3.10/dist-packages
(from matplotlib) (1.26.4)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.10/dist-
```

```
packages (from matplotlib) (24.1)
Requirement already satisfied: pillow>=6.2.0 in /usr/local/lib/python3.10/dist-packages
(from matplotlib) (9.4.0)
Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.10/dist-
packages (from matplotlib) (3.1.2)
Requirement already satisfied: python-dateutil>=2.7 in /usr/local/lib/python3.10/dist-
packages (from matplotlib) (2.8.2)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from
python-dateutil>=2.7->matplotlib) (1.16.0)
Requirement already satisfied: seaborn in /usr/local/lib/python3.10/dist-packages
(0.13.1)
Requirement already satisfied: numpy!=1.24.0,>=1.20 in /usr/local/lib/python3.10/dist-
packages (from seaborn) (1.26.4)
Requirement already satisfied: pandas>=1.2 in /usr/local/lib/python3.10/dist-packages
(from seaborn) (2.1.4)
Requirement already satisfied: matplotlib!=3.6.1,>=3.4 in /usr/local/lib/python3.10/dist-
packages (from seaborn) (3.7.1)
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.10/dist-
packages (from matplotlib!=3.6.1,>=3.4->seaborn) (1.2.1)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.10/dist-packages
(from matplotlib!=3.6.1,>=3.4->seaborn) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.10/dist-
packages (from matplotlib!=3.6.1,>=3.4->seaborn) (4.53.1)
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.10/dist-
packages (from matplotlib!=3.6.1,>=3.4->seaborn) (1.4.5)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.10/dist-
packages (from matplotlib!=3.6.1,>=3.4->seaborn) (24.1)
Requirement already satisfied: pillow>=6.2.0 in /usr/local/lib/python3.10/dist-packages
(from matplotlib!=3.6.1,>=3.4->seaborn) (9.4.0)
Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.10/dist-
packages (from matplotlib!=3.6.1,>=3.4->seaborn) (3.1.2)
Requirement already satisfied: python-dateutil>=2.7 in /usr/local/lib/python3.10/dist-
packages (from matplotlib!=3.6.1,>=3.4->seaborn) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-packages
(from pandas>=1.2->seaborn) (2024.1)
Requirement already satisfied: tzdata>=2022.1 in /usr/local/lib/python3.10/dist-
packages (from pandas>=1.2->seaborn) (2024.1)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from
python-dateutil>=2.7->matplotlib!=3.6.1,>=3.4->seaborn) (1.16.0)
```

Para verificar se estão instalados corretamente, basta executar o código a seguir:

```
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib as mpl
import seaborn as sns

print(f"Matplotlib version: {mpl.__version__}")
print(f"Seaborn version: {sns.__version__}")
```

```
Matplotlib version: 3.7.1
Seaborn version: 0.13.1
```

A base de dados a ser utiltizada nos exemplos a seguir é o *dataset* Iris,descrita na Unidade III do Ebook.

```
# Carregando o dataset do Iris diretamente do GitHub
url = 'https://raw.githubusercontent.com/mwaskom/seaborn-data/master/iris.csv'
iris = pd.read_csv(url)
iris
```

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa
...	...	...	...	...	...
145	6.7	3.0	5.2	2.3	virginica
146	6.3	2.5	5.0	1.9	virginica
147	6.5	3.0	5.2	2.0	virginica
148	6.2	3.4	5.4	2.3	virginica
149	5.9	3.0	5.1	1.8	virginica

150 rows × 5 columns

Visualizações com *Matplotlib*

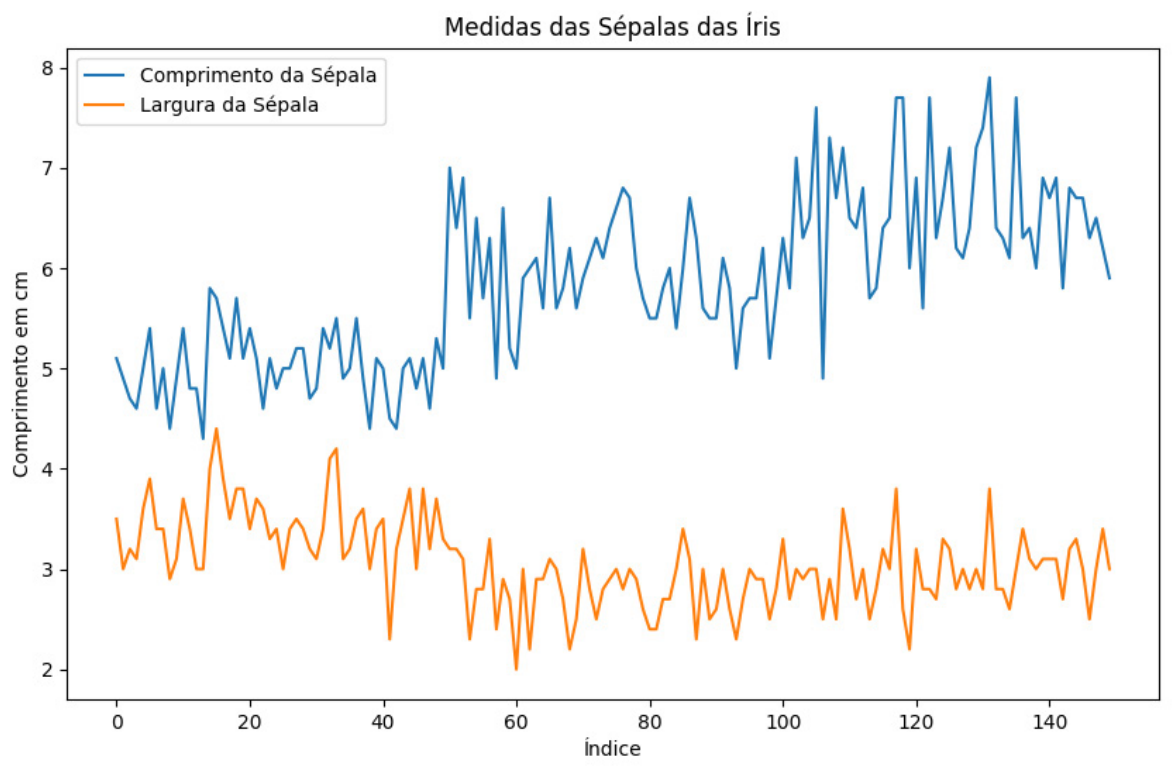
Nesta seção, é explorado o uso do *Matplotlib* para criar uma ampla variedade de visualizações de dados, desde gráficos básicos até representações mais complexas. *Matplotlib* simplifica a criação de gráficos de linhas, barras, histogramas, e gráficos de área, como é visto em exemplos práticos que ilustram essas técnicas com o *dataset* Iris. Por exemplo, é demonstrado como um gráfico de linhas pode ser utilizado para visualizar o comprimento e a largura das sépalas, enquanto os histogramas são usados para observar a distribuição de características como o comprimento das pétalas. Ressalta-se a flexibilidade do *Matplotlib* em ajustar detalhes visuais como legendas, títulos e estilos de linha, permitindo que os/as usuários/as personalizem os gráficos para atender às necessidades específicas de suas análises e apresentações de dados.

```
# Define o tamanho da figura (largura, altura) em polegadas
plt.figure(figsize=(10, 6))

# Adiciona o comprimento das sépalas ao gráfico (por padrão
```

continua

```
 será adicionada uma nova linha para cada plot)



plt.plot(iris['sepal_length'], label='Comprimento da Sépala')



# Adiciona a largura das sépalas ao gráfico
plt.plot(iris['sepal_width'], label='Largura da Sépala')

# Adiciona um título ao gráfico
plt.title('Medidas das Sépalas das Íris')

# Adiciona rótulos aos eixos x e y
plt.xlabel('Índice')
plt.ylabel('Comprimento em cm')

# Adiciona uma legenda ao gráfico
plt.legend()

# Exibe o gráfico
plt.show()
```

Figura 22 - Medidas das Sépalas das Íris



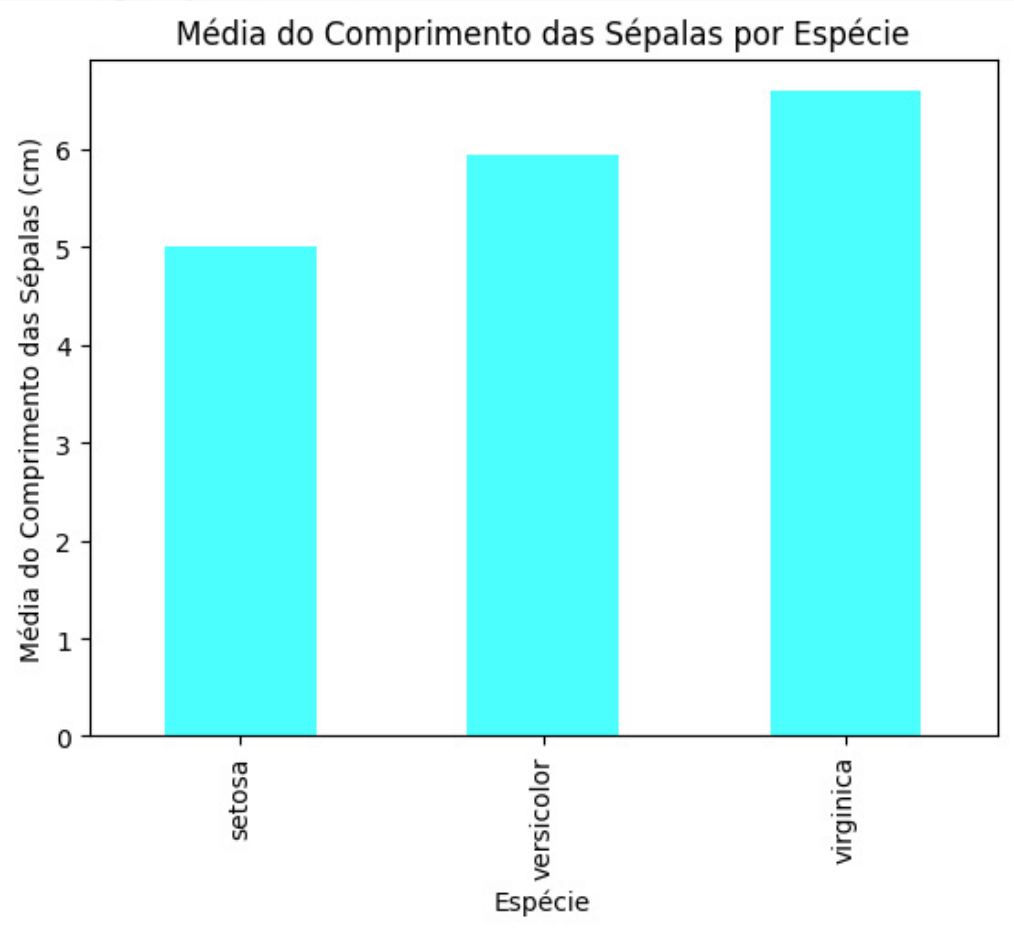
Fonte: autoria própria.

```
# Agrupando os dados por espécie (species) e calculando a média do comprimento das sépalas (sepal_length)
sepal_length_means = iris.groupby('species')['sepal_length'].mean()
sepal_length_means
```

```
species
setosa       5.006
versicolor   5.936
virginica    6.588
Name: sepal_length, dty
pe: float64
```

```
# Criando um gráfico de barras, utilizando a cor ciano (cyan) e transparência de 70% (alpha=0.7)
sepal_length_means.plot(kind='bar', color='cyan', alpha=0.7)

# Definindo os rótulos e título do gráfico
plt.title('Média do Comprimento das Sépalas por Espécie')
plt.xlabel('Espécie')
plt.ylabel('Média do Comprimento das Sépalas (cm)')
plt.show()
```

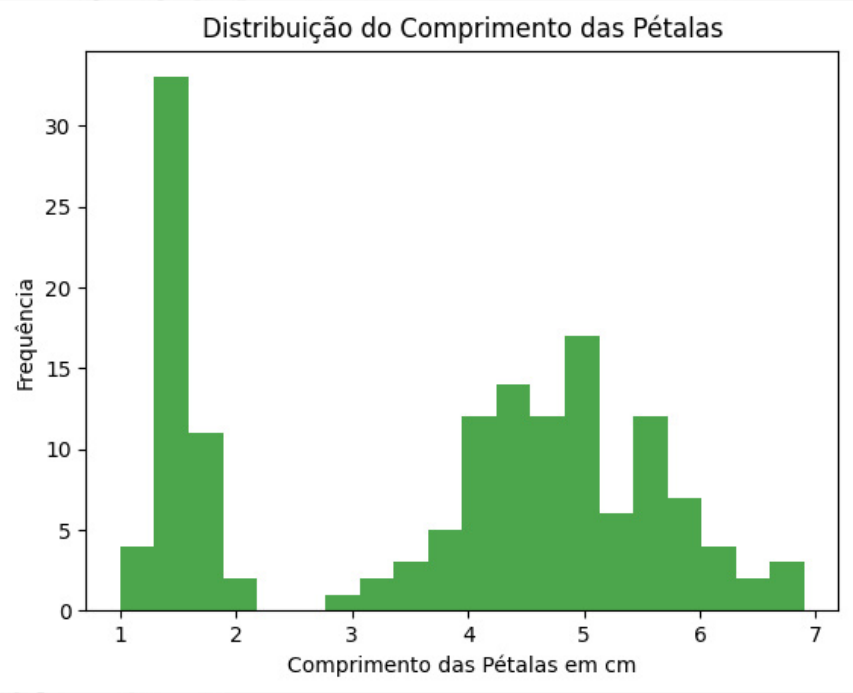
Figura 23 - Média do Comprimento das Sépalas por Espécie



Fonte: autoria própria.

```
# Define a cor, a transparência (alpha) e o número de barras (bins)
plt.hist(iris['petal_length'], bins=20, color='green', alpha=0.7)

plt.title('Distribuição do Comprimento das Pétalas')
plt.xlabel('Comprimento das Pétalas em cm')
plt.ylabel('Frequência')
plt.show()
```

Figura 24 - Distribuição do Comprimento das Pétalas



Fonte: autoria própria.

```
# Preparando os dados para o gráfico de área
x = range(len(iris))  # Índice para cada amostra
x
```

```
range(0, 150)
```

Alguns tipos de gráficos, como os gráficos de área, são melhor representados quando não há dados nulos ou faltantes na série. Para isso, pode-se substituir por um valor padrão, por meio do método fillna. O código a seguir substitui os dados faltantes nas colunas sepal_length e sepal_width.

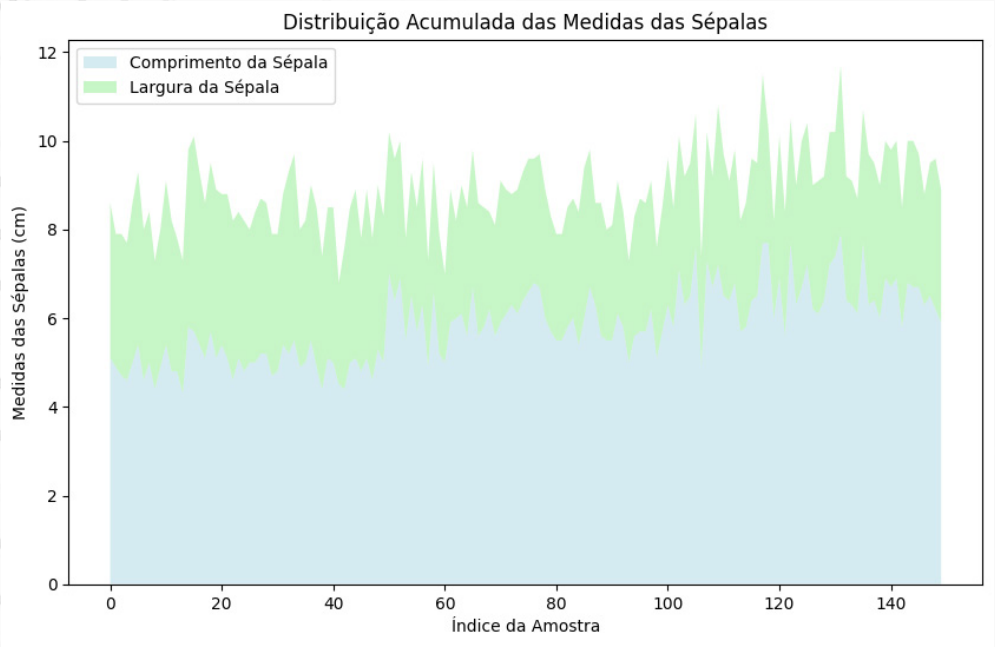
```
# Comprimento e largura da sépala, tratando possíveis dados faltantes
y_sepal_length = iris['sepal_length'].fillna(0)
y_sepal_width = iris['sepal_width'].fillna(0)
print(y_sepal_length)
print(y_sepal_width)
```

```
0    5.1
1    4.9
2    4.7
3    4.6
4    5.0
    ...
145  6.7
146  6.3
147  6.5
148  6.2
149  5.9
Name: sepal_length, Length: 150, dtype: float64
0    3.5
1    3.0
2    3.2
3    3.1
4    3.6
    ...
145  3.0
146  2.5
147  3.0
148  3.4
149  3.0
Name: sepal_width, Length: 150, dtype: float64
```

O código a seguir gera um gráfico de área (stackplot) com os dados já tratados em y_sepal_length e y_sepal_width. Os rótulos são definidos utilizando o parâmetro labels, bem como as cores por meio do parâmetro colors. Por fim, é definido um grau de transparência (alpha=0.5).

```
# Criando um gráfico de área
plt.figure(figsize=(10, 6))
plt.stackplot(x, y_sepal_length, y_sepal_width, labels=['Comprimento da Sépala', 'Largura da Sépala'], colors=['lightblue', 'lightgreen'], alpha=0.5)
plt.title('Distribuição Acumulada das Medidas das Sépalas')
plt.xlabel('Índice da Amostra')
plt.ylabel('Medidas das Sépalas (cm)')
plt.legend(loc='upper left')
plt.show()
```

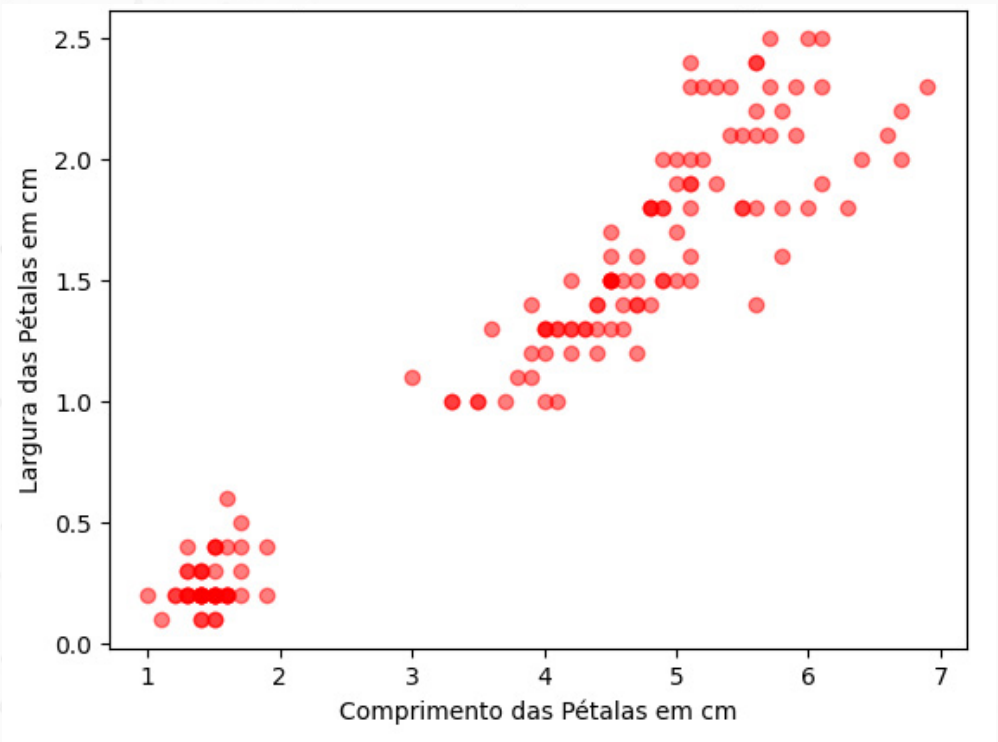
Figura 25 - Medidas das Sépalas das Íris



Fonte: autoria própria.

```
# Criando um gráfico de dispersão do Comprimento vs Largura das Pétalas
plt.scatter(iris['petal_length'], iris['petal_width'], c='red', alpha=0.5)
plt.title('Gráfico de Dispersão do Comprimento vs Largura das Pétalas')
plt.xlabel('Comprimento das Pétalas em cm')
plt.ylabel('Largura das Pétalas em cm')
plt.show()
```

Figura 26 - Gráfico de Dispersão do Comprimento vs Largura das Pétalas



Fonte: autoria própria.

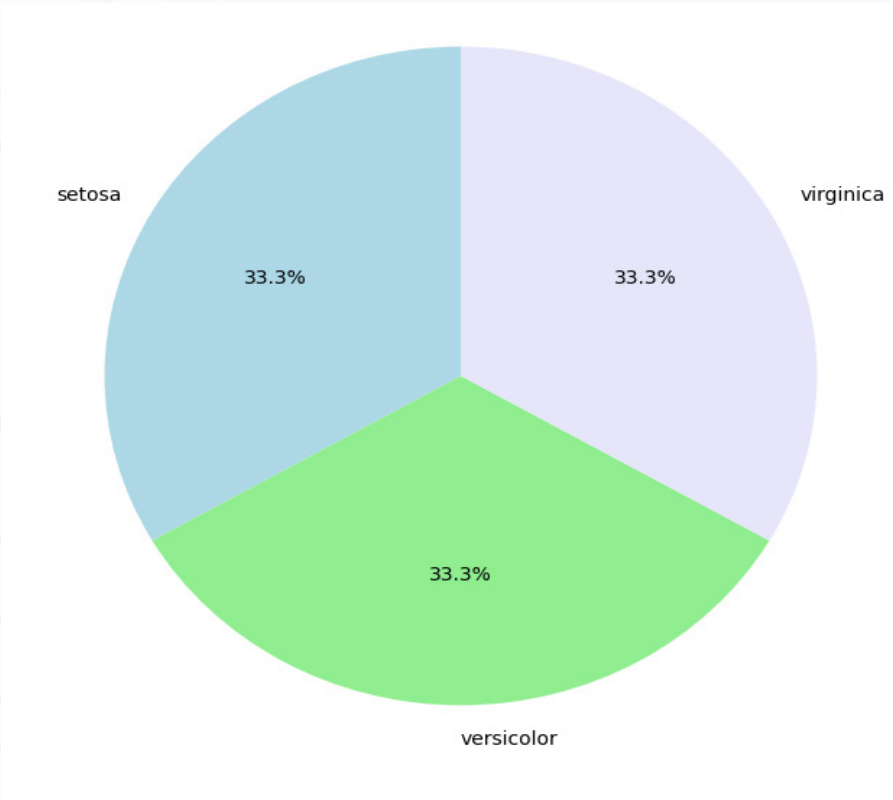
```
# Contando quantas amostras existem de cada espécie
species_counts = iris['species'].value_counts()
species_counts
```

```
species
setosa       50
versicolor   50
virginica    50
Name: count, dtype: int64
```

No exemplo a seguir, species_counts representa os valores, species_counts.index os rótulos, e autopct='%1.1f%%' as porcentagens em cada segmento. A opção startangle=90 ajusta o ângulo inicial do gráfico para facilitar a visualização, e as cores são escolhidas manualmente para cada segmento, melhorando a clareza e distinção entre as categorias.

```
# Criando um gráfico de pizza com a Distribuição das Espécies de Íris
plt.figure(figsize=(8, 8))
plt.pie(species_counts, labels=species_counts.index, autopct='%1.1f%%', startangle=90,
colors=['lightblue', 'lightgreen', 'lavender'])
plt.title('Distribuição das Espécies de Íris')
plt.show()
```

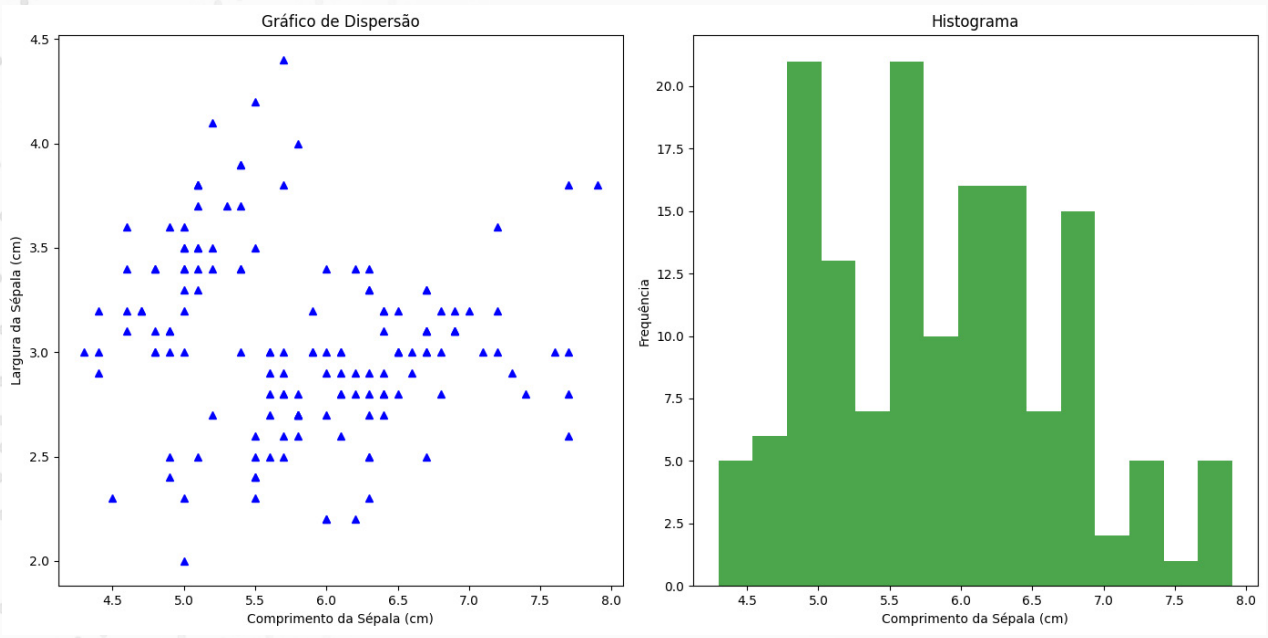
Figura 27 - Distribuição de Espécies de Íris



Fonte: autoria própria.

```
# Exemplo de gráficos múltiplos usando subplots

# Cria uma figura e dois subplots lado a lado
fig, axs = plt.subplots(1, 2, figsize=(14, 7))

# Gráfico de Dispersão com o comprimento das sépalas no eixo X e a largura no eixo Y
axs[0].plot(iris['sepal_length'], iris['sepal_width'], 'b^')
axs[0].set_xlabel('Comprimento da Sépala (cm)')
axs[0].set_ylabel('Largura da Sépala (cm)')
axs[0].set_title('Gráfico de Dispersão')

# Histograma do comprimento das sépalas
axs[1].hist(iris['sepal_length'], bins=15, color='g', alpha=0.7)
axs[1].set_xlabel('Comprimento da Sépala (cm)')

axs[1].set_ylabel('Frequência')
axs[1].set_title('Histograma')

# Ajusta o espaçamento entre os subplots
plt.tight_layout()
plt.show()
```

Figura 28 - Gráficos múltiplos usando *subplots*



Fonte: autoria própria.

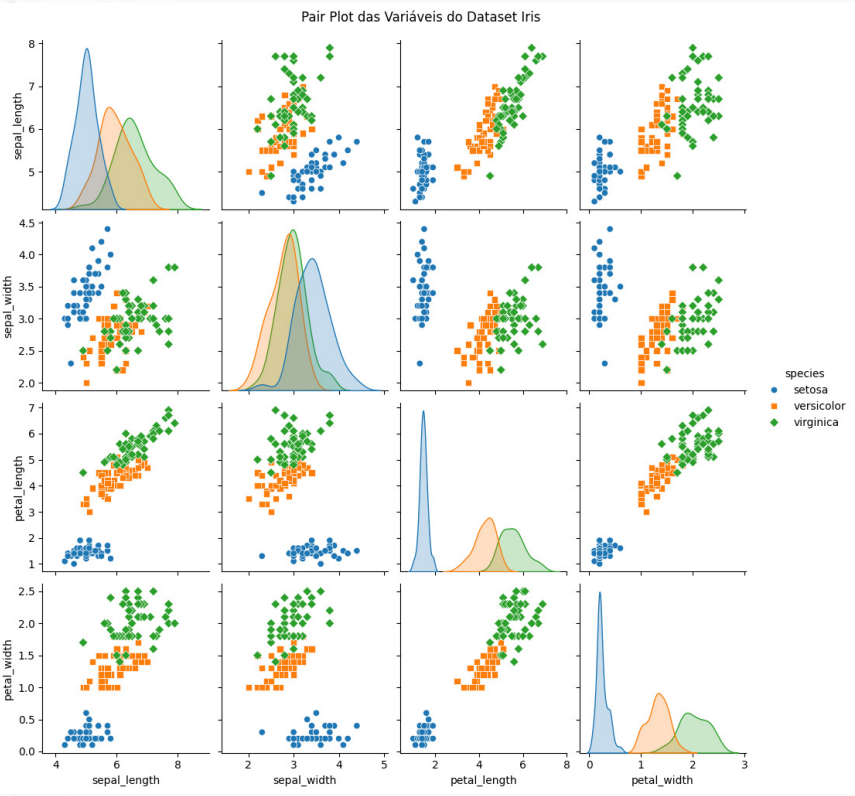
Visualizações com *Seaborn*

Esta seção aborda como *Seaborn*, uma biblioteca de visualização de dados construída sobre o *Matplotlib*, facilita a criação de gráficos estatísticos avançados e esteticamente agradáveis. *Seaborn* é destacado por sua interface de alto nível que permite aos/às

usuários/as gerar visualizações complexas, como *pair plots*, *joint plots* e *KDE plots*, com menos código e configurações padronizadas visualmente mais atraentes. A seção explora ferramentas específicas do *Seaborn* que são eficazes para analisar relações entre variáveis numéricas e categorias, como gráficos de dispersão que detalham as interações entre variáveis e gráficos de caixa e *violin plots* para comparar distribuições e identificar *outliers* entre categorias. Essas funcionalidades tornam *Seaborn* uma escolha poderosa para análises exploratórias de dados, proporcionando *insights* profundos por meio de visualizações que destacam padrões e tendências nos dados de maneira intuitiva e acessível.

```
# Criar um pair plot das variáveis do dataset Iris
# - 'iris' é o DataFrame que contém os dados do conjunto Iris
# - 'hue' define a variável para colorir os pontos (neste caso, a espécie das flores)
# - 'markers' define os símbolos usados para cada espécie: "o"
para setosa, "s" para versicolor, "D" para virginica

sns.pairplot(iris, hue='species', markers=["o", "s", "D"])

# Adicionar um título ao gráfico com ajuste na posição vertical (y) para evitar sobreposição
plt.suptitle('Pair Plot das Variáveis do Dataset Iris', y=1.02)

# Mostrar o gráfico gerado
plt.show()
```

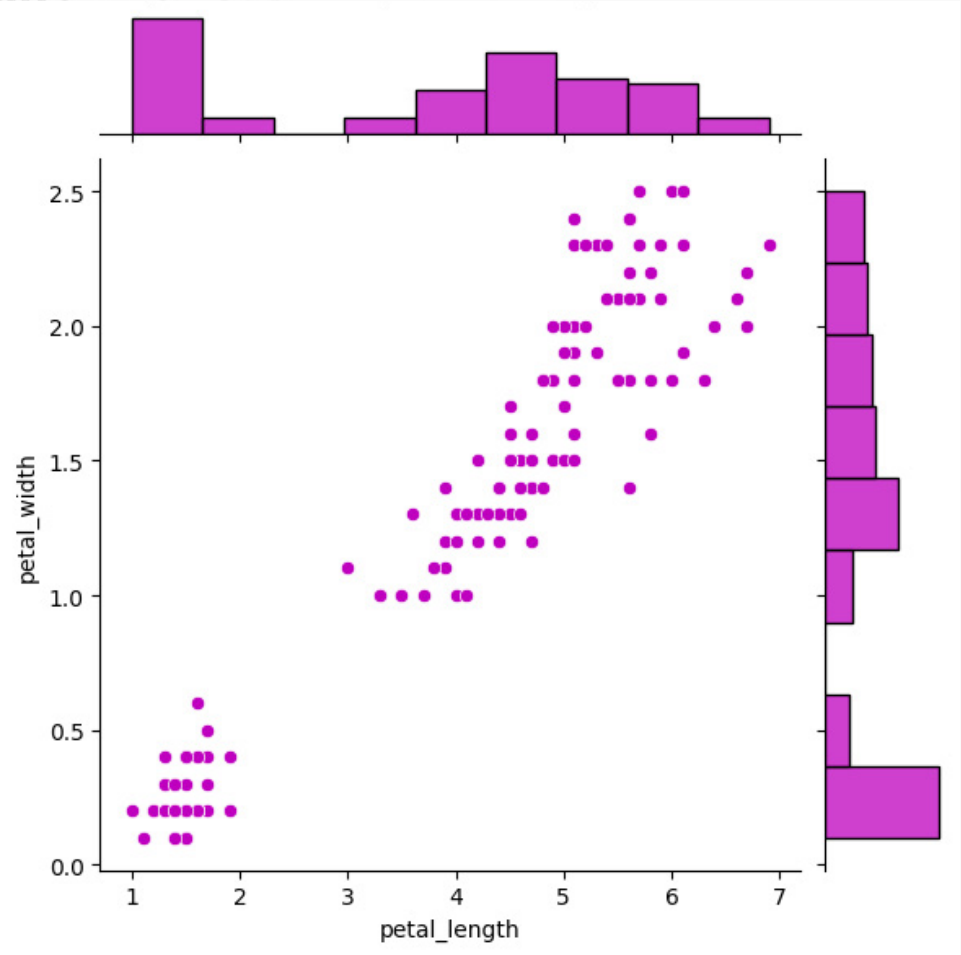
Figura 29 - *Pair plots* das variáveis do dataset Iris



Fonte: autoria própria.

```
# Cria um gráfico conjunto (joint plot) que mostra a relação entre o comprimento das pétalas e a largura das pétalas
# Define o eixo x como a coluna 'petal_length' do DataFrame
# Define o eixo y como a coluna 'petal_width' do DataFrame
# Especifica o DataFrame a ser usado (iris)
# Define o tipo de gráfico como 'scatter' (dispersão)
# Define a cor dos pontos como magenta ('m')
sns.jointplot(x='petal_length', y='petal_width', data=iris, kind='scatter', color='m')
plt.suptitle('Joint Plot de Comprimento e Largura das Pétalas', y=1.02)
plt.show()
```

Figura 30 - *Joint Plot* de Comprimento e Largura das Pétalas



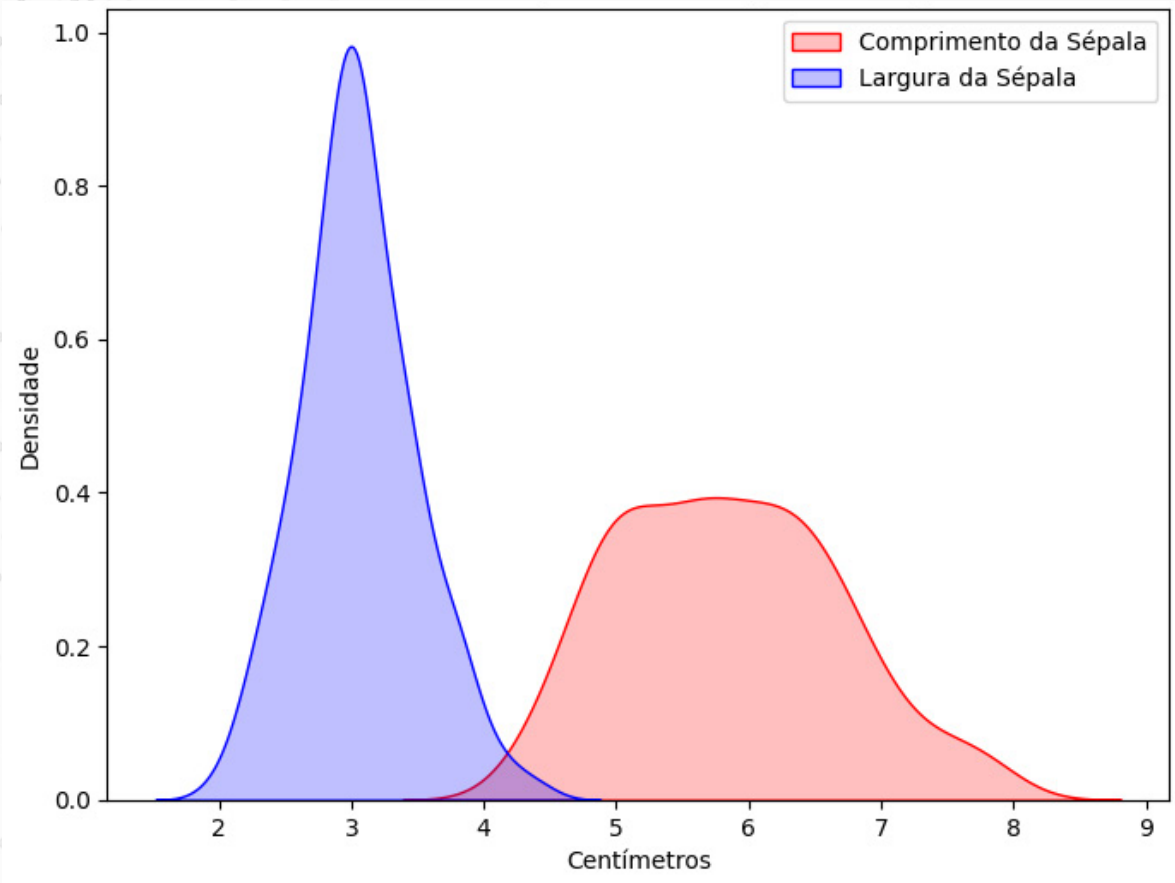
Fonte: autoria própria.

```
plt.figure(figsize=(8, 6))

# Cria um gráfico de densidade (KDE plot) para a distribuição do comprimento das sépalas
# Especifica a coluna 'sepal_length' do DataFrame iris como os dados a serem plotados
# Define a cor do gráfico como vermelho (r)
```

continua

```
# Preenche a área sob a curva
# Adiciona um rótulo ao gráfico para a legenda
sns.kdeplot(iris['sepal_length'], fill=True, color="r", label="Comprimento da Sépala")

# Cria um gráfico de densidade (KDE plot) para a distribuição da largura das sépalas
# Especifica a coluna 'sepal_width' do DataFrame iris como os dados a serem plotados
# Define a cor do gráfico como azul (b)
# Preenche a área sob a curva
# Adiciona um rótulo ao gráfico para a legenda
sns.kdeplot(iris['sepal_width'], fill=True, color="b", label="Largura da Sépala")



plt.title('KDE Plots de Comprimento e Largura das Sépalas')
plt.xlabel('Centímetros')
plt.ylabel('Densidade')
# Adiciona uma legenda ao gráfico
plt.legend()
plt.show()
```

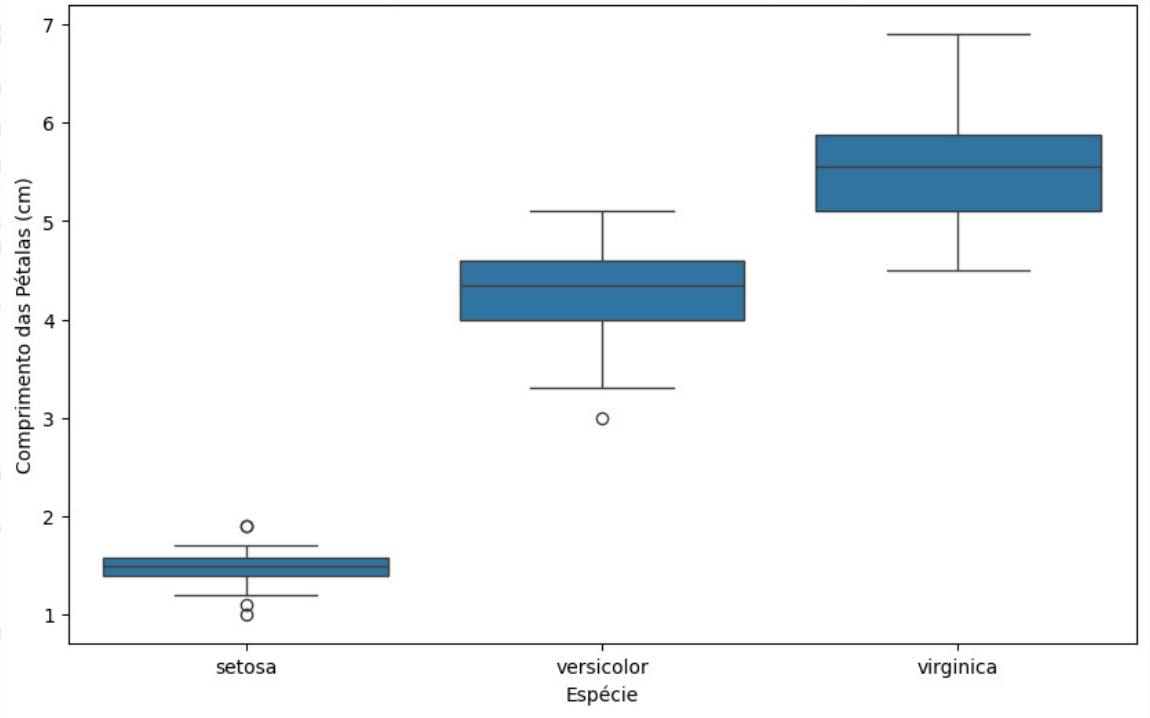
Figura 31 - KDE *Plots* de Comprimento e Largura das Sépalas



Fonte: autoria própria.

```
# Boxplot com a Distribuição do Comprimento das Pétalas por Espécie
plt.figure(figsize=(10, 6))

# Define o eixo x como a coluna 'species' do DataFrame
# Define o eixo y como a coluna 'petal_length' do DataFrame
# Especifica o DataFrame a ser usado (iris)
sns.boxplot(x='species', y='petal_length', data=iris)
plt.title('Distribuição do Comprimento das Pétalas por Espécie')
plt.xlabel('Espécie')
plt.ylabel('Comprimento das Pétalas (cm)')
plt.show()
```

Figura 32 - Distribuição do Comprimento das Pétalas por Espécie



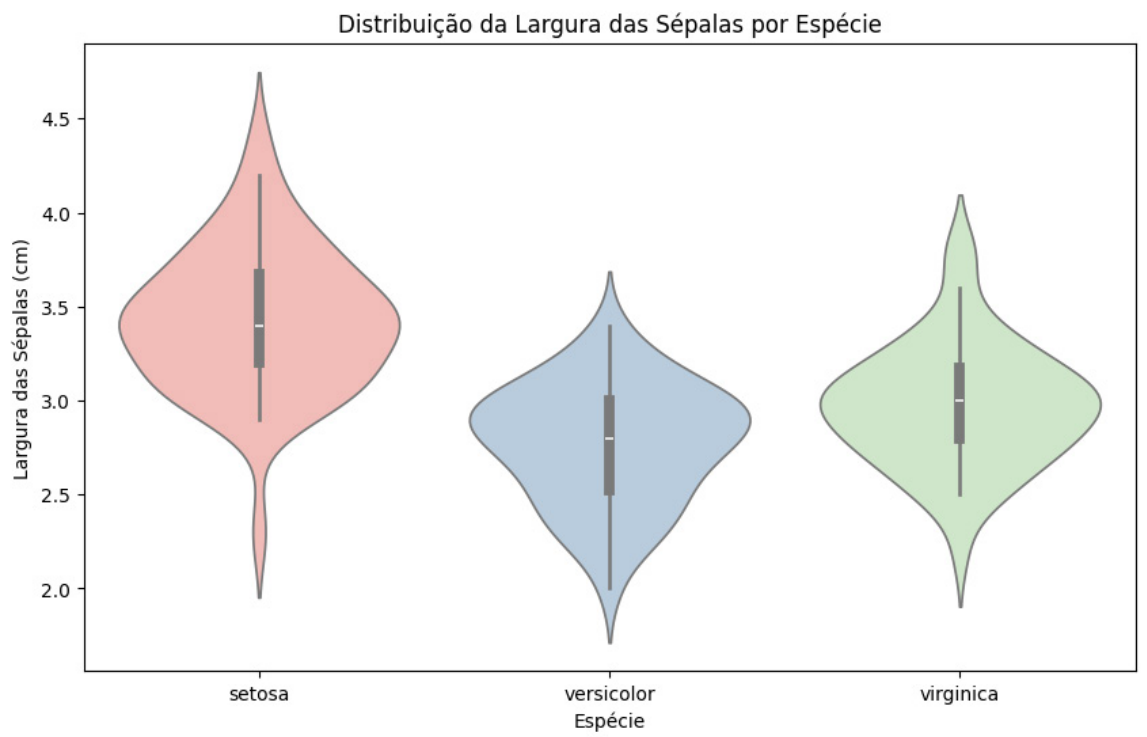
Fonte: autoria própria.

```
# Violin plot com a Distribuição da Largura das Sépalas por Espécie
plt.figure(figsize=(10, 6))

# Define o eixo x como a coluna 'species' do DataFrame
# Define o eixo y como a coluna 'sepal_width' do DataFrame
# Especifica o DataFrame a ser usado (iris)
# Define a paleta de cores a ser usada (Pastel1)
# Adiciona cor às violas com base na espécie (para melhor visualização)
# Remove a legenda
sns.violinplot(x='species', y='sepal_width', data=iris, palette='Pastel1', hue='species',
legend=False)
```

continua

```
# Adiciona um título e rótulos ao gráfico
plt.title('Distribuição da Largura das Sépalas por Espécie')
plt.xlabel('Espécie')
plt.ylabel('Largura das Sépalas (cm)')

# Mostra o gráfico gerado
plt.show()
```

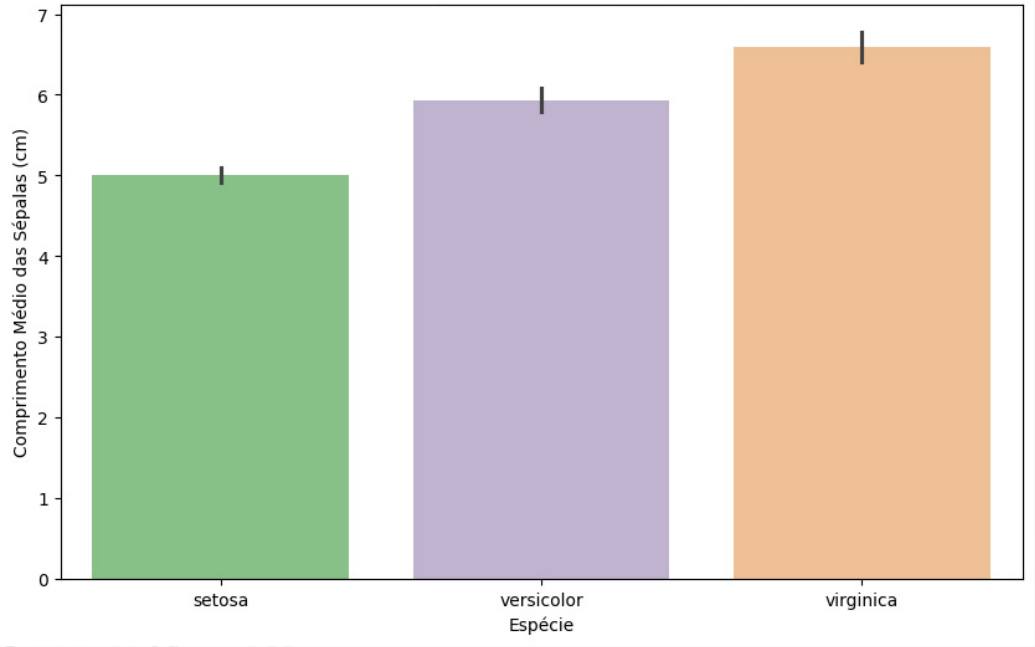
Figura 33 - Distribuição da Largura das Sépalas por Espécie



Fonte: autoria própria.

```
# Criando um bar plot para comparar a média do comprimento das sépalas entre as espécies
plt.figure(figsize=(10, 6))

# Define o eixo x como a coluna 'species' do DataFrame
# Define o eixo y como a coluna 'sepal_length' do DataFrame
# Especifica o DataFrame a ser usado (iris)
# Define a paleta de cores a ser usada (Accent)
# Adiciona cor às barras com base na espécie (para melhor visualização)
# Remove a legenda do gráfico
sns.barplot(x='species', y='sepal_length', data=iris, palette='Accent', hue='species', legend=False)
plt.title('Média do Comprimento das Sépalas por Espécie')
plt.xlabel('Espécie')
plt.ylabel('Comprimento Médio das Sépalas (cm)')
plt.show()
```

Figura 34 - Média do Comprimento das Sépalas por Espécie

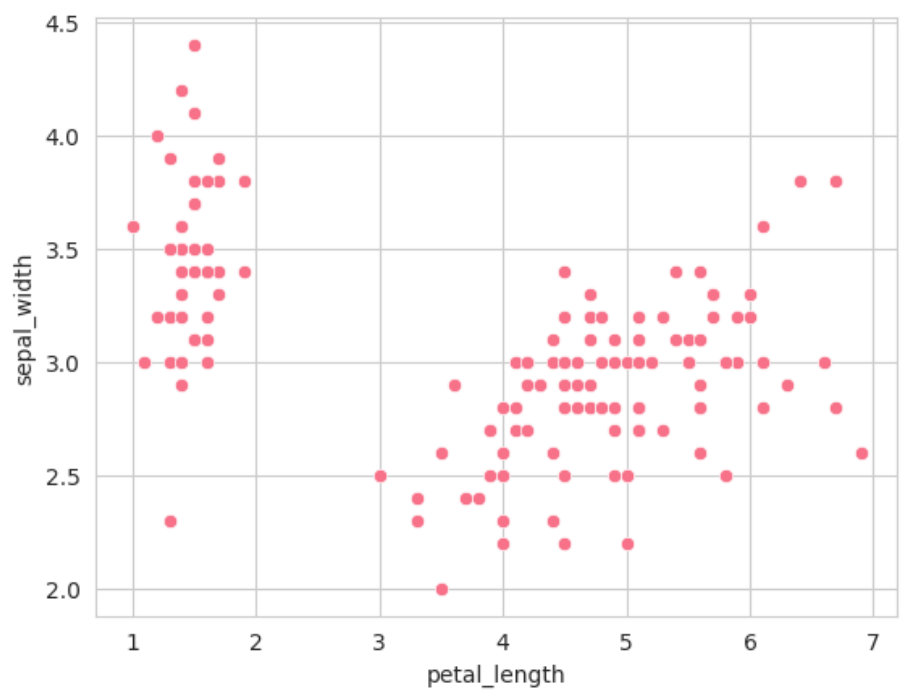


Fonte: autoria própria.

```
# Definindo o estilo do gráfico
sns.set_style("whitegrid")
```

```
sns.set_style("whitegrid")

# Cria um scatter plot das colunas petal length vs. sepal width
sns.scatterplot(data=iris, x="petal_length", y="sepal_width")
plt.show()
```

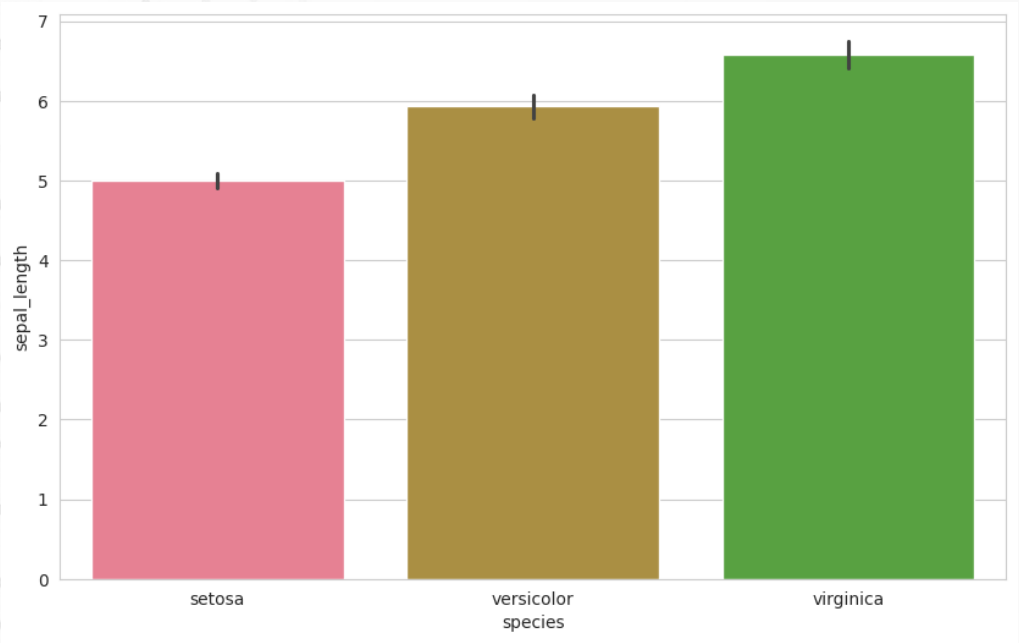
Figura 35 - *Scatter plot* das colunas *petal length vs. width*



Fonte: autoria própria.

```
# Personalizando a paleta de cores
sns.set_palette("husl")

plt.figure(figsize=(10, 6))
sns.barplot(x='species', y='sepal_length', data=iris, hue='species')
plt.show()
```

Figura 36 - Personalização da paleta de cores



Fonte: autoria própria.

3.5 Saiba Mais...

Matplotlib Developers. *Matplotlib documentation.*

Michael Waskom. *Seaborn documentation.*

CENTRO DE COMPETÊNCIA EMBRAPII EM TECNOLOGIAS IMERSIVAS

Python para Processamento de Dados

Unidade IV

Estudo de Caso

Unidade IV - Estudo de Caso

No estudo de caso detalhado que será apresentado no *Notebook Colab* anexo, utilizaremos o *dataset* Titanic, um conjunto de dados amplamente explorado na comunidade de ciência de dados. Esse *dataset* contém informações demográficas e de viagem sobre os passageiros a bordo do Titanic, que afundou em sua viagem inaugural em 1912. As variáveis incluem idade, sexo, classe da cabine, tarifa paga e sobrevivência, entre outras.

4.1 Objetivos do Estudo de Caso com o *Dataset* Titanic

- **Uso do *NumPy***: realizar operações como cálculos de estatísticas descritivas para idade e tarifas, além de manipular *arrays* para derivar novas métricas que possam ser úteis para a análise.
- **Utilização do *Pandas***: aplicar técnicas de manipulação de dados para limpar o *dataset*, preenchendo dados faltantes e filtrando informações relevantes; explorar a agregação de dados para entender melhor as taxas de sobrevivência por diferentes categorias.
- **Visualizações com *Matplotlib* e *Seaborn***: criar gráficos que ilustrem a distribuição da idade dos passageiros, as taxas de sobrevivência por classe e sexo e outras visualizações relevantes que ajudem a contar a história dos dados do Titanic.

O *dataset* Titanic é particularmente interessante para análise porque contém uma mistura de variáveis numéricas e categóricas e apresenta um desafio realista para prever a sobrevivência, tornando-se um caso clássico para métodos de classificação em aprendizado de máquina. Esse estudo de caso não apenas reforçará a aplicação prática das técnicas discutidas, mas também proporcionará uma oportunidade de entender como percepções podem ser extraídas de um evento histórico trágico, utilizando ferramentas modernas de análise de dados.

4.2 Notebook Colab

No estudo de caso detalhado que será apresentado neste *notebook*, utilizaremos o *dataset* Titanic, um conjunto de dados amplamente explorado na comunidade de ciência de dados. Este *dataset* contém informações demográficas e de viagem sobre

os passageiros a bordo do Titanic, que afundou em sua viagem inaugural em 1912. As variáveis incluem idade, sexo, classe da cabine, tarifa paga, sobrevivência, entre outras.

O *dataset* Titanic é particularmente interessante para análise porque contém uma mistura de variáveis numéricas e categóricas e apresenta um desafio *real-world* para prever a sobrevivência, tornando-se um caso clássico para métodos de classificação em aprendizado de máquina. Este estudo de caso não apenas reforçará a aplicação prática das técnicas discutidas, mas também proporcionará uma oportunidade de entender como *insights* podem ser extraídos de um evento histórico trágico utilizando ferramentas modernas de análise de dados.

Todas as estruturas, funções e métodos utilizados nesse *notebook* estão detalhadas nas Unidades do *Ebook*. É de forte recomendação que as Unidades anteriores sejam estudadas e entendidas antes de começar esta Unidade.

Configuração Inicial e Carregamento dos Dados

O *dataset* do Titanic será carregado a partir de um arquivo CSV presente no repositório seaborn-data. Todos os métodos e estruturas utilizadas foram estudados nas Unidades anteriores. Este é apenas um estudo de caso com exemplo prático, utilizando um conjunto de dados realista.

```
# Importar bibliotecas necessárias
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
# Carregar o dataset Titanic
url = 'https://raw.githubusercontent.com/mwaskom/seaborn-data/master/titanic.csv'
titanic = pd.read_csv(url)
# Exibir as primeiras linhas do dataset
titanic.head()
```

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	deck	embark_town	alive	alone
0	0	3	male	22.0	1	0	7.2500	S	Third	man	True	NaN	Southampton	no	False
1	1	1	female	38.0	1	0	71.2833	C	First	woman	False	C	Cherbourg	yes	False
2	1	3	female	26.0	0	0	7.9250	S	Third	woman	False	NaN	Southampton	yes	True
3	1	1	female	35.0	1	0	53.1000	S	First	woman	False	C	Southampton	yes	False
4	0	3	male	35.0	0	0	8.0500	S	Third	man	True	NaN	Southampton	no	True

Dicionário de Dados do *Dataset* Titanic

- **survived**: indica se o passageiro sobreviveu ao desastre do Titanic.
  - 0 = Não
  - 1 = Sim
- **pclass**: classe do bilhete do passageiro, um indicador do status socioeconômico.
  - 1 = Primeira Classe
  - 2 = Segunda Classe
  - 3 = Terceira Classe
- **sex**: sexo do passageiro.
  - male = Masculino
  - female = Feminino
- **age**: idade do passageiro em anos.
- **sibsp**: número de irmãos/cônjuges a bordo do Titanic.
- **parch**: número de pais/filhos a bordo do Titanic.
- **fare**: tarifa paga pelo passageiro.
- **embarked**: código do porto de embarque do passageiro.
  - C = Cherbourg
  - Q = Queenstown
  - S = Southampton
- **class**: categoria da classe do passageiro, similar a **pclass**.
  - First = Primeira Classe
  - Second = Segunda Classe
  - Third = Terceira Classe
- **who**: descrição do passageiro.
  - child = Criança
  - man = Homem
  - woman = Mulher
- **adult_male**: indica se o passageiro é um homem adulto.
  - True = Sim
  - False = Não
- **deck**: letra que indica o deck da cabine do passageiro.
- **embark_town**: nome da cidade de embarque do passageiro.
  - Cherbourg
  - Queenstown

- Southampton
- **alive**: estado de sobrevivência do passageiro em formato de texto.
  - yes = Sim
  - no = Não
- **alone**: indica se o passageiro estava viajando sozinho.
  - True = Sim
  - False = Não

Verificar tipos dos atributos

```
titanic.dtypes
```

```
survived        int64
pclass          int64
sex            object
age           float64
sibsp           int64
parch           int64
fare          float64
embarked       object
class          object
who            object
adult_male       bool
deck           object
embark_town    object
alive          object
alone            bool
dtype: object
```

Forma e dimensões do *dataset*

```
# [0] = Quantidade de instâncias
# [1] = Quantidade de atributos
print("O dataset contém ", titanic.shape[0], "instâncias e ", titanic.shape[1], " atributos.")
```

```
O dataset contém  891 instâncias e  15  atributos.
```

Descrever Estatisticamente os Dados

Nesta seção, realizaremos uma descrição estatística do *dataset* Titanic. Isso inclui obter medidas de tendência central, dispersão e a distribuição das variáveis numéricas e categóricas. A descrição estatística é essencial para entender o escopo dos dados, identificar possíveis *outliers* e planejar os passos subsequentes para a análise dos dados mais detalhada.

```
# Exibir estatísticas descritivas para variáveis numéricas
titanic.describe()
```

	survived	pclass	age	sibsp	parch	fare
count	891.000000	891.000000	714.000000	891.000000	891.000000	891.000000
mean	0.383838	2.308642	29.699118	0.523008	0.381594	32.204208
std	0.486592	0.836071	14.526497	1.102743	0.806057	49.693429
min	0.000000	1.000000	0.420000	0.000000	0.000000	0.000000
25%	0.000000	2.000000	20.125000	0.000000	0.000000	7.910400
50%	0.000000	3.000000	28.000000	0.000000	0.000000	14.454200
75%	1.000000	3.000000	38.000000	1.000000	0.000000	31.000000
max	1.000000	3.000000	80.000000	8.000000	6.000000	512.329200

```
# Exibir estatísticas descritivas para variáveis categóricas
titanic.describe(include=['O'])
```

	sex	embarked	class	who	deck	embark_town	alive
count	891	889	891	891	203	889	891
unique	2	3	3	3	7	3	2
top	male	S	Third	man	C	Southampton	no
freq	577	644	491	537	59	644	549

```
# Calcular e exibir a moda para as variáveis categóricas
moda_categoricas = titanic[['sex', 'embarked', 'class']].mode()
moda_categoricas
```

	sex	embarked	class
0	male	S	Third

Limpeza de Dados

A limpeza dos dados é uma etapa crucial antes de qualquer análise aprofundada. Nesta seção, focaremos na preparação do *dataset* Titanic, tratando dados faltantes, removendo duplicatas e ajustando tipos de dados quando necessário. Esta limpeza ajudará a garantir que as análises subsequentes sejam baseadas em dados precisos e representativos.

Exibição de informações iniciais: Utilizamos titanic.info() para obter um resumo das colunas, tipos de dados e a presença de valores faltantes.

Na saída a seguir, é possivel observar que foi informado o tipo (Dtype) dos dados de cada uma das colunas, bem como uma coluna de nome Non-Null Count. Essa coluna apresenta a quantidade de entradas em cada coluna que possuem valores não nulos. Essa informação é especialmente relevante para a limpeza de dados. Por exemplo, por meio dessa análise, pode-se ver que a coluna deck possui apenas 203 entradas não nulas de um total de 891. Um forte indício de que essa coluna pode ser removida, pois não descreve razoavelmente a população presente no conjunto de dados.

```
# Exibir informações iniciais para identificar dados faltantes e tipos de dados
titanic.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 15 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   survived     891 non-null    int64
 1   pclass       891 non-null    int64
 2   sex          891 non-null    object
 3   age          714 non-null    float64
 4   sibsp        891 non-null    int64
 5   parch        891 non-null    int64
 6   fare         891 non-null    float64
 7   embarked     889 non-null    object
 8   class        891 non-null    object
 9   who          891 non-null    object
 10  adult_male   891 non-null    bool
 11  deck         203 non-null    object
 12  embark_town  889 non-null    object
 13  alive        891 non-null    object
 14  alone        891 non-null    bool
dtypes: bool(2), float64(2), int64(4), object(7)
memory usage: 92.4+ KB
```

Conversão de tipos de dados: a coluna *fare* (tarifa paga pelo passageiro) é explicitamente convertida para tipo float32, um tamanho menor que o float64(padrão), ocupando metade da memória. Útil para quando não se deseja precisão com muitas casas decimais.

```
# Converter o tipo de dado da coluna 'fare' para float32, se necessário
titanic['fare'] = titanic['fare'].astype(np.float32)
titanic.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 15 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
```

```
0  survived     891 non-null    int64
1  pclass       891 non-null    int64
2  sex          891 non-null    object
3  age          714 non-null    float64
4  sibsp        891 non-null    int64
5  parch        891 non-null    int64
6  fare         891 non-null    float32
7  embarked     889 non-null    object
8  class        891 non-null    object
9  who          891 non-null    object
10 adult_male   891 non-null    bool
11 deck         203 non-null    object
12 embark_town  889 non-null    object
13 alive        891 non-null    object
14 alone        891 non-null    bool
dtypes: bool(2), float32(1), float64(1), int64(4), object(7)
memory usage: 88.9+ KB
```

Tratamento de Dados Faltantes e Duplicados:

» **Idades:** as idades faltantes são substituídas pela média das idades existentes, o que é uma prática comum para manter a distribuição geral.

```
# Mostrando dados faltantes na coluna 'age' (idades)
titanic[titanic['age'].isnull()]
```

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	deck	embark_town	alive	alone
5	0	3	male	NaN	0	0	8.458300	Q	Third	man	True	NaN	Queenstown	no	True
17	1	2	male	NaN	0	0	13.000000	S	Second	man	True	NaN	Southampton	yes	True
19	1	3	female	NaN	0	0	7.225000	C	Third	woman	False	NaN	Cherbourg	yes	True
26	0	3	male	NaN	0	0	7.225000	C	Third	man	True	NaN	Cherbourg	no	True
28	1	3	female	NaN	0	0	7.879200	Q	Third	woman	False	NaN	Queenstown	yes	True
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
859	0	3	male	NaN	0	0	7.229200	C	Third	man	True	NaN	Cherbourg	no	True
863	0	3	female	NaN	8	2	69.550003	S	Third	woman	False	NaN	Southampton	no	False
868	0	3	male	NaN	0	0	9.500000	S	Third	man	True	NaN	Southampton	no	True
878	0	3	male	NaN	0	0	7.895800	S	Third	man	True	NaN	Southampton	no	True
888	0	3	female	NaN	1	2	23.450001	S	Third	woman	False	NaN	Southampton	no	False

177 rows × 15 columns

```
# Substituir dados faltantes na coluna 'age' pela média das idades
titanic['age'].fillna(titanic['age'].mean(), inplace=True)
```

```
# Mostrando dados faltantes na coluna 'age' (idades)
titanic[titanic['age'].isnull()]
```

survived pclass sex age sibsp parch fare embarked class
who adult_male deck embark_town alive alone

» **Porto de Embarque:** dados faltantes na coluna *embarked* são substituídos pelo valor mais comum, que é o porto de embarque que aparece com maior frequência.

```
# Mostrando dados faltantes na coluna 'embarked' (porto de embarque)
titanic[titanic['embarked'].isnull()]
```

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	deck	embark_town	alive	alone
61	1	1	female	38.0	0	0	80.0	NaN	First	woman	False	B	NaN	yes	True
829	1	1	female	62.0	0	0	80.0	NaN	First	woman	False	B	NaN	yes	True

```
# Substituir dados faltantes na coluna 'embarked' pelo porto mais comum
most_common_embarked = titanic['embarked'].mode()[0]
titanic['embarked'].fillna(most_common_embarked, inplace=True)
```

```
# Mostrando dados faltantes na coluna 'embarked' (porto de embarque)
titanic[titanic['embarked'].isnull()]
```

survived pclass sex age sibsp parch fare embarked class
who adult_male deck embark_town alive alone

» **Remoção de Colunas:** a coluna *deck* é removida devido ao grande número de dados faltantes, o que pode comprometer a qualidade das análises.

```
# Mostrando dados faltantes na coluna 'deck' (deck da cabine do passageiro)
titanic[titanic['deck'].isnull()]
```

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	deck	embark_town	alive	alone
0	0	3	male	22.000000	1	0	7.250000	S	Third	man	True	NaN	Southampton	no	False
2	1	3	female	26.000000	0	0	7.925000	S	Third	woman	False	NaN	Southampton	yes	True
4	0	3	male	35.000000	0	0	8.050000	S	Third	man	True	NaN	Southampton	no	True
5	0	3	male	29.699118	0	0	8.458300	Q	Third	man	True	NaN	Queenstown	no	True
7	0	3	male	2.000000	3	1	21.075001	S	Third	child	False	NaN	Southampton	no	False
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
884	0	3	male	25.000000	0	0	7.050000	S	Third	man	True	NaN	Southampton	no	True
885	0	3	female	39.000000	0	5	29.125000	Q	Third	woman	False	NaN	Queenstown	no	False
886	0	2	male	27.000000	0	0	13.000000	S	Second	man	True	NaN	Southampton	no	True
888	0	3	female	29.699118	1	2	23.450001	S	Third	woman	False	NaN	Southampton	no	False
890	0	3	male	32.000000	0	0	7.750000	Q	Third	man	True	NaN	Queenstown	no	True

688 rows × 15 columns

```
# Remover a coluna 'deck', pois contém muitos valores faltantes
titanic.drop('deck', axis=1, inplace=True)
titanic
```

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	embark_town	alive	alone
0	0	3	male	22.000000	1	0	7.250000	S	Third	man	True	Southampton	no	False
1	1	1	female	38.000000	1	0	71.283302	C	First	woman	False	Cherbourg	yes	False
2	1	3	female	26.000000	0	0	7.925000	S	Third	woman	False	Southampton	yes	True
3	1	1	female	35.000000	1	0	53.099998	S	First	woman	False	Southampton	yes	False
4	0	3	male	35.000000	0	0	8.050000	S	Third	man	True	Southampton	no	True
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
886	0	2	male	27.000000	0	0	13.000000	S	Second	man	True	Southampton	no	True
887	1	1	female	19.000000	0	0	30.000000	S	First	woman	False	Southampton	yes	True
888	0	3	female	29.699118	1	2	23.450001	S	Third	woman	False	Southampton	no	False
889	1	1	male	26.000000	0	0	30.000000	C	First	man	True	Cherbourg	yes	True
890	0	3	male	32.000000	0	0	7.750000	Q	Third	man	True	Queenstown	no	True

891 rows × 14 columns

» **Remoção de duplicatas:** linhas duplicadas são removidas para evitar distorções nas análises estatísticas.

```
# Mostrando linhas duplicadas do dataframe titanic
titanic[titanic.duplicated()]
```

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	embark_town	alive	alone
47	1	3	female	29.699118	0	0	7.7500	Q	Third	woman	False	Queenstown	yes	True
76	0	3	male	29.699118	0	0	7.8958	S	Third	man	True	Southampton	no	True
77	0	3	male	29.699118	0	0	8.0500	S	Third	man	True	Southampton	no	True
87	0	3	male	29.699118	0	0	8.0500	S	Third	man	True	Southampton	no	True
95	0	3	male	29.699118	0	0	8.0500	S	Third	man	True	Southampton	no	True
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
870	0	3	male	26.000000	0	0	7.8958	S	Third	man	True	Southampton	no	True
877	0	3	male	19.000000	0	0	7.8958	S	Third	man	True	Southampton	no	True
878	0	3	male	29.699118	0	0	7.8958	S	Third	man	True	Southampton	no	True
884	0	3	male	25.000000	0	0	7.0500	S	Third	man	True	Southampton	no	True
886	0	2	male	27.000000	0	0	13.0000	S	Second	man	True	Southampton	no	True

111 rows × 14 columns

```
# Remover linhas duplicadas
titanic.drop_duplicates(inplace=True)
titanic
```

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	embark_town	alive	alone
0	0	3	male	22.000000	1	0	7.250000	S	Third	man	True	Southampton	no	False
1	1	1	female	38.000000	1	0	71.283302	C	First	woman	False	Cherbourg	yes	False
2	1	3	female	26.000000	0	0	7.925000	S	Third	woman	False	Southampton	yes	True
3	1	1	female	35.000000	1	0	53.099998	S	First	woman	False	Southampton	yes	False
4	0	3	male	35.000000	0	0	8.050000	S	Third	man	True	Southampton	no	True
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
885	0	3	female	39.000000	0	5	29.125000	Q	Third	woman	False	Queenstown	no	False
887	1	1	female	19.000000	0	0	30.000000	S	First	woman	False	Southampton	yes	True
888	0	3	female	29.699118	1	2	23.450001	S	Third	woman	False	Southampton	no	False
889	1	1	male	26.000000	0	0	30.000000	C	First	man	True	Cherbourg	yes	True
890	0	3	male	32.000000	0	0	7.750000	Q	Third	man	True	Queenstown	no	True

780 rows × 14 columns

Análise de Correlações e Proporções

Nesta seção, analisaremos as correlações entre as variáveis do *dataset* Titanic e as proporções significativas que podem indicar fatores influentes na sobrevivência dos passageiros. Utilizaremos técnicas de visualização para ilustrar essas relações, facilitando a identificação de padrões e *insights*.

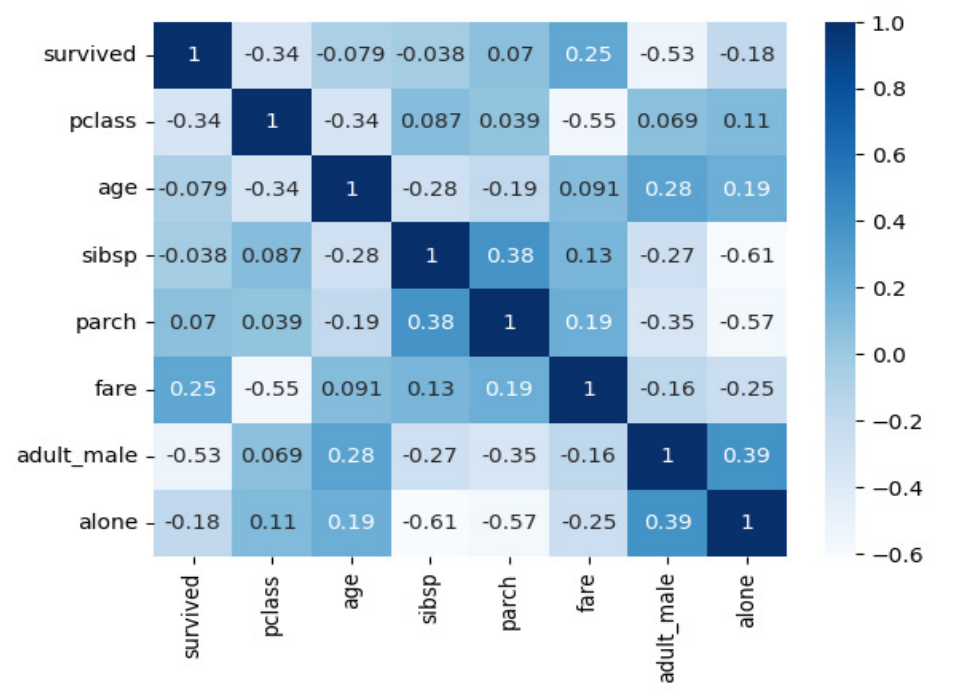
Calcular Correlações

Uma das ferramentas mais utilizadas para visualização de correlações entre variáveis é o gráfico de calor. Este gráfico é apresentado em forma de matriz quadrada, em que cada linha, e suas respectivas colunas, representam uma variável. Assim, a coluna 0 representa a mesma variável que a linha 0 (survived no exemplo abaixo). O valor apresentado nas células da matriz diz a correlação entre as variáveis em questão. Assim, é correto afirmar que o triangular inferior é igual ao triangular superior da matriz. O gráfico de calor é interessante, pois facilita a visualização utilizando cores mais escuras para correlações diretas mais fortes e cores mais claras para correlações indiretas mais fortes.

No gráfico a seguir, é possível observar nas células 5x0 e 0x5 o valor 0.25, que demonstra uma correlação importante entre a taxa paga (fare) e a sobrevivência do passageiro (survived). Uma correlação ainda mais expressiva é apresentada nas células 1x0 e 0x1, com o valor -0.34. Este valor indica uma correlação inversa entre a classe e a sobrevivência do passageiro, ou seja, passageiros de classes menores (primeira classe, por exemplo) tiveram maior sobrevivência. A maior correlação apresentada no conjunto de dados, todavia, é inversa entre sibsp e alone(-0,61), indicando que pessoas com irmãos/cônjuges a bordo do Titanic não estavam viajando sozinhas.

```
# Somente atributos numéricos são considerados
plt.suptitle("Gráfico de Calor das Correlações entre os Atributos Numéricos")
numericos = titanic.select_dtypes(include=['int', 'float', 'bool'])
sns.heatmap(numericos.corr(), annot=True, cmap='Blues')
```

Figura 37 - Gráfico de Calor das Correlações entre os Atributos Numéricos



Fonte: autoria própria.

Análisar Proporções

Homens x Mulheres

Os resultados a seguir mostram que apenas 21,72% dos homens sobreviveram. No caso das mulheres, 73,97% sobreviveram. Isso indica um reflexo da abordagem costumeira da época de salvar mulheres primeiro.

Nos códigos, inicialmente filtramos (com loc) o *DataFrame* pelo gênero (titanic['sex'] == 'male' ou titanic['sex'] == 'female'), selecionando apenas a coluna survived. Em seguida, contamos (count()) o total de pessoas daquele gênero e os totais para cada valor em survived (value_counts()). Por fim, as porcentagens são calculadas.

```
# Filtrando apenas a coluna 'survived' para homens
homens = titanic.loc[titanic['sex'] == 'male', 'survived']
homens
```

```
0      0
4      0
5      0
6      0
7      0
      ..
876    0
881    0
883    0
889    1
890    0
Name: survived, Length: 488, dtype: int64
```

```
# Porcentagem de sobrevivência de homens
totalHomens = homens.count()
(homens.value_counts() / totalHomens) * 100
```

```
    survived
0  78.278689
1  21.721311
Name: count, dtype: float64
```

```
# Filtrando apenas a coluna 'survived' para mulheres
mulheres = titanic.loc[titanic['sex'] == 'female', 'survived']
mulheres
```

```
1      1
2      1
3      1
8      1
9      1
      ..
880    1
882    0
885    0
887    1
888    0
Name: survived, Length: 292, dtype: int64
```

```
# Porcentagem de sobrevivência de mulheres
totalMulheres = mulheres.count()
(mulheres.value_counts() / totalMulheres) * 100
```

```
    survived
1  73.972603
0  26.027397
Name: count, dtype: float64
```

Crianças x Adultos

Utilizando a mesma abordagem da análise de sobrevivência de mulheres e homens, os códigos abaixo realizam a filtragem por idade, considerando pessoas com 17 anos ou menos como crianças e as demais como adultos. Nota-se que mais da metade das crianças (54,55%) foram salvas, enquanto apenas 39,10% dos adultos sobreviveram. Ainda que mais frágeis, a maior sobrevivência das crianças talvez seja um indício da abordagem comum à época de salvar crianças primeiro.

```
# Porcentagem de sobrevivência de crianças e adolescentes (<= 17 anos)
criancas = titanic.loc[titanic['age'] <= 17, 'survived']
totalCriancas = criancas.count()
(criancas.value_counts() / totalCriancas) * 100
```

```
survived
1    54.545455
0    45.454545
Name: count, dtype: float64
```

```
# Porcentagem de sobrevivência de adultos (> 17)
adultos = titanic.loc[titanic['age'] > 17, 'survived']
totalAdultos = adultos.count()
(adultos.value_counts() / totalAdultos) * 100
```

```
survived
0    60.895522
1    39.104478
Name: count, dtype: float64
```

1° Classe x 2° Classe x 3° Classe

Na comparação das classes, utilizando a mesma abordagem das comparações anteriores, também é possível identificar um padrão. Na primeira classe, 63,68% das pessoas sobreviveram. Já a segunda classe obteve uma sobrevivência de 50,61%. Por fim, na terceira classe, apenas 25,74% dos passageiros sobreviveram. Esses dados apresentam uma clara importância da classe em relação à sobrevivência dos passageiros.

```
# Porcentagem de sobrevivência na primeira classe
total1classe = titanic.loc[titanic['pclass'] == 1, 'survived'].count()
(titanic.loc[titanic['pclass'] == 1, 'survived'].value_counts() / total1classe) * 100
```

```
survived
1    63.679245
0    36.320755
Name: count, dtype: float64
```

```
# Porcentagem de sobrevivência na segunda classe
total2classe = titanic.loc[titanic['pclass'] == 2, 'survived'].count()
(titanic.loc[titanic['pclass'] == 2, 'survived'].value_counts() / total2classe) * 100
```

```
survived
1    50.609756
0    49.390244
Name: count, dtype: float64
```

```
# Porcentagem de sobrevivência na terceira classe
total3classe = titanic.loc[titanic['pclass'] == 3, 'survived'].count()
(titanic.loc[titanic['pclass'] == 3, 'survived'].value_counts() / total3classe) * 100
```

```
    survived
0  74.257426
1  25.742574
Name: count, dtype: float64
```

Tabelas Dinâmicas ou Pivô

Nos exemplos a seguir, são apresentadas análises semelhantes às anteriores mas, agora, utilizando tabelas pivô. No primeiro exemplo, pivot_class, observa-se as médias de sobrevivência em cada classe (os mesmos valores obtidos anteriormente). Já no segundo exemplo, pivot_sex a tabela pivô apresenta as médias dos valores de sobrevivência para cada gênero, indicando, como anteriormente, que 73,97% das mulheres e apenas 21,72% dos homens sobreviveram. Por fim, pivot_embarked apresenta que houve uma sobrevivencia maior daqueles que embarcaram em Cherbourg (C) com um total de 58,06%. Em seguida, está Southampton (S), com 37,39% e Queenstown (Q) com 34,48%.

```
# Criar uma pivot table para explorar a relação entre 'survived' e outras variáveis categóricas
pivot_class = pd.pivot_table(titanic, values='survived', index='class', aggfunc='mean')
display(pivot_class)

pivot_sex = pd.pivot_table(titanic, values='survived', index='sex', aggfunc='mean')
display(pivot_sex)

pivot_embarked = pd.pivot_table(titanic, values='survived', index='embarked', aggfunc='mean')
display(pivot_embarked)
```

class	survived
First	0.636792
Second	0.506098
Third	0.257426

sex	survived
female	0.739726
male	0.217213

embarked	survived
C	0.580645
Q	0.344828
S	0.373898

Análise Por Meio de Visualização de Dados

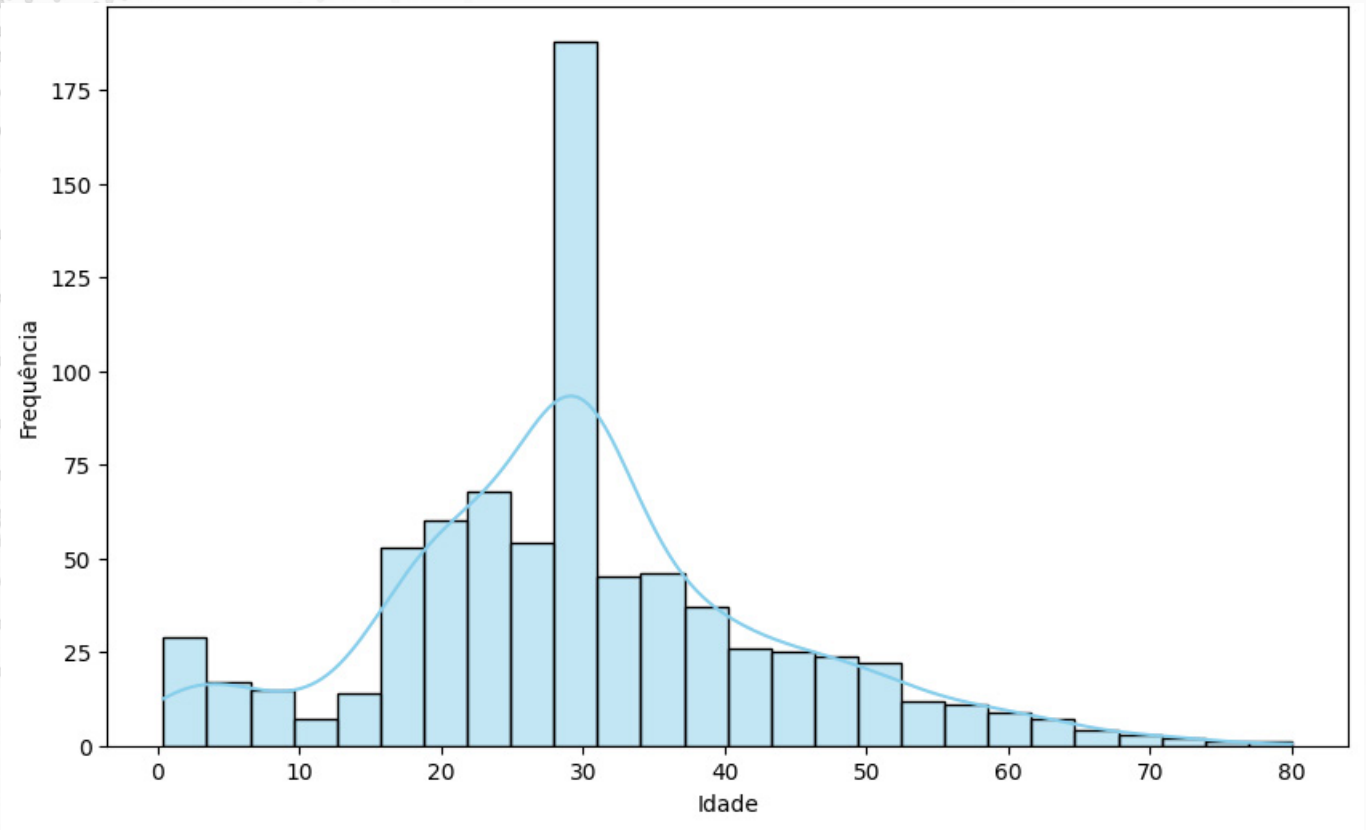
Visualizar dados é crucial para entender complexidades e extrair *insights* significativos. Nesta seção, utilizaremos gráficos para explorar diversas facetas do *dataset* Titanic, focando em distribuições, relações e padrões que podem ajudar a elucidar fatores impactantes na sobrevivência dos passageiros.

Análises em Relação à Idade

Nos gráficos a seguir, é possível observar que a grande maioria dos passageiros possuía idades entre 20 e 40 anos. Já no gráfico de violino, nota-se uma maior ocorrência de sobreviventes entre as crianças.

```
# Distribuição da Idade
plt.figure(figsize=(10, 6))
sns.histplot(titanic['age'].dropna(), kde=True, color="skyblue")
plt.title('Distribuição da Idade dos Passageiros')
plt.xlabel('Idade')
plt.ylabel('Frequência')
plt.show()
```

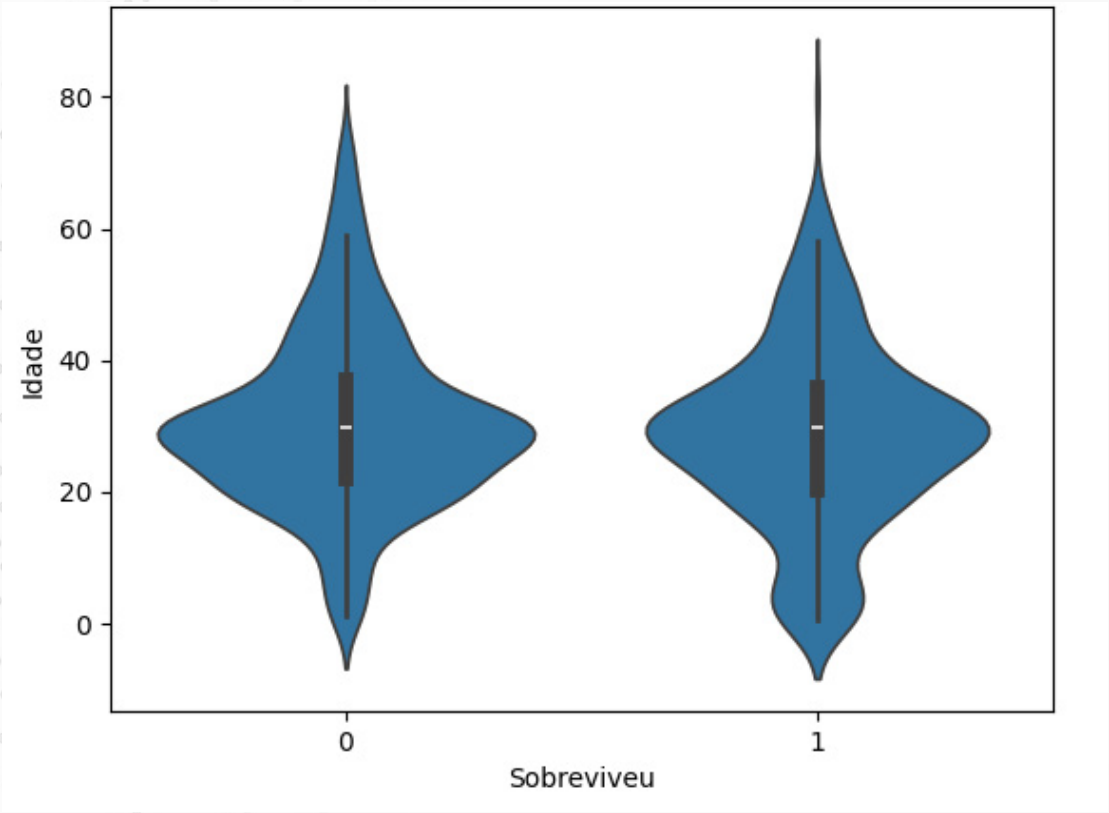
Figura 38 - Distribuição da Idade dos Passageiros



Fonte: autoria própria.

```
sns.violinplot(x='survived', y='age', data=titanic)
plt.title('Distribuição de Idades entre Sobreviventes e Não Sobreviventes')
plt.xlabel('Sobreviveu')
plt.ylabel('Idade')
plt.show()
```

Figura 39 - Distribuição de Idades entre Sobreviventes e Não Sobreviventes



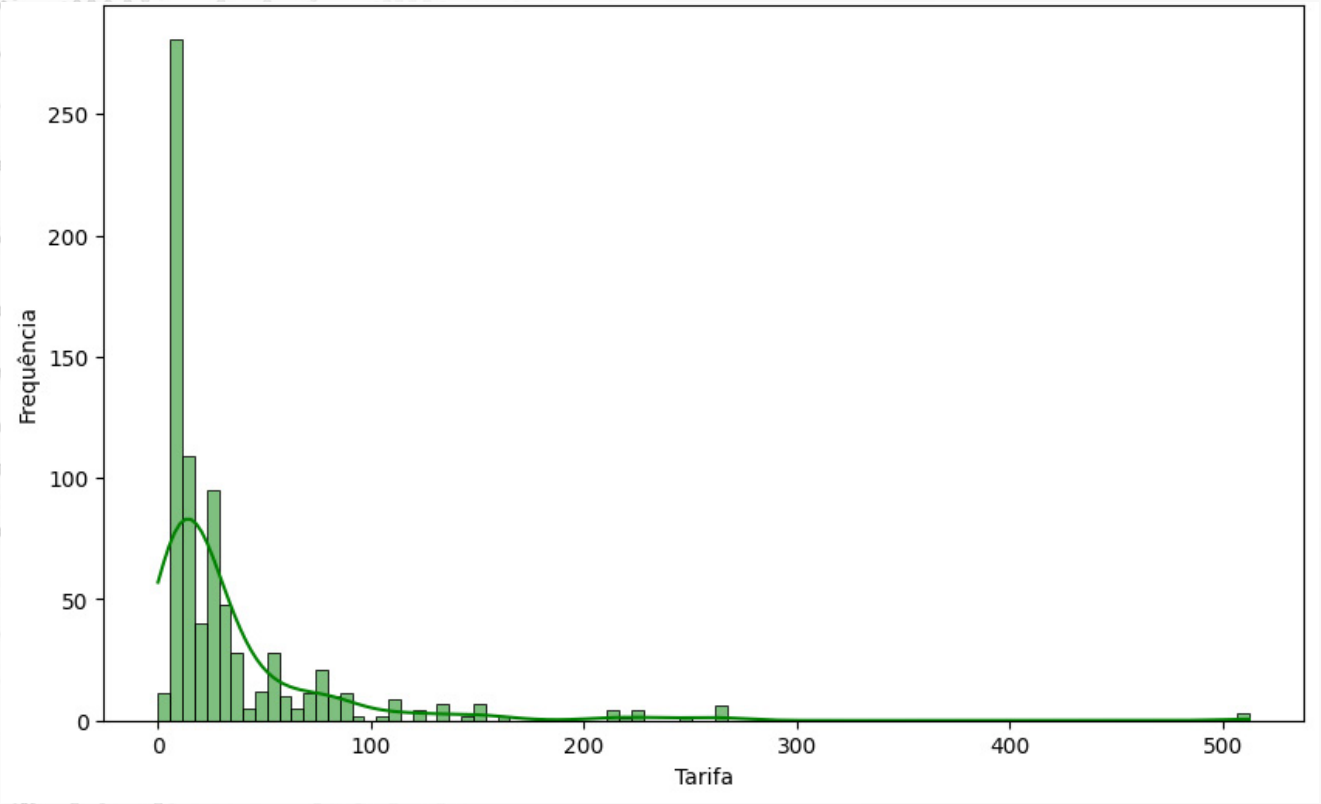
Fonte: autoria própria.

Análise em relação à tarifa paga

Nos gráficos a seguir, é possível observar que a grande maioria dos passageiros pagou uma tarifa menor que 50 dolares. Já no segundo gráfico, de dispersão, nota-se que tarifas mais altas (eixo Y) levaram à maior sobrevivência (vermelho). Além disso, as tarifas mais altas estão relacionadas com a classe (tamanho do ponto).

```
# Distribuição da Tarifa
plt.figure(figsize=(10, 6))
sns.histplot(titanic['fare'], kde=True, color="green")
plt.title('Distribuição da Tarifa Paga pelos Passageiros')
plt.xlabel('Tarifa')
plt.ylabel('Frequência')
plt.show()
```

Figura 40 - Distribuição da Tarifa Paga pelos Passageiros

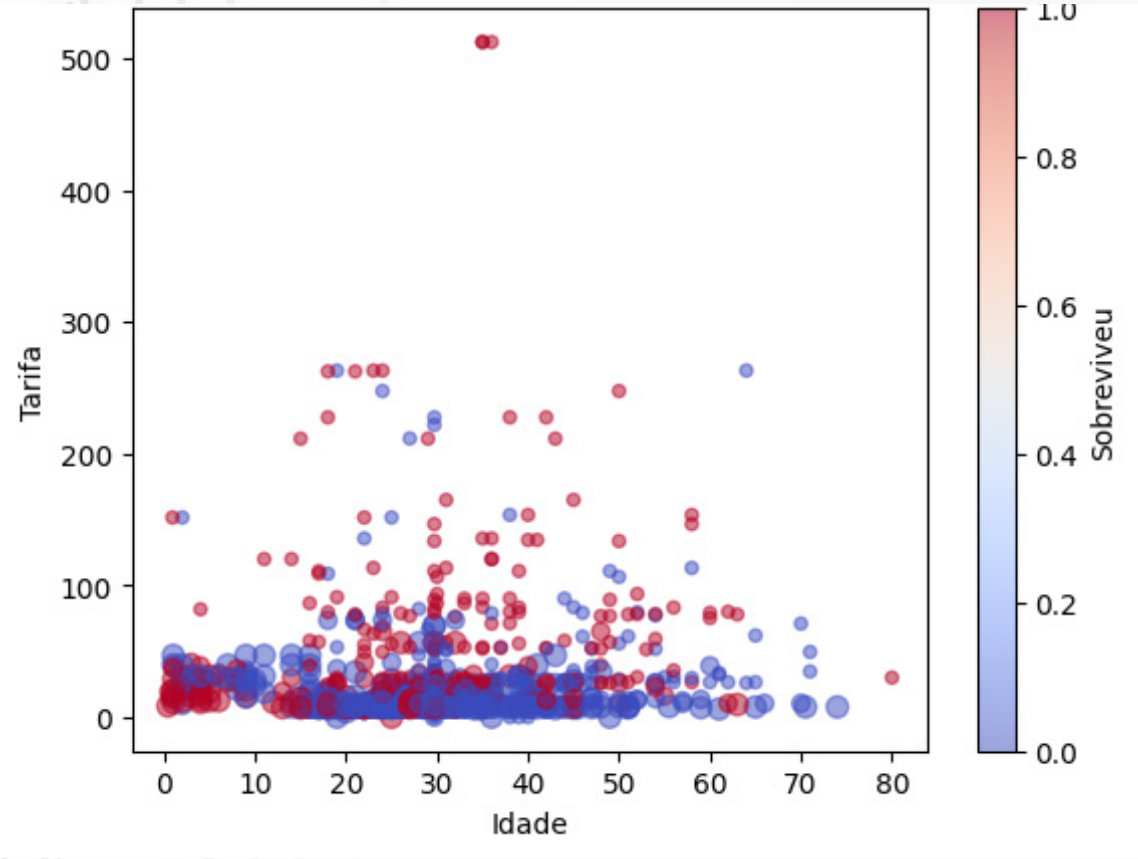


Fonte: autoria própria.

No gráfico a seguir, a coluna age é definida como eixo X e fare como eixo Y. A transparência dos pontos é configurada em 50% (alpha=0.5), o tamanho do ponto é configurado para representar a classe (quanto menor melhor), e a cor do ponto representa a sobrevivência (vermelho indica sobrevivente). Nota-se que tarifas maiores ou idades menores possuem maior aglomeração de sobreviventes.

```
plt.scatter(titanic['age'], titanic['fare'], alpha=0.5,
            s=titanic['pclass']*20,  # Tamanho do ponto baseado em pclass
            c=titanic['survived'], cmap='coolwarm')  # Cor baseada na sobrevivência
plt.colorbar(label='Sobreviveu')
plt.title('Relação entre Idade, Tarifa e Sobrevivência')
plt.xlabel('Idade')
plt.ylabel('Tarifa')
plt.show()
```

Figura 41 - Relação entre Idade, Tarifa e Sobrevivência



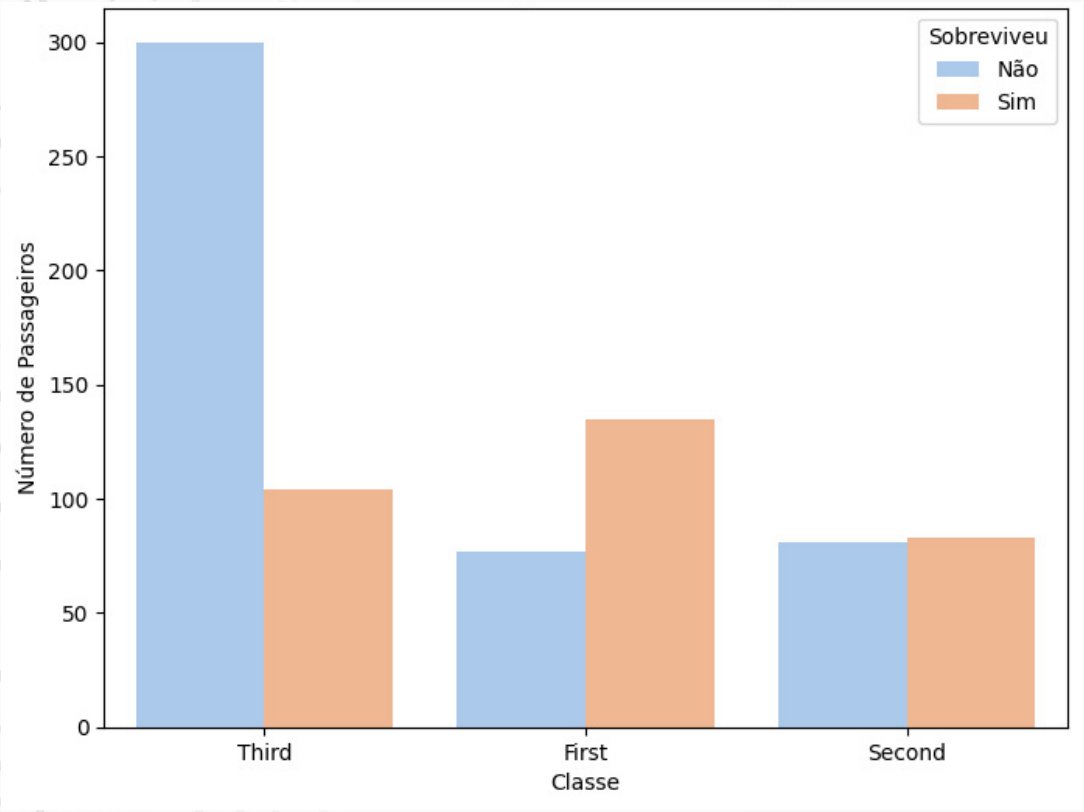
Fonte: autoria própria.

Análise em Relação à Classe e Sexo dos Passageiros

Os gráficos de barras são úteis para fazer comparações entre classes. No exemplo a seguir, estamos utilizando countplot para gerar um gráfico em barras dos sobreviventes em relação a cada classe. Nota-se que a classe que menos sobreviveu foi a terceira.

```
# Sobrevivência por Classe
plt.figure(figsize=(8, 6))
sns.countplot(x='class', hue='survived', data=titanic, palette='pastel')
plt.title('Sobrevivência por Classe de Passageiro')
plt.xlabel('Classe')
plt.ylabel('Número de Passageiros')
plt.legend(title='Sobreviveu', labels=['Não', 'Sim'])
plt.show()
```

Figura 42 - Sobrevivência por Classe de Passageiro

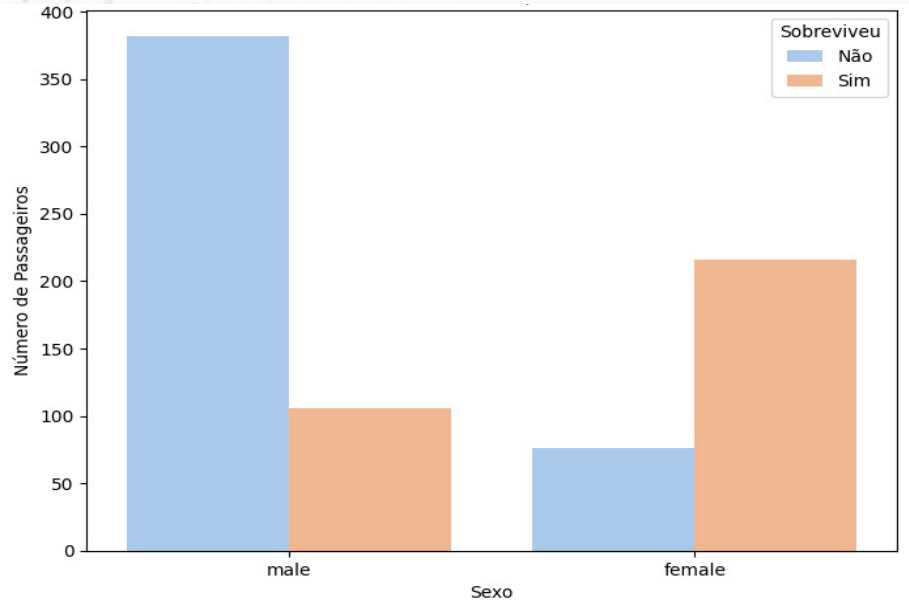


Fonte: autoria própria.

Outro exemplo de uso de countplot é dado a seguir. Neste caso, cada classe de sobrevivência (Sim ou Não) também é representada pelas cores e, no eixo X, há a separação por gênero. Nota-se que homens sobreviveram bem menos que mulheres.

```
# Sobrevivência por Sexo
plt.figure(figsize=(8, 6))
sns.countplot(x='sex', hue='survived', data=titanic, palette='pastel')
plt.title('Sobrevivência por Sexo')
plt.xlabel('Sexo')
plt.ylabel('Número de Passageiros')
plt.legend(title='Sobreviveu', labels=['Não', 'Sim'])
plt.show()
```

Figura 43 - Sobrevivência por Sexo



Fonte: autoria própria.

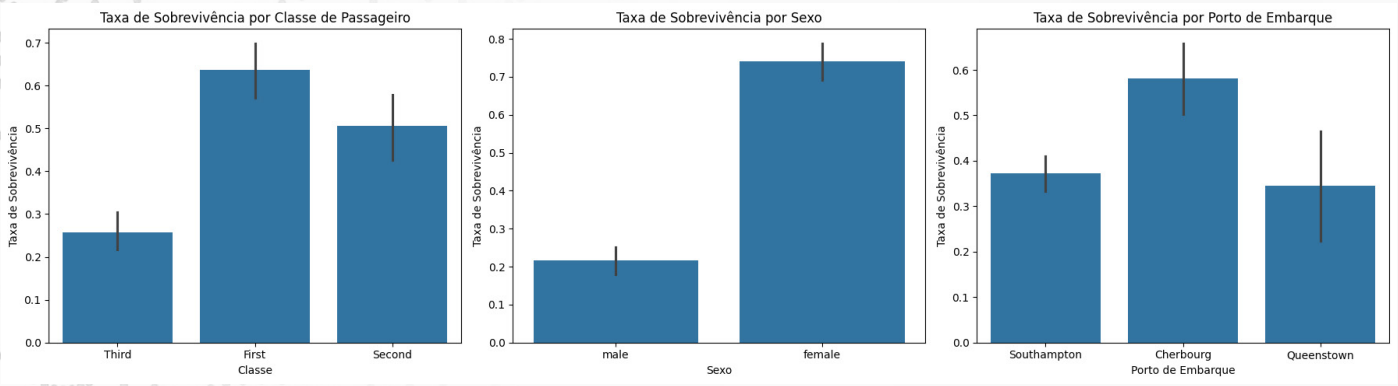
No exemplo a seguir, utilizamos subplots para agrupar três gráficos de barras (barplot) em disposição 1x3 (uma linha e 3 colunas). A variável axes recebe um *array* com os três espaços a serem utilizados pelos graficos em seguida (utilizando o parametro ax). O primeiro apresenta a taxa de sobrevivência por classe, no segundo tem-se a taxa por sexo e, por fim, a taxa por ponto de embarque. Esse tipo de gráfico apresenta o desvio padrão por meio de uma linha na vertical em cada barra. Com isso, é possível observar, por exemplo, que o desvio padrão para o ponto de embarque em Queenstown foi maior que o desvio padrão do ponto em Southampton.

```
# Análise de proporções de sobrevivência por classe, sexo e porto de embarque
fig, axes = plt.subplots(1, 3, figsize=(18, 5))

# Sobrevivência por Classe
sns.barplot(x='class', y='survived', data=titanic, ax=axes[0])
axes[0].set_title('Taxa de Sobrevivência por Classe de Passageiro')
axes[0].set_ylabel('Taxa de Sobrevivência')
axes[0].set_xlabel('Classe')

# Sobrevivência por Sexo
sns.barplot(x='sex', y='survived', data=titanic, ax=axes[1])
axes[1].set_title('Taxa de Sobrevivência por Sexo')
axes[1].set_ylabel('Taxa de Sobrevivência')
axes[1].set_xlabel('Sexo')

# Sobrevivência por Porto de Embarque
sns.barplot(x='embark_town', y='survived', data=titanic, ax=axes[2])
```

continua

```
axes[2].set_title('Taxa de Sobrevivência por Porto de Embarque')
axes[2].set_ylabel('Taxa de Sobrevivência')
axes[2].set_xlabel('Porto de Embarque')

plt.tight_layout()
plt.show()
```

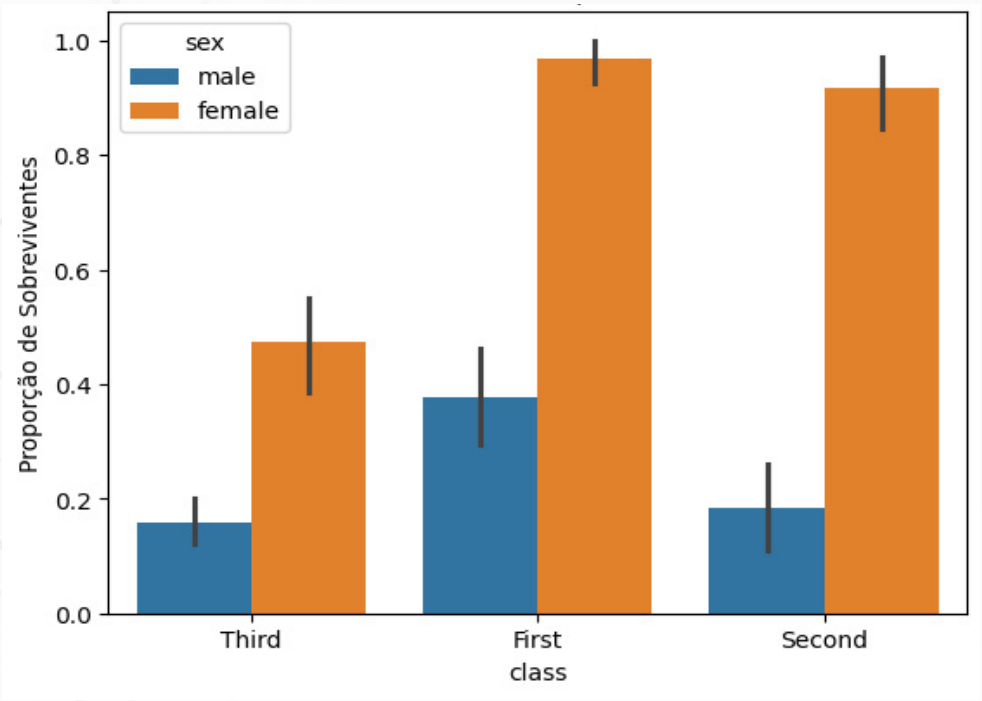
Figura 44 - Análise de proporções de sobrevivência por classe, sexo e porto de embarque



Fonte: autoria própria.

```
sns.barplot(x='class', y='survived', hue='sex', data=titanic)
plt.title('Taxa de Sobrevivência por Classe e Sexo')
plt.ylabel('Proporção de Sobreviventes')
plt.show()
```

Figura 45 - Taxa de Sobrevivência por Classe e Sexo



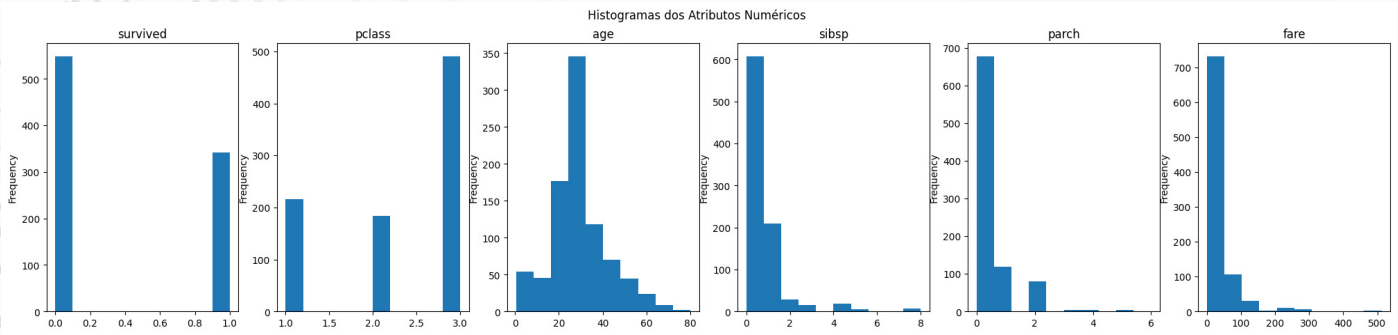
Fonte: autoria própria.

Visualizar Distribuição dos Dados (Atributos Numéricos)

No exemplo a seguir, é criado um gráfico para cada variável numérica do *dataset* (disponível em numericos). O método subplots é utilizado para criar uma disposição com uma linha e várias colunas (uma para cada coluna do *DataFrame* numericos). Em seguida, uma estrutura de repetição do tipo for percorre todas as colunas do *DataFrame*, plotando um gráfico do tipo histograma para cada uma delas. Nesse exemplo, o método plot do *DataFrame* foi utilizado, facilitando a construção de múltiplos gráficos.

```
fig, ax = plt.subplots(ncols=len(numericos.columns), nrows=1, figsize=(25, 5))
plt.suptitle("Histogramas dos Atributos Numéricos")

# Histograma para cada atributo numérico
for i in range(0, len(numericos.columns)):
    feature = numericos.columns[i]
    ax[i].set_title(feature)
    numericos[feature].plot(kind='hist', ax=ax[i])
```

Figura 46 - Histogramas dos Atributos Numéricos



Fonte: autoria própria.

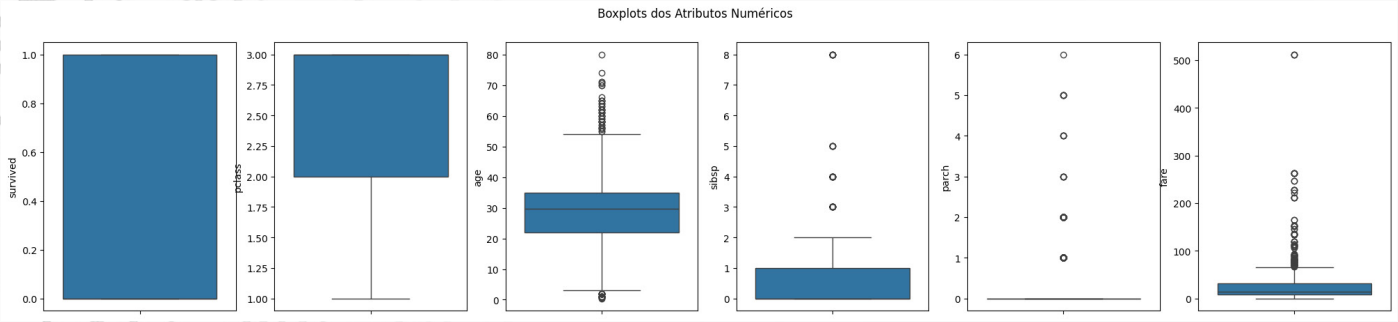
Visualizar a Presença de Outliers (Atributos Numéricos)

Outliers são valores que diferem significativamente do resto dos dados em um conjunto de observações. Eles podem ocorrer devido a erros de medição, variações genuínas nos dados ou falhas no processo de coleta de dados. Identificar e entender *outliers* é crucial porque eles podem distorcer análises estatísticas, como médias e regressões, levando a interpretações errôneas ou conclusões enganosas. Em contextos práticos, analisar *outliers* ajuda a aprimorar os modelos, garantindo que eles sejam robustos e representativos da realidade observada, ou mesmo revelar *insights* valiosos sobre comportamentos atípicos ou casos extremos.

De maneira similar ao exemplo anterior, no código a seguir, vários graficos são organizados em uma única imagem utilizando o método subplots. Neste caso, são criados gráficos do tipo boxplot para cada coluna numérica. O parâmetro orient='vertical' define a orientação dos gráficos.

```
fig, ax = plt.subplots(ncols=len(numericos.columns), nrows=1, figsize=(25, 5))
plt.suptitle("Boxplots dos Atributos Numéricos")

# Gráfico para cada atributo numérico
for i in range(0, len(numericos.columns)):
    feature = numericos.columns[i]
    sns.boxplot(numericos[feature], ax=ax[i], orient='vertical')
```

Figura 47 - *Boxplots* dos Atributos Numéricos



Fonte: autoria própria.

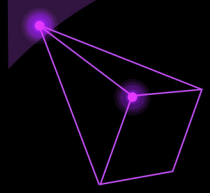
4.3 Saiba Mais...

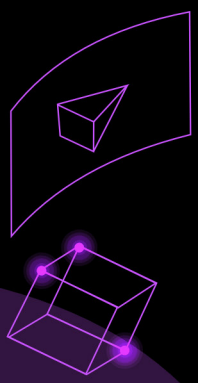
- *Dataset* oficial do Titanic
- *NumPy Team*
- *Pandas Development Team*
- *Matplotlib Developers*
- *Michael Waskom*

AKCIT
CENTRO DE COMPETÊNCIA EMBRAPII EM TECNOLOGIAS IMERSIVAS

Python para Processamento de Dados

Unidade V
Encerramento





Unidade V - Encerramento

Ao chegar ao fim deste Microcurso, ***Python* para Processamento de Dados**, pode-se recapitular alguns dos pontos principais abordados e refletir sobre as aplicações práticas desses conhecimentos. Além disso, oferecemos sugestões para ações futuras e leituras adicionais para aprofundar o entendimento e as habilidades dos(as) participantes.

5.1 Revisão dos Pontos Principais

- ***NumPy***: foi explorado como essa poderosa biblioteca pode ser utilizada para manipulação eficiente de *arrays* e matrizes, permitindo cálculos matemáticos complexos com alto desempenho.
- ***Pandas***: demonstrou-se a flexibilidade do *Pandas* em lidar com dados tabulares, desde a leitura e a limpeza de *datasets* até a realização de operações de agregação e filtragem, essenciais para a preparação e análise de dados.
- ***Matplotlib* e *Seaborn***: Os participantes tiveram a oportunidade de aprender como utilizar essas bibliotecas fundamentais para a visualização de dados, permitindo a criação de uma variedade de gráficos para interpretar e comunicar eficazmente os resultados das análises.
- **Estudo de caso:** o estudo de caso apresentado na Unidade IV consolida o conhecimento adquirido nas outras Unidades mediante um *dataset* real e análise de dados e visualizações relevantes. Espera-se que o(a) participante saia dessa Unidade preparado(a) para realizar análise de dados e visualizações por conta própria, utilizando as ferramentas apresentadas.

5.2 Reflexões Finais

A capacidade de processar e analisar dados não é apenas uma habilidade técnica; é uma competência crítica que pode ser aplicada em diversas áreas, desde a ciência até negócios e saúde pública. As ferramentas e técnicas discutidas neste Microcurso equipam você com o conhecimento necessário para explorar dados de maneira mais eficiente e tomar decisões informadas baseadas em análises robustas.

5.3 Sugestões para Ação Futura ou Leitura Adicional

1. **Prática contínua**: ao(à) participante é encorajada a aplicação do conteúdo aprendido em projetos de dados próprios. Quanto mais praticar, mais intuitiva será a manipulação e a análise de dados.
2. **Cursos avançados**: considere participar de cursos mais avançados em ciência de dados e aprendizado de máquina para construir sobre a base estabelecida neste Microcurso.
3. **Leitura adicional**:
   - "*Python for Data Analysis*", por Wes McKinney, criador do *Pandas*, é um excelente livro para aprofundar seus conhecimentos em *Pandas* e análise de dados com *Python*.
   - "*Data Science from Scratch*", por Joel Grus, é um livro que fornece uma boa introdução às teorias fundamentais da ciência de dados, com exemplos práticos em *Python*.

Esperamos que esse Microcurso tenha sido uma jornada enriquecedora para os(as) participantes e que as habilidades adquiridas aqui sirvam como um trampolim para seus futuros empreendimentos na área de análise de dados.

Boa sorte e continue explorando o vasto mundo dos dados!

Referências

DALE, K.. **Data visualization with Python and JavaScript**: scrape, clean, explore & transform your data. Sebastopol: O'Reilly Media, 2016. 589 p.

MCKINNEY, W.. **Python for data analysis**: data wrangling with Pandas, NumPy, and Python. 2. ed. Sebastopol: O'Reilly Media, 2017. 547 p.

OLIPHANT, T. E.. **Guide to NumPy**. 2. ed. CreateSpace Independent Publishing Platform, 2015. 364 p.

TOSI, S.. **Matplotlib for Python developers**: build remarkable publication quality plots the easy way. Birmingham: Packt Publishing, 2009. 293 p.

AKCIT

CENTRO DE COMPETÊNCIA EMBRAPII
EM TECNOLOGIAS IMERSIVAS

CEIA
CENTRO DE EXCELÊNCIA EM INTELIGÊNCIA ARTIFICIAL

GOVERNO DE GOIÁS
O ESTADO QUE DÁ CERTO

EMBRAPII

INF
INSTITUTO DE INFORMÁTICA

PRPI
PRÓ-REITORIA DE PESQUISA E INOVAÇÃO

UFG
UNIVERSIDADE FEDERAL DE GOIÁS

SOBRE O E-BOOK

Tipografia: Montserrat

Publicação: Cegraf UFG
Câmpus Samambaia, Goiânia - Goiás. Brasil. CEP 74690-900
Fone: (62) 3521-1358
https://cegraf.ufg.br