Digital College

FORMAÇÃO EM

# DATA ANALYTICS

MÓDULO 3:

## PYTHON PARA ANÁLISE DE DADOS

UNIDADE 2:

## PANDAS PARA ANÁLISE DE DADOS

# Sumário

# 1. Introdução ao Pandas

Pandas é uma biblioteca de código aberto em Python, amplamente utilizada para análise e manipulação de dados. Seu nome é derivado de "Panel Data" (dados em painel), que se refere a conjuntos de dados multidimensionais, geralmente obtidos por observações repetidas ao longo do tempo.

O Pandas fornece estruturas de dados flexíveis e de alto desempenho, como Séries e DataFrames, que permitem armazenar e manipular dados de forma eficiente. Ele é construído sobre a biblioteca NumPy e oferece recursos adicionais que facilitam o trabalho com dados tabulares e séries temporais.
As principais características do Pandas incluem:

- Estruturas de dados: o Pandas oferece dois principais tipos de estruturas de dados – Séries e DataFrames. A Série é uma estrutura unidimensional que pode armazenar diferentes tipos de dados, enquanto o DataFrame é uma estrutura bidimensional semelhante a uma tabela, composta por colunas e linhas;
- Manipulação de dados: o Pandas oferece uma ampla gama de recursos para manipular dados. Ele permite selecionar, filtrar, adicionar, remover e modificar dados com facilidade. Também oferece recursos avançados de indexação e alinhamento de dados;
- Limpeza de dados: o Pandas possui ferramentas para lidar com dados ausentes, duplicados e inconsistentes. Ele permite preencher valores ausentes, remover duplicatas, lidar com valores nulos e executar outras operações de limpeza e transformação de dados;
- Computação eficiente: o Pandas é construído com base na biblioteca NumPy, o que lhe confere um desempenho eficiente em operações numéricas e matemáticas. Ele também suporta operações vetorizadas, que podem ser aplicadas a todo o conjunto de dados de uma vez, em vez de processar elemento por elemento;

Em resumo, o Pandas é uma poderosa biblioteca em Python que facilita a análise, a manipulação e a transformação de dados, tornando-o uma escolha popular entre cientistas de dados, analistas e desenvolvedores.

## 1.1. Por que usar Pandas?

Existem várias razões pelas quais você deve considerar o uso do Pandas em suas tarefas de análise de dados. Aqui estão alguns motivos:

- Manipulação eficiente de dados: o Pandas oferece estruturas de dados otimizadas, como Séries e DataFrames, que são projetadas para lidar com grandes conjuntos de dados. Ele fornece métodos eficientes para realizar operações de filtragem, seleção, agregação, transformação e limpeza de dados;
- Facilidade de uso: o Pandas possui uma interface de alto nível que é intuitiva e fácil de aprender. Ele fornece uma sintaxe concisa e poderosa que permite realizar operações complexas de forma simples e eficiente. Isso torna o Pandas acessível tanto para iniciantes quanto para usuários avançados;
- Tratamento de dados ausentes: o Pandas oferece recursos robustos para lidar com valores ausentes em seus dados. Ele permite identificar, preencher, remover ou substituir esses valores de maneira flexível e conveniente. Isso é essencial ao lidar com dados reais que frequentemente apresentam valores ausentes;
- Integração com outras bibliotecas: o Pandas se integra perfeitamente com outras bibliotecas populares de análise de dados e visualização, como NumPy, Matplotlib, Seaborn e scikit-learn. Isso permite realizar análises avançadas, criar gráficos e visualizações sofisticadas, além de alimentar modelos de aprendizado de máquina com os dados processados pelo Pandas;

- Manipulação de séries temporais: o Pandas oferece recursos especializados para manipulação e análise de séries temporais. Ele permite realizar operações de indexação temporal, *resample*, agregação por período, cálculos de janelas móveis e muito mais. Esses recursos são valiosos em aplicações que envolvem dados temporais, como análise financeira, previsão de demanda, monitoramento de sensores, entre outros;
- Eficiência computacional: o Pandas é construído em cima da biblioteca NumPy, que é altamente eficiente para operações numéricas. Ele utiliza recursos de vetorização e otimizações internas para processar dados de forma rápida e eficiente. Isso é fundamental ao lidar com grandes volumes de dados e ao realizar operações computacionalmente intensivas;
- Compatibilidade com diversos formatos de dados: o Pandas suporta uma ampla variedade de formatos de dados, incluindo CSV, Excel, SQL, JSON, HDF5, entre outros. Isso facilita a importação e a exportação de dados de diferentes fontes e permite trabalhar com conjuntos de dados heterogêneos.

Esses são apenas alguns dos benefícios do uso do Pandas. No geral, o Pandas é uma ferramenta poderosa e versátil para análise e manipulação de dados em Python, tornando-o uma escolha popular entre cientistas de dados, analistas e pesquisadores.

## 1.2. Instalação do Pandas

**Para instalar o Pandas, siga as etapas abaixo:**

1. Verifique se você tem o Python instalado em sua máquina. Abra o terminal (no Windows, use o Prompt de Comando ou o PowerShell) e digite o seguinte comando:

```
python --version
```

2. Certifique-se de ter a versão correta do Python instalada (preferencialmente Python 3.x).

3. Instale o Pandas usando o pip: o pip é o gerenciador de pacotes padrão para o Python. Ele permite instalar pacotes e bibliotecas facilmente. Abra o terminal e digite o seguinte comando:

```
pip install pandas
```

O pip irá baixar e instalar o Pandas juntamente com suas dependências.

4. Verifique a instalação: para verificar se o Pandas foi instalado corretamente, você pode executar o seguinte comando no terminal:

```
python -c "import pandas as pd; print(pd.__version__)"
```

Isso importará o Pandas e exibirá a versão instalada.

5. Após a instalação bem-sucedida, você poderá importar o Pandas em seus scripts Python usando o seguinte comando:

```
import pandas as pd
```

Agora você está pronto para começar a usar o Pandas em seus projetos de análise de dados. Certifique-se de ter uma conexão com a Internet ativa durante o processo de instalação para permitir o download dos pacotes necessários.

# 2. Estrutura de dados do Pandas

O Pandas oferece duas principais estruturas de dados: Séries (Series) e DataFrames. Uma Série é uma estrutura de dados unidimensional que pode armazenar qualquer tipo de dado. Ela é semelhante a uma coluna em uma planilha ou a um array unidimensional. Cada elemento em uma Série possui um rótulo associado, chamado de índice.

Você pode criar uma Série a partir de uma lista, array NumPy, dicionário ou até mesmo de outro objeto do tipo Série. Aqui está um exemplo de como criar uma Série simples:

```
import pandas as pd

# Criando uma Série a partir de uma lista
nomes = ['Alice', 'Bob', 'Charlie', 'Dave']
idades = [25, 30, 35, 40]

serie_idades = pd.Series(idades, index=nomes)
print(serie_idades)
```

***Código 01***

Saída:

```
Alice      25
Bob        30
Charlie    35
Dave       40
dtype: int64
```

***Figura 01***

Neste exemplo, criamos uma Série chamada serie_idades, em que os nomes são os índices e as idades são os valores associados a cada nome.

Você pode acessar os elementos de uma Série usando seu índice:

```
print(serie_idades['Alice'])  # Acessando um elemento pelo rótulo do índice
print(serie_idades[0])  # Acessando um elemento pela posição do índice
```

***Código 02***

Um DataFrame é uma estrutura de dados bidimensional semelhante a uma tabela ou a uma planilha. Ele é composto por colunas e linhas, em que cada coluna pode representar uma variável e cada linha representa uma observação.

Os DataFrames são amplamente usados para análise de dados e fornecem muitos recursos para manipulação, filtragem, agregação e análise de dados tabulares.

Você pode criar um DataFrame a partir de várias fontes de dados, como dicionários, listas, arrays NumPy ou importando dados de arquivos CSV, Excel, SQL, dentre outros.

Aqui está um exemplo de criação de DataFrame a partir de um dicionário:

```
import pandas as pd

dados = {
    'Nome': ['Alice', 'Bob', 'Charlie', 'Dave'],
    'Idade': [25, 30, 35, 40],
    'Cidade': ['São Paulo', 'Rio de Janeiro', 'Belo Horizonte', 'Curitiba']
}

df = pd.DataFrame(dados)
print(df)
```

***Código 03***

Saída:

```
      Nome  Idade          Cidade
0    Alice     25       São Paulo
1      Bob     30  Rio de Janeiro
2  Charlie     35  Belo Horizonte
3     Dave     40        Curitiba
```

***Figura 02***

Neste exemplo, criamos um DataFrame chamado “df” a partir de um dicionário, em que cada chave representa o nome de uma coluna e os valores são as informações para cada coluna.

Você pode acessar, filtrar e manipular os dados em um DataFrame usando várias operações e métodos disponíveis no Pandas.

Essas são apenas introduções às estruturas de dados do Pandas, e há muito mais recursos e funcionalidades para explorar.

# 2.1. Um pouco mais sobre DataFrames

Os DataFrames do Pandas fornecem uma ampla gama de operações que podem ser usadas para manipular e analisar dados de forma eficiente. Vou abordar algumas das operações mais comuns que você pode realizar em um DataFrame, tanto na teoria quanto na prática.

- **Acessando e visualizando dados:**
  - **head() e tail():** esses métodos são usados para visualizar as primeiras (por padrão, cinco) linhas ou as últimas linhas de um DataFrame.

```
df.head() # Exibe as cinco primeiras linhas do DataFrame
df.tail(10) # Exibe as últimas dez linhas do DataFrame
```

***Código 04***

  - **shape:** retorna uma tupla com as dimensões do DataFrame (número de linhas, número de colunas).

```
print(df.shape) # Exibe as dimensões do DataFrame
```

***Código 05***

  - **columns:** retorna uma lista com os nomes das colunas do DataFrame.

```
print(df.columns) # Exibe os nomes das colunas do DataFrame
```

***Código 06***

  - **info():** exibe informações sobre o DataFrame, incluindo o tipo de dados de cada coluna e o uso de memória.

```
df.info() # Exibe informações sobre o DataFrame
```

***Código 07***

- **Seleção e filtragem de dados:**
  - **Selecionar colunas:** você pode selecionar uma ou várias colunas do DataFrame usando a notação de colchetes ou o método "loc".

```
coluna1 = df['Nome'] # Seleciona a coluna 'Nome'
colunas2 = df[['Nome', 'Idade']] # Seleciona as colunas 'Nome' e 'Idade'
colunas3 = df.loc[:, 'Nome':'Idade'] # Seleciona as colunas 'Nome' até 'Idade'
```

***Código 08***

  - **Filtrar linhas com base em condições:** você pode filtrar o DataFrame para exibir apenas as linhas que atendem a determinadas condições.

```
df_filtrado = df[df['Idade'] > 30] # Filtra as linhas com idade maior que 30
```

***Código 09***

- **Manipulação de dados:**
  - **fillna():** preenche os valores ausentes (NaN) em um DataFrame com um valor específico.

```
df_preenchido = df.fillna(0) # Preenche os valores ausentes com zero
```

***Código 10***

  - **dropna():** remove as linhas ou colunas que contêm valores ausentes.

```
df_sem_nulos = df.dropna() # Remove as linhas com valores ausentes
```

***Código 11***

- **sort_values():** classifica o DataFrame com base em uma ou várias colunas.

```
df_ordenado = df.sort_values('Idade') # Classifica o DataFrame com base na coluna 'Idade'
```

***Código 12***

- **Agregação e sumarização:**
  - **groupby():** agrupa os dados com base em uma ou várias colunas e permite aplicar operações agregadas.

```
df_agrupado = df.groupby('Cidade').mean() # Calcula a média das colunas para cada cidade
```

***Código 13***

sum(), mean(), min(), max(): essas são algumas das funções de agregação.

# 3. Leitura e escrita de dados

O Pandas oferece várias opções para ler e escrever dados de diferentes fontes usando DataFrames. A seguir, serão apresentadas as principais operações teóricas de leitura e escrita de dados usando o Pandas.

**Leitura de dados:**

- CSV (Comma Separated Values): é um formato de arquivo muito comum para dados tabulares. O Pandas oferece a função read_csv() para ler dados de um arquivo CSV e criar um DataFrame;
- Excel: o Pandas também permite ler dados de arquivos Excel (.xls ou .xlsx) usando a função read_excel();
- Banco de dados SQL: é possível ler dados diretamente de um banco de dados SQL usando o Pandas. Para isso, você precisa estabelecer uma conexão com o banco de dados e executar uma consulta SQL.

**Escrita de dados:**

- CSV: para escrever um DataFrame em um arquivo CSV, você pode usar o método to_csv();
- Excel: para escrever um DataFrame em um arquivo Excel, você pode usar o método to_excel();
- Banco de dados SQL: o Pandas permite escrever dados de um DataFrame diretamente em uma tabela de um banco de dados SQL.

Essas são apenas algumas das opções disponíveis para ler e escrever dados com o Pandas. O Pandas oferece suporte a uma ampla gama de formatos de dados, como JSON, HTML, SQL, HDF5, entre outros. A documentação oficial do Pandas é um ótimo recurso para explorar todas as opções disponíveis.

# 3.1. Leitura de arquivos CSV

Para ler um arquivo CSV com o Pandas, você pode usar a função read_csv(). Aqui está um exemplo de como fazer isso:

```
import pandas as pd

# Lê o arquivo CSV e cria um DataFrame
df = pd.read_csv('dados.csv')
```

***Código 14***

No exemplo acima, assumimos que você tem um arquivo chamado "dados.csv" no diretório atual. O Pandas irá ler o arquivo e criar um DataFrame com os dados do CSV.

O método read_csv() oferece uma série de parâmetros opcionais para personalizar a leitura do arquivo CSV. Alguns dos parâmetros mais comumente usados são:

sep: especifica o caractere delimitador usado no arquivo CSV. Por padrão, o delimitador é uma vírgula, mas você pode alterá-lo para outro caractere, como ponto e vírgula ou tabulação.

```
df = pd.read_csv('dados.csv', sep=';')  # Lê o arquivo CSV com ponto
```

***Código 15***

**header:** especifica a linha que contém os nomes das colunas. Por padrão, o Pandas assume que a primeira linha contém os nomes das colunas, mas você pode especificar uma linha diferente usando o número do índice.

```
df = pd.read_csv('dados.csv', header=0) # A primeira linha contém os nomes das colunas
```

***Código 16***

**encoding:** especifica a codificação do arquivo CSV. Por padrão, o Pandas assume a codificação UTF-8, mas você pode alterá-la para outros tipos, como Latin-1 ou UTF-16, se necessário.

```
df = pd.read_csv('dados.csv', encoding='Latin-1') # Lê o arquivo CSV com codificação Latin-1
```

***Código 17***

Esses são apenas alguns exemplos dos parâmetros disponíveis para a função read_csv(). Você pode consultar a documentação oficial do Pandas para obter mais informações sobre todos os parâmetros e opções de leitura de arquivos CSV.

A função read_csv() do Pandas possui uma variedade de parâmetros para personalizar a leitura de um arquivo CSV. Aqui estão alguns dos parâmetros mais comuns:

- **sep:** define o caractere delimitador usado no arquivo CSV. O valor padrão é ','. Outros valores comuns incluem ';' e '\t' para tabulação;

- **header:** especifica a linha que contém os nomes das colunas. O valor padrão é 'infer', que indica que o Pandas deve inferir os nomes das colunas a partir da primeira linha. Outras opções incluem usar um número inteiro para indicar a linha ou uma lista de inteiros para indicar várias linhas de cabeçalho;

- **names:** permite especificar manualmente os nomes das colunas como uma lista de strings. Isso é útil quando o arquivo CSV não tem linha de cabeçalho;

- **index_col:** define a coluna que será usada como índice do DataFrame. O valor padrão é “None”, mas você pode especificar o nome ou o número da coluna que deseja usar como índice;

- **skiprows:** permite pular um número específico de linhas no início do arquivo CSV;

- **skipfooter:** permite pular um número específico de linhas no final do arquivo CSV;

- **na_values:** especifica valores que devem ser tratados como valores ausentes (NaN). Você pode fornecer uma string, uma lista de strings ou um dicionário de colunas e valores ausentes;

- **parse_dates:** indica quais colunas devem ser analisadas como datas. Isso é útil quando as colunas do arquivo CSV contêm informações de data e hora que precisam ser convertidas em objetos datetime;

- **dtype:** permite especificar o tipo de dados de cada coluna do DataFrame. Isso pode ser útil para economizar memória ou evitar a conversão automática de tipos pelo Pandas;

- **nrows:** especifica o número máximo de linhas a serem lidas do arquivo CSV;

- **encoding:** define a codificação do arquivo CSV. O valor padrão é 'utf-8', mas você pode alterá-lo para a codificação apropriada, como 'latin-1' ou 'utf-16'.

Esses são apenas alguns dos parâmetros disponíveis na função read_csv().

## 3.2. Escrita de dados em arquivos CSV

Para escrever um DataFrame em um arquivo CSV usando o Pandas, você pode utilizar o método to_csv(). Aqui está um exemplo de como fazer isso:

```
import pandas as pd
# Criar DataFrame de exemplo
data = {'Nome': ['João', 'Maria', 'Carlos'],
        'Idade': [25, 30, 35],
          'Cidade': ['São Paulo', 'Rio de Janeiro', 'Belo
Horizonte']}
df = pd.DataFrame(data)
# Escrever DataFrame em um arquivo CSV
df.to_csv('dados.csv', index=False)
```

***Código 18***

No exemplo acima, criamos um DataFrame df com três colunas: 'Nome', 'Idade' e 'Cidade'. Em seguida, utilizamos o método to_csv() para escrever o DataFrame em um arquivo CSV chamado 'dados.csv'. O parâmetro index=False indica que não queremos incluir o índice do DataFrame no arquivo CSV.

Além do parâmetro index, a função to_csv() também possui outros parâmetros opcionais que você pode utilizar:

- **sep:** especifica o caractere delimitador usado no arquivo CSV. O valor padrão é ',', mas você pode alterá-lo para outro caractere, como ';' ou '\t' (tabulação);
- **header:** define se os nomes das colunas devem ser incluídos no arquivo CSV. O valor padrão é "True", mas você pode definir como "False" se não quiser incluir os nomes das colunas;
- **encoding:** especifica a codificação do arquivo CSV. O valor padrão é 'utf-8', mas você pode alterá-lo para outras codificações, como 'latin-1' ou 'utf-16;
- **na_rep:** define a representação a ser usada para valores ausentes (NaN) no arquivo CSV;
- **date_format:** permite especificar o formato de data a ser usado para colunas do tipo data;
- **decimal:** especifica o caractere decimal a ser usado para números de ponto flutuante.

Esses são apenas alguns dos parâmetros disponíveis na função to_csv().

# 4. Seleção de dados

Para selecionar dados específicos em um DataFrame usando o Pandas, existem várias opções e métodos disponíveis. Vou apresentar algumas das formas mais comuns de selecionar dados.

## 4.1. Selecionando linhas e colunas

Utilizando colchetes: você pode selecionar uma ou várias colunas do DataFrame usando a notação de colchetes []. Basta passar o nome da coluna ou uma lista de nomes de colunas desejadas.

```
coluna1 = df['Nome'] # Seleciona a coluna 'Nome'
colunas2 = df[['Nome', 'Idade']] # Seleciona as colunas 'Nome' e 'Idade'
```

***Código 19***

**Utilizando o método loc:** o método loc permite selecionar colunas usando o nome da coluna.

```
coluna1 = df.loc[:, 'Nome'] # Seleciona a coluna 'Nome'
colunas2 = df.loc[:, ['Nome', 'Idade']] # Seleciona as colunas 'Nome' e 'Idade'
```

***Código 20***

**Utilizando o método loc:** o método loc também pode ser usado para selecionar linhas com base em um rótulo de índice ou condição

```
linha1 = df.loc[0] # Seleciona a primeira linha do DataFrame
linhas_filtradas = df.loc[df['Idade'] > 30] # Seleciona as linhas com idade maior que 30
```

***Código 21***

**Utilizando o método iloc:** o método iloc é usado para selecionar linhas com base em um índice numérico.

```
linha1 = df.iloc[0] # Seleciona a primeira linha do DataFrame
linhas2_4 = df.iloc[2:5] # Seleciona as linhas de índice 2 a 4 (exclusivo)
```

***Código 22***

**Selecionando dados condicionalmente:**

**Utilizando condições booleanas:** você pode usar condições booleanas para selecionar dados que atendam a determinadas condições.

```
dados_filtrados = df[df['Idade'] > 30] # Seleciona as linhas com idade maior que 30
```

***Código 23***

**Selecionando dados com base em valores específicos:**

**Utilizando o método isin():** o método isin() permite selecionar linhas com base em valores específicos em uma coluna.

```
dados_selecionados = df[df['Cidade'].isin(['São Paulo', 'Rio de Janeiro'])] # Seleciona as linhas com cidade igual a 'São Paulo' ou 'Rio de Janeiro'
```

***Código 24***

Essas são apenas algumas das formas de selecionar dados com o Pandas. O Pandas oferece muitos outros recursos e métodos de seleção, como seleção condicional em várias colunas, seleção por índices booleanos e muito mais.

## 4.2. Filtrando dados

Filtrar dados com o Pandas usando uma abordagem avançada pode envolver o uso de múltiplas condições, combinação de filtros e aplicação de funções personalizadas. Aqui estão algumas técnicas avançadas para filtrar dados usando o Pandas:

***Filtro com múltiplas condições:***

**Utilizando operadores lógicos:** você pode usar os operadores lógicos & (e) e | (ou) para combinar várias condições em uma única expressão.

```
# Seleciona as linhas com idade maior que 30 e cidade igual a 'São Paulo' ou 'Rio de Janeiro'
dados_filtrados = df[(df['Idade'] > 30) & ((df['Cidade'] == 'São Paulo') | (df['Cidade'] == 'Rio de Janeiro'))]
```

***Código 25***

***Filtro com base em padrões de texto:***

**Utilizando expressões regulares:** a função str.contains() permite filtrar dados com base em padrões de texto usando expressões regulares.

```
# Seleciona as linhas com nomes que começam com 'A' ou 'B'
dados_filtrados = df[df['Nome'].str.contains('^(A|B)')]
```

***Código 26***

***Filtro com base em funções personalizadas:***

**Utilizando a função apply():** a função apply() permite aplicar uma função personalizada a cada elemento de uma coluna e, em seguida, filtrar os dados com base nos resultados.

```
# Define uma função para verificar se a idade é um número par
def idade_par(x):
    return x % 2 == 0
# Filtra as linhas com idade par
dados_filtrados = df[df['Idade'].apply(idade_par)]
```

***Código 27***

***Filtrar com base em condições complexas:***

**Utilizando o método query():** o método query() permite escrever expressões mais complexas usando uma sintaxe semelhante a de uma linguagem de consulta.

```
# Seleciona as linhas com idade maior que 30 e cidade igual a 'São Paulo' ou 'Rio de Janeiro'
dados_filtrados = df.query('Idade > 30 and (Cidade == "São Paulo" or Cidade == "Rio de Janeiro")')
```

***Código 28***

Essas são apenas algumas técnicas avançadas para filtrar dados com o Pandas. O Pandas oferece uma ampla gama de recursos e métodos flexíveis para manipulação e filtragem de dados, permitindo que você execute operações complexas de forma eficiente.

## 4.3. Usando expressões booleanas para selecionar dados

O Pandas oferece uma poderosa funcionalidade para selecionar dados usando expressões booleanas. Você pode usar expressões booleanas para criar filtros e selecionar as linhas que atendem a determinadas condições. Aqui estão algumas formas de usar expressões booleanas para selecionar dados com o Pandas:

***Filtro direto:***

Usando uma expressão booleana diretamente dentro dos colchetes:

```
# Seleciona as linhas onde a idade é maior que 30
dados_filtrados = df[df['Idade'] > 30]
```

***Código 29***

***Combinando múltiplas condições:***

Usando operadores lógicos & (e) e | (ou) para combinar múltiplas condições:

```
# Seleciona as linhas onde a idade é maior que 30 e a cidade é igual a 'São Paulo' ou 'Rio de Janeiro'
dados_filtrados = df[(df['Idade'] > 30) & ((df['Cidade'] == 'São Paulo') | (df['Cidade'] == 'Rio de Janeiro'))]
```

***Código 30***

***Negando uma condição:***

Usando o operador ~ para negar uma condição:

```
# Seleciona as linhas onde a idade não é maior que 30
dados_filtrados = df[~(df['Idade'] > 30)]
```

***Código 31***

***Selecionando valores específicos com base em condições booleanas:***

Usando o método isin() para selecionar valores específicos em uma coluna:

```
# Seleciona as linhas onde a cidade é 'São Paulo' ou 'Rio de Janeiro'
dados_filtrados = df[df['Cidade'].isin(['São Paulo', 'Rio de Janeiro'])]
```

***Código 32***

***Utilizando funções booleanas:***

Usando funções booleanas, como any() e all(), para aplicar condições a um grupo de colunas:

```
# Seleciona as linhas onde todas as colunas numéricas são maiores que zero
dados_filtrados = df[df[['Coluna1', 'Coluna2', 'Coluna3']].all(axis=1) > 0]
```

***Código 33***

Essas são apenas algumas das maneiras de usar expressões booleanas para selecionar dados com o Pandas. O Pandas oferece muitas outras opções e recursos para filtragem de dados com base em expressões booleanas.

# 5. Operações com dados

O Pandas oferece uma variedade de operações que podem ser realizadas em dados, permitindo manipular, transformar e analisar os dados de forma eficiente.

## 5.1. Operações aritméticas

O Pandas permite realizar operações aritméticas entre colunas e séries de um DataFrame. Essas operações são aplicadas elemento a elemento, alinhando automaticamente os dados com base nos rótulos dos índices. Abaixo estão algumas das operações aritméticas comuns que podem ser realizadas com o Pandas:

- **Adição:**

```
df['Resultado'] = df['Coluna1'] + df['Coluna2'] # Adiciona os valores das colunas 'Coluna1' e 'Coluna2' e armazena o resultado na coluna 'Resultado'
```

***Código 34***

- **Subtração:**

```
df['Resultado'] = df['Coluna1'] - df['Coluna2'] # Subtrai os valores da coluna 'Coluna2' dos valores da coluna 'Coluna1' e armazena o resultado na coluna 'Resultado'
```

***Código 35***

- **Multiplicação:**

```
df['Resultado'] = df['Coluna1'] * df['Coluna2'] # Multiplica os valores das colunas 'Coluna1' e 'Coluna2' e armazena o resultado na coluna 'Resultado'
```

***Código 36***

- **Divisão:**

```
df['Resultado'] = df['Coluna1'] / df['Coluna2'] # Divide os valores da coluna 'Coluna1' pelos valores da coluna 'Coluna2' e armazena o resultado na coluna 'Resultado'
```

***Código 37***

- **Potenciação:**

```
df['Resultado'] = df['Coluna1'] ** df['Coluna2'] # Calcula a potência dos valores da coluna 'Coluna1' elevados aos valores da coluna 'Coluna2' e armazena o resultado na coluna 'Resultado'
```

***Código 38***

- **Operações com escalares:**

```
df['Resultado'] = df['Coluna1'] + 10 # Adiciona 10 a todos os valores da coluna 'Coluna1' e armazena o resultado na coluna 'Resultado'
df['Resultado'] = df['Coluna2'] * 2 # Multiplica todos os valores da coluna 'Coluna2' por 2 e armazena o resultado na coluna 'Resultado'
```

***Código 39***

É importante observar que as operações aritméticas no Pandas respeitam o alinhamento dos rótulos de índice, preenchendo com NaN (valores ausentes) onde não há correspondência nos rótulos dos índices. Além disso, o Pandas oferece métodos como add(), sub(), mul(), div(), que podem ser usados para realizar operações aritméticas mais complexas com controle de valores ausentes e especificação de parâmetros adicionais, como preenchimento de valores ausentes.

Por exemplo:

```
df['Resultado']     =     df['Coluna1'].add(df['Coluna2'], fill_value=0) # Realiza a adição das colunas 'Coluna1' e 'Coluna2', preenchendo valores ausentes com 0 e armazena o resultado na coluna 'Resultado'
```

***Código 40***

Essas são algumas das operações aritméticas com o Pandas. O Pandas oferece flexibilidade e facilidade na realização de cálculos aritméticos em dados tabulares, tornando a manipulação e análise de dados mais eficiente.

## 5.2. Operações lógicas

O Pandas oferece suporte a operações lógicas em DataFrames, permitindo filtrar dados com base em condições booleanas. Aqui estão algumas das operações lógicas mais comuns que podem ser realizadas com o Pandas:

- **Operadores de comparação:**

Igualdade (==):

```
df[df['Coluna1'] == 'Valor'] # Seleciona as linhas onde o valor da coluna 'Coluna1' é igual a 'Valor'
```

***Código 41***

Diferença (!=):

```
df[df['Coluna1'] != 'Valor'] # Seleciona as linhas onde o valor da coluna 'Coluna1' é diferente de 'Valor'
```

***Código 42***

Maior que (>), menor que (<), maior ou igual a (>=), menor ou igual a (<=):

```
df[df['Coluna1'] > 10] # Seleciona as linhas onde o valor da coluna 'Coluna1' é maior que 10
df[df['Coluna1'] <= 5] # Seleciona as linhas onde o valor da coluna 'Coluna1' é menor ou igual a 5
```

***Código 43***

- **Operadores lógicos:**

"E" lógico (&):

```
df[(df['Coluna1'] > 5) & (df['Coluna2'] < 10)] # Seleciona as linhas onde o valor da coluna 'Coluna1' é maior que 5 e o valor da coluna 'Coluna2' é menor que 10
```

***Código 44***

"Ou" lógico (|):

```
df[(df['Coluna1'] == 'Valor1') | (df['Coluna1'] == 'Valor2')] # Seleciona as linhas onde o valor da coluna 'Coluna1' é igual a 'Valor1' ou 'Valor2'
```

***Código 45***

Negação (~):

```
df[~(df['Coluna1'] == 'Valor')] # Seleciona as linhas onde o valor da coluna 'Coluna1' é diferente de 'Valor'
```

***Código 46***

Essas são apenas algumas das operações lógicas que podem ser realizadas com o Pandas. Você pode combinar essas operações lógicas para criar expressões booleanas mais complexas e aplicá-las na filtragem de dados do DataFrame. O Pandas oferece uma flexibilidade significativa para lidar com operações lógicas em dados tabulares.

# 5.3. Agregação de dados

A agregação de dados é uma operação comum no Pandas, que permite resumir e agrupar os dados com base em determinadas condições. O Pandas fornece várias funções de agregação que podem ser aplicadas em colunas ou grupos de dados. Aqui estão algumas das principais funções de agregação disponíveis no Pandas:

- **Funções de agregação básicas:**

Média (mean()):

```
df['Coluna1'].mean() # Calcula a média da coluna 'Coluna1'
```

***Código 47***

Soma (sum()):

```
df['Coluna2'].sum() # Calcula a soma da coluna 'Coluna2'
```

***Código 48***

Valor mínimo (min()):

```
df['Coluna3'].min() # Retorna o valor mínimo da coluna 'Coluna3'
```

***Código 49***

Valor máximo (max()):

```
df['Coluna4'].max() # Retorna o valor máximo da coluna 'Coluna4'
```

***Código 50***

Contagem (count()):

```
df['Coluna5'].count() # Conta o número de valores não nulos na coluna 'Coluna5'
```

*Código 51*

- **Funções de agregação por grupo:**

groupby() e agg():

```
df.groupby('Coluna1')['Coluna2'].mean() # Calcula a média da coluna 'Coluna2' agrupada pelos valores únicos da coluna 'Coluna1'
```

*Código 52*

agg() com múltiplas funções:

```
df.groupby('Coluna1')['Coluna2'].agg(['mean', 'sum', 'count']) # Calcula a média, a soma e a contagem da coluna 'Coluna2' agrupada pelos valores únicos da coluna 'Coluna1'
```

*Código 53*

agg() com funções personalizadas:

```
def minha_funcao(x):
    return x.max() - x.min()

df.groupby('Coluna1')['Coluna2'].agg(minha_funcao) # Aplica uma função personalizada à coluna 'Coluna2' agrupada pelos valores únicos da coluna 'Coluna1'
```

*Código 54*

- **Funções de agregação condicional:**

apply() com função condicional:

```
def minha_funcao(x):
    if x['Coluna2'] > 10:
        return x['Coluna1'] + x['Coluna2']
    else:
        return x['Coluna1'] - x['Coluna2']

df.apply(minha_funcao, axis=1) # Aplica uma função condicional a cada linha do DataFrame, retornando um resultado agregado
```

***Código 55***

Essas são apenas algumas das funções de agregação disponíveis no Pandas. A agregação de dados é uma etapa importante na análise de dados, permitindo resumir informações e obter insights sobre os dados. O Pandas oferece uma ampla gama de opções para realizar operações de agregação de maneira eficiente e flexível.

# 6. Limpeza e transformação de dados

A limpeza e a transformação de dados são etapas cruciais no processo de preparação de dados antes de realizar análises ou construir modelos de machine learning. Essas etapas visam garantir que os dados estejam em um formato adequado, livre de erros, inconsistências e valores ausentes, além de transformá-los de maneira que sejam mais adequados para a análise ou modelagem. Aqui estão alguns conceitos-chave relacionados à limpeza e transformação de dados:

***Limpeza de dados:***

- Remoção de dados duplicados: identificar e remover registros duplicados em um conjunto de dados;
- Tratamento de valores ausentes: preencher, remover ou estimar valores ausentes para evitar problemas durante a análise;
- Correção de erros de dados: identificar e corrigir erros nos dados, como erros de digitação ou formatação incorreta;
- Padronização de dados: garantir que os dados sigam uma única convenção de nomenclatura, formatação ou codificação;
- Lidar com outliers: identificar e lidar com valores atípicos que possam afetar negativamente as análises ou modelos.

***Transformação de dados:***

- Normalização: transformar os dados para uma escala comum, geralmente entre 0 e 1, para evitar a predominância de recursos com grandes variações;
- Codificação de variáveis categóricas: converter variáveis categóricas em representações numéricas adequadas para análise ou modelagem;
- Discretização: converter variáveis numéricas em categorias discretas para facilitar a análise ou criar modelos baseados em regras;
- Criação de novas variáveis: derivar novas variáveis a partir das existentes, com base em fórmulas, transformações ou operações específicas;
- Agregação de dados: agrupar dados em níveis mais altos de granularidade para obter estatísticas resumidas ou agregadas.

***Tratamento de formatos de dados:***

- Conversão de tipos de dados: converter os tipos de dados para formatos adequados, como datas, números inteiros ou strings;
- Formatação de dados: garantir que os dados estejam em uma estrutura e formato coerentes, como remover espaços em branco ou caracteres especiais indesejados.

Essas são apenas algumas das técnicas e dos conceitos envolvidos na limpeza e transformação de dados. O objetivo é garantir que os dados estejam prontos para análises subsequentes, melhorando a qualidade e confiabilidade dos resultados obtidos a partir dos dados. O Pandas oferece uma ampla gama de funcionalidades para auxiliar nessas tarefas de limpeza e transformação de dados de forma eficiente.

## 6.1. Tratamento de valores ausentes

O tratamento de valores ausentes é uma etapa importante na limpeza e preparação de dados. O Pandas oferece várias funções e métodos para lidar com valores ausentes em um DataFrame. Aqui estão algumas das principais técnicas para tratamento de valores ausentes com o Pandas:

***Identificar valores ausentes:***

isnull() e notnull(): verificar se cada elemento de um DataFrame é nulo ou não nulo, retornando um DataFrame booleano com o mesmo formato.

```
df.isnull() # Retorna um DataFrame booleano indicando valores ausentes
df.notnull() # Retorna um DataFrame booleano indicando valores não nulos
```

***Código 56***

***Remover valores ausentes:***

dropna(): remover linhas ou colunas que contenham valores ausentes.

```
df.isnull() # Retorna um DataFrame booleano indicando valores ausentes
df.notnull() # Retorna um DataFrame booleano indicando valores não nulos
```

***Código 57***

***Preencher valores ausentes:***

**fillna():** preencher valores ausentes com um valor específico, como uma constante, a média, a mediana ou algum valor calculado.

```
df.fillna(0) # Preenche todos os valores ausentes com 0
df.fillna(df.mean()) # Preenche os valores ausentes com a média de cada coluna
df['Coluna1'].fillna(df['Coluna1'].median(), inplace=True) # Preenche os valores ausentes na coluna 'Coluna1' com a mediana da coluna
```

***Código 58***

*Interpolação de valores ausentes:*

```
interpolate(): Preencher valores ausentes por meio de uma interpolação linear ou com base em métodos mais avançados.
df.interpolate() # Preenche valores ausentes usando interpolação linear
```

***Código 59***

***Marcar valores ausentes:***

**isna() e notna():** retornar um DataFrame booleano indicando valores ausentes ou não ausentes, mas preservando a estrutura original.

```
df.isna() # Retorna um DataFrame booleano indicando valores ausentes, mantendo a mesma estrutura do DataFrame original
df.notna() # Retorna um DataFrame booleano indicando valores não ausentes, mantendo a mesma estrutura do DataFrame original
```

***Código 60***

É importante escolher a técnica de tratamento de valores ausentes adequada para o contexto dos dados e do problema em questão. O Pandas oferece flexibilidade e opções para lidar com valores ausentes de forma eficiente.

## 6.2. Transformação de dados

A transformação de dados é uma etapa importante na preparação dos dados para análise ou modelagem. O Pandas oferece várias funções e métodos para realizar transformações nos dados de um DataFrame. Aqui estão algumas das principais técnicas de transformação de dados com o Pandas:

**Normalização:**

**apply():** aplicar uma função a cada elemento de uma coluna ou DataFrame.

```
df['Coluna1'] = df['Coluna1'].apply(lambda x: (x -
df['Coluna1'].min()) / (df['Coluna1'].max() -
df['Coluna1'].min()))
```

***Código 61***

***Codificação de variáveis categóricas*:**

**map():** mapear valores de uma coluna para outros valores usando um dicionário ou uma função.

```
mapeamento = {'Valor1': 0, 'Valor2': 1, 'Valor3': 2}
df['Coluna2'] = df['Coluna2'].map(mapeamento)
```

***Código 62***

*Discretização:*

```
cut(): Dividir valores numéricos em intervalos discretos.
df['Coluna3'] = pd.cut(df['Coluna3'], bins=3, labels=
['Baixo', 'Médio', 'Alto'])
```

***Código 63***

***Criação de novas variáveis:***

Operações aritméticas: realizar operações aritméticas entre colunas para criar uma nova coluna.

```
df['NovaColuna'] = df['Coluna1'] + df['Coluna2']
```

***Código 64***

***Agregação de dados:***

groupby(): agrupar dados com base em uma ou mais colunas e aplicar funções de agregação.

```
df_agregado = df.groupby('Coluna4').agg({'Coluna5': 'mean',
'Coluna6': 'sum'})
```

***Código 65***

***Tratamento de formatos de dados:***

astype(): converter o tipo dos dados de uma coluna.

```
df['Coluna7'] = df['Coluna7'].astype(int)
```

***Código 66***

Essas são apenas algumas das técnicas de transformação de dados disponíveis no Pandas. O Pandas oferece uma ampla variedade de funções e métodos que permitem manipular e transformar os dados de acordo com as necessidades específicas de análise ou modelagem.

## 6.3. Redimensionamento de dados

O redimensionamento de dados, também conhecido como dimensionamento de recursos, é uma etapa comum na preparação de dados para análise ou modelagem. O Pandas oferece métodos para redimensionar os dados de um DataFrame. Aqui estão algumas das principais técnicas de redimensionamento de dados com o Pandas:

***Normalização Min-Max:***

MinMaxScaler: redimensionar os valores de uma coluna para um intervalo específico, geralmente entre 0 e 1.

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
df['Coluna1'] =
scaler.fit_transform(df['Coluna1'].values.reshape(-1, 1))
```

***Código 67***

***Padronização:***

StandardScaler: padronizar os valores de uma coluna para ter média zero e desvio-padrão igual a um.

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
df['Coluna2'] =
scaler.fit_transform(df['Coluna2'].values.reshape(-1, 1))
```

***Código 68***

***Redimensionamento por Vetorização:***

```
apply() com uma função de redimensionamento:
df['Coluna3'] = df['Coluna3'].apply(lambda x: x / 1000)
```

***Código 69***

Redimensionamento de Dados Categóricos: codificação de variáveis categóricas usando técnicas como codificação one-hot ou codificação ordinal.

É importante aplicar o redimensionamento aos dados com base nas características específicas do conjunto de dados e do algoritmo que será utilizado. O redimensionamento pode ajudar a evitar a predominância de recursos com grandes variações e garantir que os dados estejam em uma escala adequada para a análise ou modelagem. O Pandas, em combinação com outras bibliotecas, como o scikit-learn, oferece uma variedade de técnicas de redimensionamento de dados para atender às necessidades específicas.

# Referências

CHEN, Daniel. **Análise de dados com Python e Pandas**. São Paulo: Novatec, 2018.

BANIN, Sérgio Luiz. **Python 3 - Conceitos e Aplicações - Uma Abordagem Didática [BV:MB]**. São Paulo: Érica, 2018.

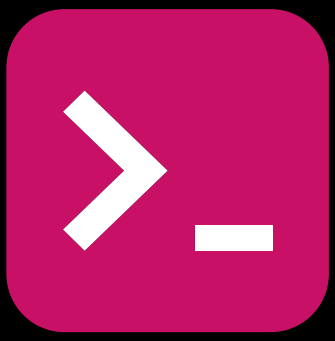
FORBELLONE, André L. V.; EBERSPACHER, Henri F**. Lógica de Programação: a construção de algoritmos e estruturas de dados [BV:PE]**. 3. ed. São Paulo: Editora Pearson, 2005.

LIMA, Janssen dos Reis. **Consumindo a API do Zabbix com Python [BV:PE]**. Rio de Janeiro: Editora Brasport, 2016.

PERKOVIC, Ljubomir. **Introdução à Computação Usando Python - Um Foco no Desenvolvimento de Aplicações [BV:MB]**. 1ª ed. Rio de Janeiro: LTC, 2016.
SEBESTA,Robert W. **Conceitos de Linguagens de Programação**. 11. ed. Porto Alegre: Grupo A, 2011.

TUCKER, Allen; NOONAN, Robert. **Linguagens de Programação: Princípios e Paradigmas**. Porto Alegre: Grupo A.

VLADISHEV, A. **Consumindo a API do Zabbix com Python**. Rio de Janeiro: Brasport.



# Digital College

ENSINO DE HABILIDADES DIGITAIS

@digitalcollegebr

/school/digitalcollegebr

/digitalcollegebr

digitalcollege.com.br